МИНИСТЕРСТВО НАУКИ И ВЫСШЕГО ОБРАЗОВАНИЯ
РОССИЙСКОЙ ФЕДЕРАЦИИ
федеральное государственное автономное образовательное учреждение
высшего образования «Санкт-Петербургский политехнический
университет Петра Великого»

Институт компьютерных наук и кибербезопасности

Высшая школа технологий искусственного интеллекта

Направление: 02.03.01 Математика и компьютерные науки

«Архитектура суперкомпьютерных систем»
Отчет по выполению лабораторной работы
Вариант 18

Студент,
группы 5130201/20101 _____ Тищенко А. А.

Преподаватель _____ Чуватов М. В.

«_____»_____ 2026г.

Санкт-Петербург, 2026

РЕФЕРАТ

ГЕТЕРОГЕННЫЕ ВЫЧИСЛИТЕЛЬНЫЕ СИСТЕМЫ, ПАРАЛЛЕЛЬНОЕ ВЫЧИСЛЕНИЕ, MPI, GPU-ВЫЧИСЛЕНИЯ, OPENMPI, HYPER-V, CUDA, ВРЕМЕННЫЕ РЯДЫ, АГРЕГАЦИЯ ДАННЫХ.

Объектом исследования в текущей работе является гетерогенный вычислительный кластер, использующий ресурсы GPU и CPU вычислителей. Так как ресурсы физических суперкомпьютерных систем зачастую не доступны, кластер такого рода самостоятельно создается с использованием доступных вычислительных ресурсов. Целью работы является создание, настройка и тестирование высокопроизводительного вычислительного кластера, способного эффективно выполнять задачи параллельных вычислений с использованием разнородных аппаратных ресурсов.

В разработанной системе воссоздается окружение суперкомпьютерного вычислителя, использующего разные узлы (виртуальные машины) и конфигурацию slurm.

На базе реализованного кластера разработано параллельное приложение, использующее технологии CUDA и OpenMPI, выполняющее анализ временных рядов исторических данных о стоимости Bitcoin с целью выявления интервалов значительного изменения цены на основе агрегированных дневных статистик.

Содержание

ТЕРМИНЫ И ОПРЕДЕЛЕНИЯ

1. **Гетерогенные вычислительные системы** — электронные системы, использующие различные типы вычислительных блоков. Они позволяют эффективно решать задачи за счёт использования компонентов с различными архитектурами.

2. **GPU (Graphics Processing Unit)** — графический процессор, предназначенный для параллельной обработки данных, особенно эффективен для вычислений с высокой степенью параллелизма.

3. **CPU (Central Processing Unit)** — центральный процессор общего назначения, оптимизированный для последовательных вычислений.

4. **Hyper-V** — технология виртуализации от Microsoft, позволяющая создавать и управлять виртуальными машинами на Windows.

5. **NFS (Network File System)** — протокол сетевой файловой системы, позволяющий разделять данные между узлами кластера.

6. **MPI (Message Passing Interface)** — стандарт взаимодействия между процессами в параллельных вычислительных системах.

7. **Slurm** — менеджер ресурсов и планировщик задач для кластерных систем.

8. **Контейнеризация** — технология виртуализации, которая позволяет изолировать программное обеспечение в контейнерах для повышения переносимости и устранения конфликтов версий.

9. **CUDA (Compute Unified Device Architecture)** — программная платформа от NVIDIA для разработки параллельных приложений на графических процессорах.

10. **OpenMPI** — высокопроизводительная реализация стандарта MPI, обеспечивающая взаимодействие между процессами в распределённых системах.

11. **MUNGE** — инструмент аутентификации, используемый для обеспечения безопасности в вычислительных кластерах.

12. **Rank** — идентификатор процесса в системе MPI, используемый для определения роли процесса.

13. **MPI_Send, MPI_Recv** — функции MPI для отправки и получения данных между процессами.

ПЕРЕЧЕНЬ СОКРАЩЕНИЙ И ОБОЗНАЧЕНИЙ

1. **GPU** — Graphics Processing Unit (графический процессор).
2. **CPU** — Central Processing Unit (центральный процессор).
3. **NFS** — Network File System (сетевая файловая система).
4. **MPI** — Message Passing Interface (интерфейс передачи сообщений).
5. **Slurm** — Simple Linux Utility for Resource Management (утилита управления ресурсами для Linux).
6. **CUDA** — Compute Unified Device Architecture (единая архитектура вычислений от NVIDIA).
7. **OpenMPI** — Open Message Passing Interface (реализация интерфейса передачи сообщений).
8. **MUNGE** — MUNGE Uid 'N' Gid Emporium (инструмент аутентификации для кластеров).
9. **HPC** — High-Performance Computing (высокопроизводительные вычисления).
10. **SSH** — Secure Shell (безопасная оболочка).
11. **RAM** — Random Access Memory (оперативная память).
12. **TCP/IP** — Transmission Control Protocol/Internet Protocol (протокол управления передачей/интернет-протокол).
13. **OS** — Operating System (операционная система).

ВВЕДЕНИЕ

Использование графических ускорителей (GPU) наряду с центральными процессорами (CPU) является распространенной практикой в области параллельных вычислений на гетерогенных платформах. Гетерогенные вычислительные системы объединяют различные типы вычислительных блоков, что позволяет эффективно решать задачи, выбирая оптимальный вычислительный ресурс для каждого этапа обработки данных.

GPU и CPU имеют различную архитектуру и изначально проектировались для решения разных классов задач. GPU обладает большим количеством простых вычислительных ядер, оптимизированных для массово-параллельных операций, в то время как CPU имеет меньше ядер, но с более сложной логикой и лучшей производительностью на одно ядро. Совместное использование GPU и CPU осложняется рядом особенностей: они имеют раздельную память, различные адресные пространства, и для передачи данных требуется явное копирование через системные вызовы. Однако при правильном распределении задач и росте объёма данных использование GPU может значительно ускорить вычисления.

Обеспечение взаимодействия между узлами вычислительного кластера осуществляется с помощью Message Passing Interface (MPI) — стандарта передачи сообщений между процессами. Основными реализациями являются Open MPI и MPICH, которые позволяют процессам синхронизироваться и обмениваться данными.

Для GPU-вычислений используются специализированные технологии, такие как CUDA (для устройств NVIDIA), ROCm (для устройств AMD) и OpenCL (кроссплатформенный стандарт). В данной работе используется CUDA Toolkit и библиотека CUB для эффективной параллельной обработки данных на GPU.

В качестве операционной системы в вычислительных кластерах традиционно используются Linux-дистрибутивы благодаря их производительности, открытости исходного кода и широкой поддержке необходимых технологий. В рамках текущей работы используется Ubuntu Server на виртуальных машинах.

Доступ к физическим суперкомпьютерным системам не всегда возможен, поэтому для разработки, тестирования и отладки параллельных приложений целесообразно создание собственного виртуального кластера. В данной работе используется технология виртуализации Hyper-V, разработанная Microsoft, которая позволяет создавать и управлять виртуальными машинами с возможностью проброса GPU-ресурсов. [2]

Задача анализа временных рядов больших объёмов данных является хорошим примером для демонстрации эффективности параллельных вычислений на гетерогенных системах. Обработка исторических данных о стоимости криптовалюты Bitcoin включает операции чтения больших файлов, агрегации данных по временным интервалам и поиска паттернов изменения цены — задачи, которые естественным образом поддаются распараллеливанию между узлами кластера и ускорению на GPU.

Целью данной работы является создание виртуального гетерогенного вычислительного кластера и разработка параллельного приложения для анализа временных рядов с эффективным использованием ресурсов как CPU, так и GPU узлов.

ПОСТАНОВКА ЗАДАЧИ

В рамках лабораторных работ необходимо выполнить следующие задачи:

1. Создание виртуальных машин с разнородными типами вычислительных ресурсов:
    - CPU-узлы;
    - GPU-узлы.
2. Настройка сети для связи хост-системы и виртуальных узлов;
3. Решение задачи по выбранному варианту:
    - Необходимо разработать параллельное приложение, задействующее вычислительные ресурсы CPU-узлов и CUDA-узлов, используя механизм OpenMPI, выполняющее на предоставленном наборе данных следующие действия. Задача разбита на 2 этапа, которые необходимо выполнить, используя разнородный тип вычислительных ресурсов;
    - Этап 1. Агрегация данных: для временного ряда исторических данных о стоимости Bitcoin (исходные данные содержат информацию по каждым 10 секундам) необходимо выполнить группировку по дням и для каждого дня вычислить среднюю цену как математическое ожидание значений Low и High, а также минимальные и максимальные значения Open и Close.
    - Этап 2. Поиск интервалов изменения цены: на основе дневных агрегированных данных необходимо выявить интервалы дат (начиная с начальной даты в наборе данных), в которых средняя дневная цена изменилась не менее чем на 10% относительно начала интервала. Для каждого интервала необходимо вывести начальную и конечную даты, а также минимальные и максимальные значения Open и Close за все дни внутри интервала.

Файл с исходными данными доступен в свободном доступе в сети интернет[1].

1 ОСНОВНАЯ ЧАСТЬ РАБОТЫ

1.1 Создание виртуального кластера

В рамках лабораторной работы необходимо создать виртуальный вычислительный кластер, использующий разнородный вид вычислителей: GPU и CPU. В рамках лабораторной работы необходимо создать виртуальные машины используя нативный механизм аппаратной виртуализации Windows: Hyper-V.

В качестве общей конфигурации виртуальной машины выбраны следующие параметры:

- Ubuntu Server 22.04.05 LTS;
- выделенное ОЗУ: 4096 MB;
- число виртуальных процессоров: 2;
- имя виртуальной машины: "tishcpuX "tishgpuX"(X - номер узла).

Создание виртуальной машины

Для создания виртуальной машины (узла) для будущего кластера, необходимо выполнить следующие шаги:

1. Открыть "Диспетчер Hyper-V".
2. Открыть меню для сервера (В текущей работе сервер: XSPMAIN).
3. Выбрать "Создать"→ "Виртуальная машина...". Для мастера создания виртуальной машины выбрать следующие параметры:
   (a) "Укажите имя и местанохождение"
      - Имя: `"tishcpu1"`;
      - Сохранить виртуальную машину: `V:\ Virtual machines`.
   (b) "Укажите поколение"
      - Выбрать `"Поколение 2"` (Далее этот параметр поменять невозможно).
   (c) "Выделить память":
      - "Память, выделяемое при запуске": `4096 МБ`;
      - "Использовать для этой виртуальной машины динамическую память": `Ставим галочку`.
   (d) "Настройка сети": пока данный раздел просто пропускается.
   (e) "Подключить виртуальный жесткий диск":
      - "Имя": tishcpu1.vhdx;
      - "Расположение": `V:\Virtual machines\Virtual Hard Disks\`;
      - "Размер": `100 ГБ`.
   (f) "Параметры установки":

- "Установить операционную систему из загрузочного образа": выбираем путь к `*.iso` файлу образа ОС.

После создания виртуальной машины необходимо настроить параметры:

1. В списке виртуальных машин в контекстном меню выбрать "Параметры...".
2. Далее необходимо настроить следующие параметры:
   - "Процессор"→"Число виртуальных процессоров": `2`;
   - "Безопасность"→"Включить безопасную загрузку": `Не ставим галочку`;
   - "SCSI-контроллер": DVD-дисковод;
   - "Сетевой адаптер"→"Виртуальный коммутатор": `Default Switch`;
   - "Автоматическое действие при запуске": `Ничего`;
   - "Память"→"Включить динамическую память": `Не ставим галочку`.
3. Нажать "Применить"→"Ок".

Для установки ОС необходимо запустить созданную ранее виртуальную машину. Для этого необходимо через консоль диспетчера Hyper-V и запустить ее:

Установка ОС

При установке ОС выполняются следующие шаги:

1. Выбор языка: `"English"`.
2. Для раскладки выбираем предлагаемую английскую раскладку.
3. Тип установки: `"Ubuntu Server"`.
4. Настройки сети пока не трогаем.
5. Настройки пространства памяти:
   (a) Выбираем опцию: `"Custom Storage Layout"`.
   (b) Для нашего свободного пространства: заводим swap и основное пространство памяти. (Размеры 50 и 4 ГБ)
6. Настраиваем параметры профиля системы:
   - "Your name": имя (например: `Arity`);
   - "Your server's name": `tishcpu1` (как у виртуальной машины);
   - "Pick a username": `arity`;
   - Пароль выбирается на свое усмотрение.

Создание пользователя root

При создании виртуальной машины, не был сконфигурирован пользователь root. Для его создания необходимо выполнить следующие команды:

```
aritytishcpu1:~$ sudo su -
root@tishcpu1:~# passwd
# Конфигурация пароля для пользователя root
```

1.2 Конфигурация пакетов

В рамках лабораторной работы необходимо установить следующие пакеты:

- libopenmpi3 - пакет для библиотеки Open MPI;
- slurmd - пакет для управляющего задачами демона;
- openssh-client - клиент OpenSSH;
- openssh-server - сервер OpenSSH.

Для этого необходимо выполнить следующие команды:

```
aritytishcpu1:~$ sudo apt update
aritytishcpu1:~$ sudo apt upgrade
aritytishcpu1:~$ sudo apt install libopenmpi3
aritytishcpu1:~$ sudo apt install slurmd
aritytishcpu1:~$ sudo apt install openssh-client
aritytishcpu1:~$ sudo apt install openssh-server
aritytishcpu1:~$ sudo apt clean
```

[3]

1. `apt update` - обновляет локальный индекс пакетов в системе, скачивая актуальную информацию о доступных пакетах из репозиториев, указанных в файлах `/etc/apt/sources.list` и `/etc/apt/sources.list.d/`. Это нужно для того, чтобы ОС знала о новых версиях пакетов и их зависимостях.
2. `apt upgrade` - обновляет установленные пакеты в системе. Предварительно рекомендуется обновить локальные индексы пакетов в системе.
3. `apt install <name>` - устанавливает в системе пакет с именем `<name>`.
4. `apt clean` - удаляет все загруженные архивы пакетов из кеша APT, освобождая место на диске.

Настройка сети на виртуальной машине

Для настройки сети на виртуальных машинах необходимо настроить следующие файлы:

1. `/etc/cloud/cloud.cfg.d/99-disable-network-config.cfg` - отключение управления сетевыми настройками через cloud-init, чтобы использовать другие способы конфигурации сети на системе.

2. `/etc/netplan/50-cloud-init.yaml` - используется для конфигурации сетевых настроек в системе через cloud-init, и обычно генерируется автоматически в облачных окружениях для настройки сети при запуске. Однако при настройке файла `.../99-disable-network-config.cfg` можно самостоятельно настроить сетевые параметры так, чтобы система не генерировала файл автоматически.

Конфигурация netplan:

```
root@tishgpu1:/home/arity# cat /etc/cloud/cloud.cfg.d/99-disable-network-config.cfg
network: {config: disabled}
```

Содержимое файла `/etc/netplan/50-cloud-init.yaml`:

```
aritytishcpu1:~$ sudo vim /etc/netplan/50-cloud-init.yaml
# This file is generated from information provided by the datasource.  Changes
# to it will not persist across an instance reboot.  To disable cloud-init's
# network configuration capabilities, write a file
# /etc/cloud/cloud.cfg.d/99-disable-network-config.cfg with the following:
# network: {config: disabled}
network:
    ethernets:
        eth0:
            addresses:
            - 10.200.166.125/24
            nameservers:
                addresses:
                - 8.8.8.8
                search:
                - tish
            routes:
            -   to: default
                via: 10.200.166.254
    version: 2
```

1. Интерфейс `eth0`:

   - `addresses: [10.200.166.125/24]`: Интерфейсу `eth0` присваивается IP-адрес `10.200.166.125` с маской подсети `255.255.255.0` (CIDR `/24`).
   - `routes:`
     - Указывает маршрут по умолчанию (default route), который направляет весь остальной трафик через шлюз `10.200.166.254`.
   - `nameservers:`
     - Настраиваются DNS-серверы: `8.8.8.8` (это публичные DNS от Google);
     - Указываем `tish` для параметра search.

2. Версия netplan:

   - `version: 2`: Указывает, что используется вторая версия формата Netplan.

1.3 Конфигурация сети

Для взаимодействия с виртуальными машинами с помощью хост-узла, необходимо пробросить порты на настроенные ранее адреса. Для этого необходимо воспользоваться следующими PowerShell Cmdlet'ами:

1. New-VMSwitch.
2. New-NetIPAddress.
3. New-NetNat.
4. Add-NetNatStaticMapping.

New-VMSwitch

`New-VMSwitch` - Создает новый виртуальный сетевой адаптер для виртуальных машин.

Принимаемые параметры:

- `SwitchName` - алиас для `Name`. Уточняет имя виртуального сетевого адаптера. Обязательный параметр; `SwitchType` - Уточняет тип создаваемого адаптера. Доступные значения для типа коммутатора — `Internal` (внутренний) и `Private` (частный). Чтобы создать `External` (внешний) виртуальный коммутатор, нужно указать либо параметр `NetAdapterInterfaceDescription`, либо `NetAdapterName`, что автоматически установит тип коммутатора как `External`. `Internal` и `Private` — это типы сетевых адаптеров, которые могут быть использованы для настройки виртуальных машин в Hyper-V:
  - `Internal` - позволяет виртуальной машине обмениваться данными с хостовой машиной и другими виртуальными машинами на том же хосте;
  - `External` - позволяет виртуальной машине общаться только с другими виртуальными машинами, но не имеет доступа к хосту или внешней сети;

  [4]

Создание виртуального сетевого адаптера в PowerShell:

```
PS C:\Windows\system32> New-VMSwitch -SwitchName "TishNet" -SwitchType Internal
```

New-NetIPAddress

`New-NetIPAddress` - Создает новый IP-адрес и привязывает его к указанному сетевому интерфейсу.

Принимаемые параметры:

- `InterfaceAlias` - Указывает имя сетевого интерфейса, к которому будет привязан новый IP-адрес. Обязательный параметр;
- `IPAddress` - Указывает IP-адрес, который нужно настроить. Обязательный параметр;

- `PrefixLength` - Указывает длину префикса подсети для IP-адреса. Например, для маски подсети `255.255.255.0` длина префикса равна `24`. Обязательный параметр;
- `DefaultGateway` - Указывает адрес шлюза по умолчанию, который будет использоваться для указанного IP-адреса. Необязательный параметр.

Пример настройки нового IP-адреса в PowerShell:

```
PS C:\Windows\system32> New-NetIPAddress -InterfaceAlias "vEthernet (TishNet)" -IPAddress 10.200.166.254 -
    PrefixLength 24
```

Параметры:

- `InterfaceAlias` - Имя интерфейса, например, `"vEthernet (TishNet)"`, к которому привязывается IP-адрес;
- `IPAddress` - Указанный IP-адрес (например, `10.200.166.254`);
- `PrefixLength` - Длина префикса подсети, например, `24` для `255.255.255.0`;
- `DefaultGateway` - Адрес шлюза по умолчанию (опционально).

[5]

Для проверки что IP-адресс корректно сконфигурирован можно воспользоваться команд-летом: `Get-NetIPAddress -AddressFamily IPv4 -InterfaceAlias "vEthernet (t1)"`

```
PS C:\Windows\system32> Get-NetIPAddress -AddressFamily IPv4 -InterfaceAlias "vEthernet (TishNet)"
IPAddress         : 10.200.166.254
InterfaceIndex    : 10
InterfaceAlias    : vEthernet (TishNet)
AddressFamily     : IPv4
Type              : Unicast
PrefixLength      : 24
PrefixOrigin      : Manual
SuffixOrigin      : Manual
AddressState      : Preferred
ValidLifetime     :
PreferredLifetime :
SkipAsSource      : False
PolicyStore       : ActiveStore
```

New-NetNat

`New-NetNat` - Создает новый NAT (Network Address Translation) для указанного интерфейса внутренней сети.

Принимаемые параметры:

- `Name` - Указывает имя NAT-объекта. Обязательный параметр;
- `InternalIPInterfaceAddressPrefix` - Указывает адресный префикс внутренней сети, который будет использоваться для NAT. Обязательный параметр;
- `ExternalIPInterfaceAddressPrefix` - Указывает адресный префикс внешней сети. Необязательный параметр;
- `Description` - Добавляет описание к NAT-объекту. Необязательный параметр.

Пример создания NAT в PowerShell:

```
PS C:\Windows\system32> New-NetNat -Name TishNat -InternalIPInterfaceAddressPrefix 10.200.166.0/24
```

Параметры:

- `Name` - Имя NAT-объекта, например, `TishNat`;
- `InternalIPInterfaceAddressPrefix` - Префикс внутренней сети, например, `10.200.166.0/24`.

```
PS C:\Windows\system32> New-NetNat -Name TishNat -InternalIPInterfaceAddressPrefix 10.200.166.0/24
Name                             : TishNat
ExternalIPInterfaceAddressPrefix :
InternalIPInterfaceAddressPrefix : 10.200.166.0/24
IcmpQueryTimeout                 : 30
TcpEstablishedConnectionTimeout  : 1800
TcpTransientConnectionTimeout    : 120
TcpFilteringBehavior             : AddressDependentFiltering
UdpFilteringBehavior             : AddressDependentFiltering
UdpIdleSessionTimeout            : 120
UdpInboundRefresh                : False
Store                            : Local
Active                           : True
```

[6]

Add-NetNatStaticMapping

`Add-NetNatStaticMapping` - Добавляет статическое сопоставление портов между внешними и внутренними адресами для NAT (Network Address Translation). Это позволяет направлять трафик, поступающий на внешний адрес и порт, к определенному внутреннему адресу и порту.

Принимаемые параметры:

- `NatName` - Указывает имя существующего объекта NAT, к которому применяется статическое сопоставление. Обязательный параметр;
- `ExternalIPAddress` - Указывает внешний IP-адрес, на который поступает входящий трафик. Для разрешения любых IP-адресов можно использовать `0.0.0.0/0`. Обязательный параметр;
- `ExternalPort` - Указывает порт внешнего IP-адреса, на который будет перенаправляться трафик. Обязательный параметр;
- `InternalIPAddress` - Указывает IP-адрес устройства внутри сети, к которому будет перенаправляться трафик. Обязательный параметр;
- `InternalPort` - Указывает порт устройства внутри сети, который будет использоваться для трафика. Обязательный параметр;
- `Protocol` - Указывает протокол (например, `TCP` или `UDP`), который будет использоваться для сопоставления. Обязательный параметр.

```
PS C:\Windows\system32> Add-NetNatStaticMapping -NatName TishNat -ExternalIPAddress 0.0.0.0/0 -ExternalPort
     22801 -InternalIPAddress 10.200.166.125 -InternalPort 22 -Protocol TCP
```

[7]

```
PS C:\Windows\system32> Add-NetNatStaticMapping -NatName TishNat -ExternalIPAddress 0.0.0.0/0 -ExternalPort
      22801 -InternalIPAddress 10.200.166.125 -InternalPort 22 -Protocol TCP


StaticMappingID               : 0
NatName                       : TishNat
Protocol                      : TCP
RemoteExternalIPAddressPrefix : 0.0.0.0/0
ExternalIPAddress             : 0.0.0.0
ExternalPort                  : 22801
InternalIPAddress             : 10.200.166.125
InternalPort                  : 22
InternalRoutingDomainId       : {00000000-0000-0000-0000-000000000000}
Active                        : True



PS C:\Windows\system32> Add-NetNatStaticMapping -NatName TishNat -ExternalIPAddress 0.0.0.0/0 -ExternalPort
      22802 -InternalIPAddress 10.200.166.126 -InternalPort 22 -Protocol TCP


StaticMappingID               : 1
NatName                       : TishNat
Protocol                      : TCP
RemoteExternalIPAddressPrefix : 0.0.0.0/0
ExternalIPAddress             : 0.0.0.0
ExternalPort                  : 22802
InternalIPAddress             : 10.200.166.126
InternalPort                  : 22
InternalRoutingDomainId       : {00000000-0000-0000-0000-000000000000}
Active                        : True



PS C:\Windows\system32> Add-NetNatStaticMapping -NatName TishNat -ExternalIPAddress 0.0.0.0/0 -ExternalPort
      22803 -InternalIPAddress 10.200.166.127 -InternalPort 22 -Protocol TCP


StaticMappingID               : 4
NatName                       : TishNat
Protocol                      : TCP
RemoteExternalIPAddressPrefix : 0.0.0.0/0
ExternalIPAddress             : 0.0.0.0
ExternalPort                  : 22803
InternalIPAddress             : 10.200.166.127
InternalPort                  : 22
InternalRoutingDomainId       : {00000000-0000-0000-0000-000000000000}
Active                        : True



PS C:\Windows\system32> Add-NetNatStaticMapping -NatName TishNat -ExternalIPAddress 0.0.0.0/0 -ExternalPort
      22804 -InternalIPAddress 10.200.166.128 -InternalPort 22 -Protocol TCP


StaticMappingID               : 5
NatName                       : TishNat
Protocol                      : TCP
RemoteExternalIPAddressPrefix : 0.0.0.0/0
ExternalIPAddress             : 0.0.0.0
ExternalPort                  : 22804
InternalIPAddress             : 10.200.166.128
InternalPort                  : 22
InternalRoutingDomainId       : {00000000-0000-0000-0000-000000000000}
```

```
Active                        : True
```

Для более быстрой настройки других виртуальных машин, можно скопировать tishcpu1, а при настройке первой tishgpu1 выполнить копирование в tishgpu2.

В результате создания виртуального кластеры были получены следующие виртуальные машины:

- tishcpu1 - "главный"вычислительный узел;
- tishcpu2;
- tishgpu1;
- tishgpu2.

1.4 Конфигурация ресурсов GPU

Для конфигурации ресурсов GPU для узлов, необходимо воспользоваться следующими PowerShell Cmdlet'ами:

1. Get-VMHostPartitionableGpu.
2. Add-VMGpuPartitionAdapter.
3. Set-VM.

Также необходимо установить драйвера GPU, CUDA-toolkit.

Get-VMHostPartitionableGpu

`Get-VMHostPartitionableGpu` - Возвращает список GPU, установленных на хосте Hyper-V, которые поддерживают разделение ресурсов (Partitioning). Эти GPU могут быть разделены и назначены виртуальным машинам для эффективного использования.

Принимаемые параметры:

- `-CimSession` (необязательный) - Позволяет указать удаленную сессию CIM (Common Information Model) для выполнения команды на другом компьютере. Если параметр не указан, команда выполняется локально.
- `-ThrottleLimit` (необязательный) - Ограничивает количество одновременных операций. Если параметр не указан, используется системное значение по умолчанию.

Выходные данные команды включают информацию о GPU, например:

- `Name` - Имя GPU;
- `TotalMemory` - Общий объем памяти GPU;
- `AvailableMemory` - Доступный объем памяти GPU;
- `Status` - Текущий статус GPU (например, `OK`).

Пример использования команды:

```
PS C:\Windows\system32> Get-VMHostPartitionableGpu
```

Результат выполнения:

```
PS C:\Windows\system32> Get-VMHostPartitionableGpu



Name                            : \\?\PCI#VEN_1002&DEV_164E&SUBSYS_D0001458&REV_C5#4&16012499&0&0041#{064092b3-625e-43bf-
                                  9eb5-dc845897dd59}\GPUPARAV
ValidPartitionCounts            : {32}
PartitionCount                  : 32
TotalVRAM                       : 1000000000
AvailableVRAM                   : 1000000000
SupportsIncomingLiveMigration : False
MinPartitionVRAM                : 0
MaxPartitionVRAM                : 1000000000
OptimalPartitionVRAM            : 1000000000
TotalEncode                     : 18446744073709551615
AvailableEncode                 : 18446744073709551615
MinPartitionEncode              : 0
MaxPartitionEncode              : 18446744073709551615
OptimalPartitionEncode          : 18446744073709551615
TotalDecode                     : 1000000000
AvailableDecode                 : 1000000000
MinPartitionDecode              : 0
MaxPartitionDecode              : 1000000000
OptimalPartitionDecode          : 1000000000
TotalCompute                    : 1000000000
AvailableCompute                : 1000000000
MinPartitionCompute             : 0
MaxPartitionCompute             : 1000000000
OptimalPartitionCompute         : 1000000000
CimSession                      : CimSession: .
ComputerName                    : XSPMAIN
IsDeleted                       : False

Name                            : \\?\PCI#VEN_10DE&DEV_2F04&SUBSYS_F3261569&REV_A1#740E0A73882DB04800#{064092b3-625e-43bf
                                  -9eb5-dc845897dd59}\GPUPARAV
ValidPartitionCounts            : {32}
PartitionCount                  : 32
TotalVRAM                       : 1000000000
AvailableVRAM                   : 1000000000
SupportsIncomingLiveMigration : False
MinPartitionVRAM                : 0
MaxPartitionVRAM                : 1000000000
OptimalPartitionVRAM            : 1000000000
TotalEncode                     : 18446744073709551615
AvailableEncode                 : 18446744073709551615
MinPartitionEncode              : 0
MaxPartitionEncode              : 18446744073709551615
OptimalPartitionEncode          : 18446744073709551615
TotalDecode                     : 1000000000
AvailableDecode                 : 1000000000
MinPartitionDecode              : 0
MaxPartitionDecode              : 1000000000
OptimalPartitionDecode          : 1000000000
TotalCompute                    : 1000000000
AvailableCompute                : 1000000000
MinPartitionCompute             : 0
MaxPartitionCompute             : 1000000000
OptimalPartitionCompute         : 1000000000
CimSession                      : CimSession: .
```

```
ComputerName                  : XSPMAIN
IsDeleted                     : False
```

[8] Эта команда является первым шагом в настройке GPU для виртуальных машин. Она позволяет проверить, какие GPU на хосте поддерживают разделение и сколько ресурсов доступно для выделения.

CUDA-ядра есть в `VEN_10DE&DEV`. Это графическое устройство с ядром Nvidia 5070.

Add-VMGpuPartitionAdapter

`Add-VMGpuPartitionAdapter` - Добавляет адаптер для разделения ресурсов GPU к указанной виртуальной машине (VM). Этот адаптер позволяет виртуальной машине использовать определенную часть вычислительных, графических и других ресурсов физического GPU.

Принимаемые параметры:

- `-VMName` - Указывает имя виртуальной машины, к которой будет добавлен GPU-адаптер. Обязательный параметр;
- `-InstancePath` - Указывает путь к конкретному GPU, который будет разделен для использования виртуальной машиной. Обязательный параметр;
- `-MinPartitionVRAM`, `-MaxPartitionVRAM`, `-OptimalPartitionVRAM` - Устанавливают минимальный, максимальный и оптимальный объем видеопамяти (VRAM), который будет доступен виртуальной машине;
- `-MinPartitionEncode`, `-MaxPartitionEncode`, `-OptimalPartitionEncode` - Устанавливают минимальное, максимальное и оптимальное количество ресурсов для кодирования видео;
- `-MinPartitionDecode`, `-MaxPartitionDecode`, `-OptimalPartitionDecode` - Устанавливают минимальное, максимальное и оптимальное количество ресурсов для декодирования видео;
- `-MinPartitionCompute`, `-MaxPartitionCompute`,

  `-OptimalPartitionCompute` - Устанавливают минимальное, максимальное и оптимальное количество вычислительных ресурсов (Compute), доступных виртуальной машине.

Пример использования:

```
PS C:\Windows\system32>  Add-VMGpuPartitionAdapter -VMName tishgpu1 -InstancePath "\\?\PCI#VEN_10DE&
    DEV_2F04&SUBSYS_F3261569&REV_A1#740E0A73882DB04800#{064092b3-625e-43bf-9eb5-dc845897dd59}\GPUPARAV" -
    MinPartitionVRAM 100000000 -MaxPartitionVRAM 1000000000 -OptimalPartitionVRAM 1000000000 -
    MinPartitionCompute 100000000 -MaxPartitionCompute 1000000000 -OptimalPartitionCompute 1000000000

PS C:\Windows\system32> Get-VMGpuPartitionAdapter -VMName tishgpu1


InstancePath                  : \\?\PCI#VEN_10DE&DEV_2F04&SUBSYS_F3261569&REV_A1#740E0A73882DB04800#{064092
    b3-625e-43bf
                                -9eb5-dc845897dd59}\GPUPARAV
SupportsOutgoingLiveMigration : False
CurrentPartitionVRAM          : 1000000000
MinPartitionVRAM              : 100000000
```

```
MaxPartitionVRAM              : 1000000000
OptimalPartitionVRAM          : 1000000000
CurrentPartitionEncode        : 1000000000
MinPartitionEncode            :
MaxPartitionEncode            :
OptimalPartitionEncode        :
CurrentPartitionDecode        : 1000000000
MinPartitionDecode            :
MaxPartitionDecode            :
OptimalPartitionDecode        :
CurrentPartitionCompute       : 0
MinPartitionCompute           : 100000000
MaxPartitionCompute           : 1000000000
OptimalPartitionCompute       : 1000000000
PartitionId                   : 0
PartitionVfLuid               : 050760292
Name                          : Параметры раздела GPU
Id                            : Microsoft:AE124752-47A6-4788-9C91-8DAE6D45A744\5B2FD022-36CF-4FD9-83D7-
    D5B60274737E
VMId                          : ae124752-47a6-4788-9c91-8dae6d45a744
VMName                        : tishgpu1
VMSnapshotId                  : 00000000-0000-0000-0000-000000000000
VMSnapshotName                :
CimSession                    : CimSession: .
ComputerName                  : XSPMAIN
IsDeleted                     : False
VMCheckpointId                : 00000000-0000-0000-0000-000000000000
VMCheckpointName              :
```

Set-VM

`Set-VM` - Изменяет параметры виртуальной машины (VM), включая настройки памяти, процессора и других ресурсов.

Принимаемые параметры:

- `-VMName` - Указывает имя виртуальной машины, для которой изменяются настройки. Обязательный параметр;
- `-GuestControlledCacheTypes` - Указывает, разрешено ли гостевой операционной системе управлять типами кэширования. Значение `$true` включает эту возможность;
- `-LowMemoryMappedIoSpace` - Устанавливает объем выделенного адресного пространства для низкоуровневого памяти, используемой устройствами, например, 3GB;
- `-HighMemoryMappedIoSpace` - Устанавливает объем выделенного адресного пространства для высокоуровневого памяти, например, 32GB;
- `-ProcessorCount` (необязательный) - Позволяет задать количество процессоров, доступных виртуальной машине;
- `-DynamicMemory` (необязательный) - Разрешает использование динамической памяти для виртуальной машины.

Пример использования команды:

```
Set-VM -VMName tishgpu1 -GuestControlledCacheTypes $true -LowMemoryMappedIoSpace 3GB -
    HighMemoryMappedIoSpace 32GB
```

Команда изменяет настройки виртуальной машины `tishgpu1`, разрешая гостевой ОС управлять типами кэширования и выделяя 3GB для низкоуровневого адресного пространства и 32GB для высокоуровневого адресного пространства.

[9]

Установка ПО

Для установки драйверов графического устройства необходимо скопировать их с хост-устройства с помощью `scp`.

Выполнение копирования драйверов GPU:

```
# Копируем содержимое папки с драйверами с хоста на виртуальную машину через SCP
# -r: копирование рекурсивно (включая подкаталоги)
# -P 22803: указание порта для подключения (22803)
scp -r -P 22803 C:\WINDOWS\System32\DriverStore\FileRepository\nv_dispi.inf_amd64_20ae8f14a487d5db
    arity@127.0.0.1:/tmp/

# Создаем директорию для драйверов в WSL (если еще не существует)
root@tishgpu1:/tmp# mkdir -p /usr/lib/wsl/drivers

# Переходим в созданную директорию
root@tishgpu1:/tmp# cd /usr/lib/wsl/drivers

# Проверяем содержимое директории (на данный момент она пуста)
root@tishgpu1:/usr/lib/wsl/drivers# ls

# Переходим на уровень выше, в директорию WSL
root@tishgpu1:/usr/lib/wsl/drivers# cd ..

# Проверяем содержимое директории /usr/lib/wsl (содержит только папку drivers)
root@tishgpu1:/usr/lib/wsl# ls drivers

# Создаем новую директорию lib в WSL (может быть нужна для других целей)
root@tishgpu1:/usr/lib/wsl# mkdir lib

# Проверяем, что теперь в /usr/lib/wsl есть две папки: drivers и lib
root@tishgpu1:/usr/lib/wsl# ls drivers lib

# Перемещаем скачанную папку драйвера из /tmp в директорию drivers
root@tishgpu1:/usr/lib/wsl# mv /tmp/nv_dispi.inf_amd64_20ae8f14a487d5db/ /usr/lib/wsl/drivers/

# Проверяем содержимое директории /usr/lib/wsl (папка drivers содержит перемещенные данные)
root@tishgpu1:/usr/lib/wsl# ls
drivers  lib

# Убеждаемся, что в папке drivers теперь находится папка с драйверами
root@tishgpu1:/usr/lib/wsl# ls drivers/
nv_dispi.inf_amd64_20ae8f14a487d5db

# Операция завершена, драйверы находятся в правильной директории
root@tishgpu1:/usr/lib/wsl#
```

Конфигурация драйверов на виртуальной машине:

```
# Копируем библиотеку lib из Windows на виртуальную машину через SCP
# -r: копирование рекурсивно
# -P 22803: указание порта для подключения
PS C:\Windows\system32> scp -r -P 22803 C:\Windows\System32\lxss\lib arity@127.0.0.1:/tmp/

# Перемещаем скопированную библиотеку в директорию WSL
root@tishgpu1:/usr/lib/wsl# mv /tmp/lib/ /usr/lib/wsl/

```

```
# Проверяем содержимое директории /usr/lib/wsl, чтобы убедиться, что библиотека перемещена
root@tishgpu1:/usr/lib/wsl# ls
drivers  lib

# Проверяем содержимое директории lib в WSL
root@tishgpu1:/usr/lib/wsl# ls lib
libcudadebugger.so.1  libd3d12core.so  libnvcuvid.so.1        libnvidia-ml.so.1           libnvwgf2umx.so
libcuda.so            libd3d12.so      libnvdxdlkernels.so    libnvidia-opticalflow.so    nvidia-smi
libcuda.so.1          libdxcore.so     libnvidia-encode.so    libnvidia-opticalflow.so.1
libcuda.so.1.1        libnvcuvid.so    libnvidia-encode.so.1  libnvoptix.so.1

# Устанавливаем права доступа на директорию WSL: только чтение и выполнение (555)
root@tishgpu1:/usr/lib# chmod -R 555 wsl/

# Устанавливаем владельцем директории WSL пользователя root и группу root
root@tishgpu1:/usr/lib# chown -R root:root wsl/

# Редактируем файл ld.so.conf.d для добавления пути к библиотекам WSL
root@tishgpu1:/usr/lib/wsl/lib# vim /etc/ld.so.conf.d/ld.wsl.conf

# Добавляем путь к библиотекам в файл ld.wsl.conf
/usr/lib/wsl/lib

# Применяем изменения с помощью ldconfig, чтобы обновить кэш динамических библиотек
root@tishgpu1:/usr/lib/wsl/lib# ldconfig
/sbin/ldconfig.real: /usr/lib/wsl/lib/libcuda.so.1 is not a symbolic link

# Редактируем или создаем файл /etc/profile.d/wsl.sh для добавления пути в системный PATH
root@tishgpu1:/usr/lib# vim /etc/profile.d/wsl.sh

# Проверяем содержимое файла wsl.sh, чтобы убедиться в правильности пути
root@tishgpu1:/usr/lib/wsl/lib# cat /etc/profile.d/wsl.sh
export PATH=$PATH:/usr/lib/wsl/lib

# Делаем файл wsl.sh исполняемым
root@tishgpu1:/usr/lib# chmod +x /etc/profile.d/wsl.sh
```

Сборка ядра с использованием готового shell скрипта:

```
# Загружаем и выполняем скрипт установки DXGKRNL (DirectX Kernel) через DKMS (Dynamic Kernel Module Support):
# - curl: утилита для загрузки данных из интернета.
#   -fsSL:
#     -f: завершить с ошибкой, если произошел сбой HTTP-запроса.
#     -s: тихий режим, скрывающий прогресс загрузки.
#     -S: отображение ошибок даже в тихом режиме.
#     -L: автоматическое следование за перенаправлениями.
# https://content.staralt.dev/dxgkrnl-dkms/main/install.sh: URL скрипта установки.
# |: передача загруженного содержимого скрипта как ввода следующей команде.
# sudo bash -es:
#   - bash: запускает загруженный скрипт в интерпретаторе команд bash.
#   - -e: завершает выполнение скрипта при любой ошибке.
#   - -s: интерпретирует данные, подаваемые через стандартный ввод, как сценарий bash.
aritytishgpu1:~$ curl -fsSL https://content.staralt.dev/dxgkrnl-dkms/main/install.sh | sudo bash -es

Target Kernel Version: 5.15.0-124-generic

Installing dependencies...
```

Для проверки корректности работы GPU на виртуальном узле можно воспользоваться утилитами:

- lspci;

– Используется для отображения списка PCI-устройств, подключенных к системе. Ключ -v выводит подробную информацию о каждом устройстве. В данном случае, команда подтверждает наличие GPU, который использует драйвер dxgkrnl;

- nvidia-smi;

  – Утилита для управления и мониторинга графических карт NVIDIA. Показывает информацию о версии драйвера, состоянии GPU, использовании памяти, температуре и запущенных процессах. В данном примере отображается статус GPU NVIDIA GeForce RTX 5070, использование 1543 MiB памяти и базовая загрузка GPU;

```
aritytishgpu1:~$ lspci -v
c556:00:00.0 3D controller: Microsoft Corporation Device 008e
        Physical Slot: 4111917767
        Flags: bus master, fast devsel, latency 0, NUMA node 0
        Capabilities: <access denied>
        Kernel driver in use: dxgkrnl
        Kernel modules: dxgkrnl


aritytishgpu1:~$ nvidia-smi
Sat Jan  3 17:25:15 2026
+-----------------------------------------------------------------------------------------+
| NVIDIA-SMI 580.102.01              Driver Version: 581.80         CUDA Version: 13.0     |
+-----------------------------------------+------------------------+----------------------+
| GPU  Name                 Persistence-M | Bus-Id          Disp.A | Volatile Uncorr. ECC |
| Fan  Temp   Perf          Pwr:Usage/Cap |           Memory-Usage | GPU-Util  Compute M. |
|                                         |                        |               MIG M. |
|=========================================+========================+======================|
|   0  NVIDIA GeForce RTX 5070        On  |   00000000:01:00.0  On |                  N/A |
|  0%   57C    P0             36W /  250W |    1543MiB /  12227MiB |      6%      Default |
|                                         |                        |                  N/A |
+-----------------------------------------+------------------------+----------------------+

+-----------------------------------------------------------------------------------------+
| Processes:                                                                              |
|  GPU   GI   CI              PID   Type   Process name                        GPU Memory |
|        ID   ID                                                               Usage      |
|=========================================================================================|
|  No running processes found                                                             |
+-----------------------------------------------------------------------------------------+
```

Для установки библиотеки для работы с CUDA необходимо выполнить следующие команды:

```
# Загружаем публичный ключ репозитория NVIDIA для подписи пакетов.
# Важно для обеспечения безопасности и проверки пакетов.
aritytishgpu1:~$ sudo apt-key adv --fetch-keys https://developer.download.
nvidia.com
/compute/cuda/repos/ubuntu2204/x86_64/3bf863cc.pub

# Внимание: apt-key устарел. Рекомендуется использовать новый метод управления ключами через директорию
    trusted.gpg.d.
# Ключ успешно импортирован:
# "cudatools <cudatools@nvidia.com>"

# Добавляем репозиторий CUDA в список источников пакетов.
# Это позволяет получать доступ к последним версиям инструментов CUDA.
```

```
aritytishgpu1:~$ sudo add-apt-repository "deb http://developer.download.nvidia.com/compute/
cuda/repos/ubuntu2204/x86_64/ /"

# Указанный репозиторий добавлен в файл /etc/apt/sources.list.d.
# Обновляем список пакетов для загрузки метаданных из нового репозитория.
# Получаем пакеты из репозитория NVIDIA и стандартных репозиториев Ubuntu.
# Замечание: ключ репозитория сохранён в устаревшем формате, как указано в предупреждении.

# Устанавливаем пакет CUDA Toolkit версии 12.
# Этот пакет включает библиотеки, компиляторы и утилиты для разработки с использованием GPU.
aritytishgpu1:~$ sudo apt install cuda-toolkit-12

# Создаём скрипт окружения для автоматической настройки переменных PATH, CUDA_HOME и LD_LIBRARY_PATH.
aritytishgpu1:/usr/local/cuda/bin$ sudo touch /etc/profile.d/cuda.sh

# Редактируем файл cuda.sh для добавления переменных окружения.
# Эти переменные позволяют системе находить бинарные файлы и библиотеки CUDA.
aritytishgpu1:/usr/local/cuda$ sudo vim /etc/profile.d/cuda.sh

# Проверяем содержимое файла, чтобы убедиться в правильной настройке переменных.
aritytishgpu1:/usr/local/cuda$ cat /etc/profile.d/cuda.sh
export PATH=$PATH:/usr/local/cuda/bin
export CUDA_HOME=$CUDA_HOME:/usr/local/cuda
export LD_LIBRARY_PATH=$LD_LIBRARY_PATH:/usr/local/cuda/lib64

# Делаем файл cuda.sh исполняемым, чтобы переменные окружения применялись при входе в систему.
aritytishgpu1:~$ sudo chmod +x /etc/profile.d/cuda.sh
```

1.5 Конфигурация NFS

NFS (Network File System) — это сетевой протокол, разработанный для предоставления общего доступа к файловым системам через сеть. С помощью NFS пользователи или приложения могут работать с файлами, расположенными на удалённом сервере, так, как будто они находятся на локальной машине. [10]

Для корректной работы исполняемых и других файлов на виртуальных машинах, вместо постоянного копирования можно воспользоваться протоколом сетевого доступа к файловой системе одной из машин. Для этого можно настроить NFS на одном из узлов (например tishgpu1) и далее выполнить монтирование в каталоги других виртуальных машин.

Настройка etc/hosts

Для удобной работы в среде виртуальных машин можно задать доменные имена в файле `etc/hosts`. Для этого его необходимо отредактировать на каждой виртуальной машине.

Формат создания доменного имени:

```
# IP-адрес, Основное имя хоста, Полное доменное имя
10.200.166.126 tishcpu2 tishcpu2.tish
```

Пример файла `etc/hosts`:

```
aritytishcpu1:~$ cat /etc/hosts
127.0.0.1 localhost
10.200.166.125 tishcpu1 tishcpu1.tish
10.200.166.126 tishcpu2 tishcpu2.tish
10.200.166.127 tishgpu1 tishgpu1.tish
```

```
10.200.166.128 tishgpu2 tishgpu2.tish

# The following lines are desirable for IPv6 capable hosts
::1     ip6-localhost ip6-loopback
fe00::0 ip6-localnet
ff00::0 ip6-mcastprefix
ff02::1 ip6-allnodes
```

```
# Содержимое файла /etc/exports на tishcpu1
# Экспортируем директорию /home/arity для общего доступа через NFS
# *: доступ разрешён для всех хостов
# rw: разрешение на чтение и запись
# nohide: дочерние файловые системы видны
# no_subtree_check: отключает проверку вложенных поддеревьев для повышения производительности
aritytishcpu1:~$ cat /etc/exports
/home/arity *(rw,nohide,no_subtree_check)

# Применяем изменения в настройках NFS (перезапускаем экспорт)
# -a: экспорт всех записей
# -r: перезапускает экспорт
# -v: подробный вывод
aritytishcpu1:~$ sudo exportfs -arv
exporting *:/home/arity

# Устанавливаем клиент NFS на tishcpu2
sudo apt install nfs-common

# Удаляем старую папку (если существует) для монтирования
aritytishcpu2:/mnt$ sudo rmdir share

# Проверяем содержимое директории /mnt
aritytishcpu2:/mnt$ ls

# Создаём новую папку share для монтирования NFS
aritytishcpu2:/mnt$ sudo mkdir share

# Монтируем экспортированную директорию /home/arity с tishgpu1 в локальную папку /mnt/share
# -t nfs: указывает тип файловой системы (NFS)
aritytishcpu2:/mnt$ sudo mount -t nfs tishgpu1:/home/arity /mnt/share

# Проверяем содержимое смонтированной директории
aritytishgpu2:/mnt$ ls share/
hello.txt  install.sh

# Читаем файл hello.txt из смонтированной директории
aritytishgpu2:/mnt$ cat share/hello.txt
happy hacking
aritytishgpu2:/mnt$
```

1.6 Конфигурация slurm

SLURM (Simple Linux Utility for Resource Management) — это открытая система управления задачами и ресурсами в кластерах. Она используется для распределения вычислительных задач между узлами и управления их выполнением.

С помощью `conf.html` файла конфигурации необходимо сформировать конфигурацию в `/etc/slurm/slurm.conf`.

- **ClusterName** - Имя кластера SLURM (например, "tish"), используется для идентификации кластера.

- **SlurmctldHost** - Имя хоста, на котором работает демон управления SLURM (slurmctld), в данном случае это `tishcpu1`.
- **MpiDefault** - Указывает стандартную реализацию MPI (по умолчанию `none`, т.е. MPI не используется).
- **ProctrackType** - Метод отслеживания процессов; `proctrack/cgroup` означает использование cgroup для изоляции процессов.
- **ReturnToService** - Указывает, должен ли узел автоматически возвращаться в работу после восстановления (`1` - включено).
- **SlurmctldPidFile** - Путь к файлу PID для демона slurmctld.
- **SlurmdPidFile** - Путь к файлу PID для демона slurmd.
- **SlurmdSpoolDir** - Директория, где slurmd хранит временные файлы и информацию о заданиях.
- **SlurmUser** - Имя пользователя, под которым запускаются процессы SLURM (обычно `slurm`).
- **StateSaveLocation** - Директория для сохранения состояния кластера, необходима для восстановления после перезапуска.
- **SwitchType** - Тип сетевого коммутатора; `switch/none` указывает, что коммутатор не используется.
- **TaskPlugin** - Плагин для управления задачами; `task/affinity` позволяет задавать привязку задач к CPU.
- **SchedulerType** - Тип планировщика задач; `sched/backfill` разрешает задачи меньшего размера выполняться параллельно с крупными.
- **SelectType** - Механизм выбора ресурсов;
- **SelectTypeParameters** - Параметры выбора ресурсов; `CR_Core` указывает распределение по ядрам.
- **JobAcctGatherType** - Метод сбора данных о выполнении задач;
- **SlurmctldLogFile** - Путь к файлу журнала для демона slurmctld.
- **SlurmdLogFile** - Путь к файлу журнала для демона slurmd.
- **NodeName** - Описание вычислительных узлов; описывает 4 узла с 2 CPU на каждом.
- **PartitionName** - Имя раздела (partition), включающего все узлы (`Nodes=ALL`); используется для распределения задач.
- **Default** - Указывает, что данный раздел является разделом по умолчанию (`YES`).
- **MaxTime** - Максимальное время выполнения задачи; `INFINITE` означает, что ограничений нет.

- **State** - Статус узлов или раздела (`UP` - узлы в рабочем состоянии).

Пример установки конфигурации:

```
aritytishcpu1:~$ sudo apt install slurm-wlm

aritytishcpu1:~$ nano /etc/slurm/slurm.conf
#Указываем:

# slurm.conf file generated by configurator easy.html.
# Put this file on all nodes of your cluster.
# See the slurm.conf man page for more information.
#
ClusterName=tish
SlurmctldHost=tishcpu1
#
#MailProg=/bin/mail
#MpiDefault=
#MpiParams=ports=#-#
ProctrackType=proctrack/cgroup
ReturnToService=2
SlurmctldPidFile=/var/run/slurmctld.pid
#SlurmctldPort=6817
SlurmdPidFile=/var/run/slurmd.pid
#SlurmdPort=6818
SlurmdSpoolDir=/var/spool/slurmd
SlurmUser=slurm
#SlurmdUser=root
StateSaveLocation=/var/spool/slurmctld
#SwitchType=
TaskPlugin=task/affinity,task/cgroup
#
#
# TIMERS
#KillWait=30
#MinJobAge=300
#SlurmctldTimeout=120
#SlurmdTimeout=300
#
#
# SCHEDULING
SchedulerType=sched/backfill
SelectType=select/cons_tres
#
#
# LOGGING AND ACCOUNTING
#AccountingStorageType=
#JobAcctGatherFrequency=30
#JobAcctGatherType=
#SlurmctldDebug=info
SlurmctldLogFile=/var/log/slurmctld.log
#SlurmdDebug=info
SlurmdLogFile=/var/log/slurmd.log
#
#
# COMPUTE NODES
NodeName=tishcpu[1-2],tishgpu[1-2] CPUs=2 State=UNKNOWN
PartitionName=tishpartition Nodes=ALL Default=YES MaxTime=INFINITE State=UP

#Копируем этот конфиг в машины tishcpu2, tishgpu1, tishgpu2
```

1.7 Конфигурация munge

MUNGE — это инструмент для аутентификации, который используется для обеспечения безопасности в кластерах.

Ниже приведены шаги конфигурации ключей munge для конфигурации машин.

```
# Копируем файл ключа MUNGE с узла tishgpu1 на все машины кластера.
# Этот файл необходим для аутентификации в кластере.

root@tishcpu2:/tmp# chown munge:munge ./munge.key
# Изменяем владельца и группу файла ключа на пользователя и группу MUNGE.
# Это необходимо для корректной работы службы MUNGE.

root@tishcpu2:/tmp# ls -l
# Проверяем содержимое текущей директории, чтобы убедиться, что файл ключа имеет нужные права:
# -rw-------: доступ только для владельца.

root@tishcpu2:/tmp# mv /etc/munge/munge.key /etc/munge/munge_old.key
# Переименовываем существующий ключ в ‘munge_old.key‘ для резервного копирования.

root@tishcpu2:/tmp# mv munge.key /etc/munge/munge.key
# Перемещаем новый файл ключа в директорию /etc/munge и задаём ему правильное имя.

root@tishcpu2:/etc/munge# ls
# Проверяем содержимое директории /etc/munge:
# Убедились, что есть два файла: новый ключ (‘munge.key‘) и резервный ключ (‘munge_old.key‘).

# Переходим на другой узел (tishgpu1), повторяем процесс.
aritytishgpu1:/tmp$ su
Password:
root@tishgpu1:/tmp# mv /etc/munge/munge.key /etc/munge/munge_old.key
# Создаём резервную копию существующего ключа.

root@tishgpu1:/tmp# chown munge:munge munge.key
# Изменяем владельца нового ключа на пользователя MUNGE.

root@tishgpu1:/tmp# mv /tmp/munge.key /etc/munge/munge.key
# Перемещаем новый ключ в директорию /etc/munge.

# На узле tishgpu2 повторяем процесс.
root@tishgpu2:/tmp# chown munge:munge munge.key
# Меняем владельца файла ключа.

root@tishgpu2:/tmp# mv /etc/munge/munge.key /etc/munge/munge_old.key
# Резервируем старый ключ.

root@tishgpu2:/tmp# mv munge.key /etc/munge/munge.key
# Перемещаем новый ключ на место старого.

root@tishgpu2:/tmp# ls -l /etc/munge
# Проверяем права и владельца ключей в директории /etc/munge:
# Убедились, что оба ключа принадлежат пользователю ‘munge‘ и имеют доступ только для владельца.

# Вносим изменения в конфигурацию SLURM:
# Меняем механизм отслеживания процессов с ‘cgroup‘ на ‘linuxproc‘ в файле конфигурации /etc/slurm/slurm.
    conf.

# Перезапускаем службы, чтобы применить изменения:
sudo systemctl restart munge
# Перезапускаем службу MUNGE для применения нового ключа.

sudo systemctl restart slurmd
# Перезапускаем демон SLURM для рабочих узлов.

```

```
sudo systemctl restart slurmctld
# Перезапускаем демон SLURM для управляющего узла.

sudo systemctl status munge
sudo systemctl status slurmd
sudo systemctl status slurmctld
# Проверяем статус всех служб, чтобы убедиться в их корректной работе.
```

1.8 Конфигурация OpenMPI

OpenMPI (Open Message Passing Interface) — это высокопроизводительная, гибкая и открытая реализация стандарта MPI (Message Passing Interface). MPI — это стандарт для взаимодействия между процессами в распределённых и параллельных вычислительных системах, таких как кластеры и суперкомпьютеры.

Для обмена сообщениями узлы должны обменяться публичными ключами и каждый имел прямой доступ к друг другу через `ssh`.

Чтобы произвести обмен ключами необходимо выполнить следующие команды:

```
# Создаём новую пару SSH-ключей:
ssh-keygen

# Копируем публичный SSH-ключ на удалённый узел:
# - ssh-copy-id: утилита для добавления публичного ключа на удалённый сервер.
# - aritytishcpu1: имя пользователя и адрес узла, куда копируется ключ.
# После выполнения этой команды ключ будет добавлен в файл authorized_keys на удалённом сервере,
# что позволит подключаться по SSH без ввода пароля.
ssh-copy-id aritytishcpu1
```

Для установки библиотеки OpenMPI необходимо выполнить следующие пакеты:

```
sudo apt install openmpi-bin # установить на все узлы
sudo apt install openmpi-dev # установить на 1 узел где будет разработка приложения
```

1.9 Постановка задачи и прототип решения

Описание задачи

Необходимо разработать параллельное приложение, задействующее вычислительные ресурсы двух CPU-узлов и двух CUDA-узлов, используя механизм OpenMPI, выполняющее анализ временных рядов исторических данных о стоимости Bitcoin.

Задача разбита на два этапа:

1. **Этап 1. Агрегация данных**: для временного ряда исторических данных о стоимости Bitcoin (исходные данные содержат информацию по каждым 10 секундам) необходимо выполнить группировку по дням и для каждого дня вычислить среднюю цену как математическое ожидание значений Low и High, а также минимальные и максимальные значения Open и Close.

2. **Этап 2. Поиск интервалов изменения цены**: на основе дневных агрегированных данных необходимо выявить интервалы дат (начиная с начальной даты в наборе данных), в которых средняя дневная цена изменилась не менее чем на 10% относительно начала интервала. Для каждого интервала необходимо вывести начальную и конечную даты, а также минимальные и максимальные значения Open и Close за все дни внутри интервала.

Описание входных данных

В задаче используется файл в формате CSV с историческими данными о стоимости Bitcoin[1]. В качестве разделителя используется запятая. Во входном файле заданы следующие поля:

1. **Timestamp** - временная метка Unix в секундах.
2. **Open** - цена открытия за период.
3. **High** - максимальная цена за период.
4. **Low** - минимальная цена за период.
5. **Close** - цена закрытия за период.
6. **Volume** - объём торгов (может быть пустым).

Прототип решения на Python

Для отладки алгоритма был создан прототип на языке Python. Код прототипа представлен ниже:

```
import pandas as pd

# Загрузка данных
df = pd.read_csv("data.csv")
df['Timestamp'] = pd.to_datetime(df['Timestamp'], unit='s', utc=True)

# Вычисление средней цены
df['Avg'] = (df['Low'] + df['High']) / 2

```

```python
# Группировка по дням и агрегация
df_days = (
    df.groupby(df["Timestamp"].dt.date)
    .agg(
        Avg=("Avg", "mean"),
        OpenMin=("Open", "min"),
        OpenMax=("Open", "max"),
        CloseMin=("Close", "min"),
        CloseMax=("Close", "max"),
    )
    .reset_index()
)

# Поиск интервалов изменения цены на 10%
intervals = []
start_idx = 0
price_base = df_days.loc[start_idx, "Avg"]

for i in range(1, len(df_days)):
    price_now = df_days.loc[i, "Avg"]
    change = abs(price_now - price_base) / price_base

    if change >= 0.10:
        interval = df_days.loc[start_idx:i]

        intervals.append({
            "start_date": df_days.loc[start_idx, "Timestamp"],
            "end_date": df_days.loc[i, "Timestamp"],
            "min_open": interval["OpenMin"].min(),
            "max_open": interval["OpenMax"].max(),
            "min_close": interval["CloseMin"].min(),
            "max_close": interval["CloseMax"].max(),
            "start_avg": price_base,
            "end_avg": price_now,
            "change": change,
        })

        start_idx = i + 1
        if start_idx >= len(df_days):
            break
        price_base = df_days.loc[start_idx, "Avg"]

df_intervals = pd.DataFrame(intervals)
```

Увеличение объёма данных

Исходные данные содержат информацию по каждой минуте и имеют размер около 360 МБ. При тестировании параллельной реализации обработка таких данных занимала слишком мало времени, что не позволяло достоверно оценить эффективность параллельных вычислений и преимущества использования GPU.

Для решения этой проблемы был разработан скрипт `upsample.py`, выполняющий линейную интерполяцию данных. Алгоритм работы скрипта следующий:

1. Для каждой пары соседних записей $(t_1, o_1, h_1, l_1, c_1, v_1)$ и $(t_2, o_2, h_2, l_2, c_2, v_2)$ вычисляется временной интервал $\Delta t = t_2 - t_1$.

2. Интервал делится на $n = \Delta t / \text{step}$ равных частей, где step - новый временной шаг (10 секунд).

3. Для каждой промежуточной точки $i \in [0, n)$ вычисляются интерполированные значения с помощью линейной интерполяции:

$$\begin{aligned}
\alpha &= i/n \\
t_i &= t_1 + i \cdot \text{step} \\
o_i &= o_1 + (o_2 - o_1) \cdot \alpha \\
h_i &= h_1 + (h_2 - h_1) \cdot \alpha \\
l_i &= l_1 + (l_2 - l_1) \cdot \alpha \\
c_i &= c_1 + (c_2 - c_1) \cdot \alpha \\
v_i &= v_1 + (v_2 - v_1) \cdot \alpha
\end{aligned}$$

В результате применения интерполяции данные были преобразованы из формата "каждая минута"в формат "каждые 10 секунд что увеличило объём данных в 6 раз - с примерно 360 МБ до 2.3 ГБ. Такой объём данных позволяет наглядно продемонстрировать эффективность параллельных вычислений и преимущества использования GPU-ускорения.

1.10 Параллельная реализация на CPU

Проблема последовательной обработки

При профилировании первоначальной реализации было выявлено, что операция чтения и парсинга CSV-файла размером 2.3 ГБ занимает значительную часть времени выполнения программы. Последовательное чтение такого объёма данных на одном узле приводило к неэффективному использованию вычислительных ресурсов кластера, так как остальные узлы простаивали в ожидании данных.

Параллельное чтение с перекрытием

Для решения этой проблемы было реализовано параллельное чтение файла, при котором каждый MPI-ранк читает только свою часть файла. Алгоритм работает следующим образом:

1. **Вычисление диапазонов байт**: размер файла делится на части пропорционально долям, указанным в переменной окружения `DATA_READ_SHARES`. Для каждого ранка вычисляется диапазон байт $[\text{start}, \text{end})$, который он должен прочитать.

2. **Добавление перекрытия**: к каждому диапазону добавляется перекрытие размером `READ_OVERLAP_BYTES` (по умолчанию 128 КБ). Это необходимо для корректной обработки строк CSV, которые могут быть разделены на границах диапазонов:

$$\text{start}_{\text{adj}} = \max(0, \text{start} - \text{overlap})$$
$$\text{end}_{\text{adj}} = \min(\text{file\_size}, \text{end} + \text{overlap})$$

3. **Обработка границ строк**:

   - Ранк 0 пропускает заголовок CSV (первую строку) и начинает парсинг со второй строки.
   - Ранки $1 \ldots n-1$ пропускают неполную строку в начале своего диапазона, начиная парсинг с первого символа новой строки после `\n`.
   - Ранк $n-1$ (последний) читает до конца файла, остальные ранки заканчивают чтение на последнем символе `\n` перед концом диапазона.

Такой подход обеспечивает равномерное распределение нагрузки по чтению между узлами кластера и исключает дублирование или потерю данных на границах диапазонов.

Агрегация данных по периодам

После параллельного чтения каждый ранк имеет свой набор записей `Record`. Агрегация выполняется локально на каждом ранке:

1. Записи последовательно обрабатываются, для каждой записи вычисляется индекс периода:

$$\text{period} = \lfloor \text{timestamp} / \text{AGGREGATION\_INTERVAL} \rfloor$$

2. Для каждого периода накапливаются следующие статистики:

   - Сумма средних цен: $\sum_i (Low_i + High_i)/2$
   - Минимальное и максимальное значение Open: $\min(Open_i)$, $\max(Open_i)$
   - Минимальное и максимальное значение Close: $\min(Close_i)$, $\max(Close_i)$
   - Количество записей в периоде: *count*

3. При смене периода статистики сохраняются в структуру `PeriodStats`, и начинается накопление для следующего периода.

Агрегация может выполняться на CPU (последовательная обработка) или на GPU (параллельная обработка с использованием CUDA). При недоступности GPU автоматически выполняется fallback на CPU-версию.

Удаление граничных периодов

Из-за параллельного чтения с перекрытием первый и последний периоды каждого ранка могут содержать неполные данные. Например, если период охватывает временной интервал $[t_1, t_2)$, а ранк прочитал записи только начиная с $t_1 + \delta$, то статистики для этого периода будут искажены.

Для устранения этой проблемы применяется функция `trim_edge_periods`:

- Ранк 0 удаляет только последний период (первый период гарантированно полный, так как чтение начинается с начала файла).
- Ранки $1 \dots n-2$ удаляют первый и последний периоды.
- Ранк $n-1$ удаляет только первый период (последний период гарантированно полный, так как чтение идёт до конца файла).

Параллельный поиск интервалов изменения цены

После агрегации каждый ранк имеет список периодов `PeriodStats`, упорядоченных по времени. Для параллельного поиска интервалов используется следующий алгоритм:

1. **Приём данных от предыдущего ранка**: ранк $i > 0$ ожидает от ранка $i-1$ информацию о незавершённом интервале. Если предыдущий ранк передал начало интервала, текущий ранк продолжает его обработку.

2. **Локальная обработка периодов**: ранк последовательно обходит свои периоды, проверяя условие изменения цены:

$$\text{change} = \frac{|\text{avg}_{\text{current}} - \text{avg}_{\text{start}}|}{\text{avg}_{\text{start}}} \geq 0.10$$

   Если условие выполнено, интервал завершается и сохраняется в результаты. Начинается новый интервал.

3. **Передача данных следующему ранку**: если у ранка остался незавершённый интервал (не достигнуто изменение на 10%), информация о начале этого интервала передаётся ранку $i + 1$ через MPI.

Такой подход обеспечивает корректность параллельного поиска интервалов: интервалы, пересекающие границы данных между ранками, корректно обрабатываются через передачу состояния по цепочке.

Сбор результатов

После завершения локальной обработки ранк 0 собирает найденные интервалы от всех остальных ранков через MPI, сортирует их по времени начала и записывает в выходной файл `result.csv`.

1.11 GPU-ускорение агрегации данных

Общий алгоритм GPU-агрегации

Агрегация данных на GPU реализована в модуле `gpu_plugin.cu` и использует библиотеку CUB (CUDA Unbound) для эффективной параллельной обработки. Общий алгоритм состоит из следующих шагов:

1. **Копирование данных на GPU**: массивы timestamp, open, high, low, close копируются из оперативной памяти CPU в память GPU.

2. **Вычисление индексов периодов**: для каждой записи параллельно вычисляется индекс периода:

$$\text{period\_id}_i = \lfloor \text{timestamp}_i / \text{AGGREGATION\_INTERVAL} \rfloor$$

   Используется простое ядро с одномерной сеткой блоков.

3. **Run-Length Encoding (RLE)**: применяется операция RLE из библиотеки CUB для нахождения уникальных последовательных периодов и подсчёта количества записей в каждом периоде. На выходе получаем:

   - Массив уникальных периодов: $[\text{period}_0, \text{period}_1, \ldots, \text{period}_{n-1}]$
   - Массив длин последовательностей: $[\text{count}_0, \text{count}_1, \ldots, \text{count}_{n-1}]$

4. **Exclusive Scan**: применяется префиксная сумма (exclusive scan) к массиву длин для вычисления смещений начала каждого периода в исходном массиве данных:

$$\text{offset}_i = \sum_{j=0}^{i-1} \text{count}_j$$

5. **Агрегация по периодам**: для каждого периода параллельно вычисляются статистики (среднее значение, минимумы и максимумы). Используется одно из двух ядер в зависимости от настроек (см. следующий раздел).

6. **Копирование результатов обратно на CPU**: агрегированные статистики копируются из памяти GPU обратно в оперативную память CPU.

Два варианта ядер агрегации

Для агрегации по периодам реализовано два варианта CUDA-ядер, оптимизированных для разных сценариев использования:

1. Блочное ядро (Block Kernel):

Используется когда `USE_BLOCK_KERNEL=1`. Оптимизировано для случая, когда в каждом периоде много записей (большой `AGGREGATION_INTERVAL`).

Алгоритм работы:

- Один блок потоков обрабатывает один период.
- Потоки внутри блока параллельно обрабатывают записи периода, каждый поток накапливает локальные статистики.

- Используется shared memory для промежуточного хранения результатов.
- Атомарные операции (`atomicAdd`, `atomicMin`, `atomicMax`) используются для объединения локальных результатов потоков.
- Первый поток блока записывает финальный результат в глобальную память.

Преимущества: эффективное использование параллелизма внутри периода, минимизация обращений к глобальной памяти за счёт shared memory.

2. Простое ядро (Simple Kernel):

Используется когда `USE_BLOCK_KERNEL=0`. Оптимизировано для случая, когда периодов много, но в каждом периоде мало записей (малый `AGGREGATION_INTERVAL`).

Алгоритм работы:

- Один поток обрабатывает один период полностью.
- Поток последовательно обходит все записи своего периода, накапливая статистики.
- Не используется shared memory и атомарные операции.
- Результат сразу записывается в глобальную память.

Преимущества: отсутствие overhead на синхронизацию потоков и атомарные операции, эффективно при большом количестве независимых периодов.

Выбор ядра:

Выбор между ядрами осуществляется через переменную окружения `USE_BLOCK_KERNEL`:

- Блочное ядро предпочтительно при агрегации по дням/часам (86400 или 3600 секунд) - много записей в каждом периоде.
- Простое ядро предпочтительно при агрегации по минутам/секундам (60 или 10 секунд) - мало записей в каждом периоде, но много периодов.

1.12 Конфигурация через переменные окружения

Все настройки параллельного приложения вынесены в переменные окружения, которые задаются в SLURM-скрипте `run.slurm`. Это обеспечивает гибкость настройки без необходимости перекомпиляции программы.

Описание переменных окружения

- **DATA_PATH** - полный путь к CSV-файлу с входными данными. Файл должен быть доступен на всех узлах кластера (рекомендуется использовать общую директорию, например, через NFS).

  Пример: `/mnt/shared/supercomputers/data/data_10s.csv`

- **DATA_READ_SHARES** - доли данных для каждого ранка при параллельном чтении файла, разделённые запятыми. Позволяет неравномерно распределить нагрузку по чтению, если узлы имеют разную производительность.

  Пример: `10,11,13,14` означает, что файл будет разделён на части пропорционально 10 : 11 : 13 : 14. Если количество значений не совпадает с количеством ранков, используется равномерное распределение.

- **READ_OVERLAP_BYTES** - размер перекрытия в байтах при параллельном чтении файла. Необходим для корректной обработки строк CSV на границах диапазонов. Значение должно быть достаточным для размещения хотя бы одной полной строки CSV.

  Значение по умолчанию: `131072` (128 КБ)

- **AGGREGATION_INTERVAL** - интервал агрегации в секундах. Определяет размер временного периода, по которому группируются данные.

  Типичные значения:

  - `60` - агрегация по минутам
  - `600` - агрегация по 10 минутам
  - `3600` - агрегация по часам
  - `86400` - агрегация по дням

- **USE_CUDA** - флаг использования GPU для агрегации данных. Если установлен в `1`, программа попытается использовать GPU. Если GPU недоступен или флаг установлен в `0`, используется CPU-версия агрегации.

  Значения: `0` (отключено) или `1` (включено)

- **USE_BLOCK_KERNEL** - выбор варианта CUDA-ядра для GPU-агрегации (действует только при `USE_CUDA=1`). Определяет, какое ядро будет использоваться для параллельной обработки на GPU.

  Значения:

  - `0` - использовать простое ядро (один поток на период)
  - `1` - использовать блочное ядро (один блок на период)

Рекомендации по выбору:

- Для больших интервалов агрегации (дни, часы) - `USE_BLOCK_KERNEL=1`
- Для малых интервалов агрегации (минуты, секунды) - `USE_BLOCK_KERNEL=0`

Пример конфигурации в `run.slurm`:

```
export DATA_PATH="/mnt/shared/supercomputers/data/data_10s.csv"
export DATA_READ_SHARES="10,11,13,14"
export READ_OVERLAP_BYTES=131072
export AGGREGATION_INTERVAL=60
export USE_CUDA=1
export USE_BLOCK_KERNEL=0
```

1.13 Структура проекта

Исходный проект содержит в себе следующие зависимости:

- CUDA-Toolkit 12.8;
- OpenMPI 3.

Можно выделить следующие основные сущности:

- `run.slurm` - SLURM-скрипт для запуска параллельного приложения на 4 узлах с настройкой переменных окружения (путь к данным, доли данных для каждого ранка, интервал агрегации, использование CUDA). Исходный текст файла представлен в Приложение А;
- `Makefile` - файл системы сборки, описывающий компиляцию C++ исходников с помощью mpic++ и компиляцию CUDA-плагина с помощью nvcc, а также правила запуска и очистки. Исходный текст файла представлен в Приложение Б;
- `src/main.cpp` - основная MPI-программа: координирует выполнение параллельного чтения CSV данных, агрегацию данных по временным периодам (на CPU или GPU), поиск интервалов изменения цены и запись результатов. Исходный текст файла представлен в Приложение В;
- `src/csv_loader.cpp`, `src/csv_loader.hpp` - модуль параллельной загрузки CSV-файла: каждый MPI-ранк читает свою часть файла с перекрытием для обработки границ строк, парсит записи Bitcoin данных. Исходный текст файла представлен в Приложение Г;
- `src/aggregation.cpp`, `src/aggregation.hpp` - модуль агрегации временных рядов на CPU: группирует записи по временным интервалам, вычисляет среднее значение (Low+High)/2, минимумы и максимумы Open/Close за каждый период. Исходный текст файла представлен в Приложение Д;
- `src/gpu_loader.cpp`, `src/gpu_loader.hpp` - модуль динамической загрузки GPU-плагина: проверяет доступность GPU, загружает функции из `libgpu_compute.so`, преобразует данные из AoS в SoA для передачи на GPU. Исходный текст файла представлен в Приложение Е;
- `src/gpu_plugin.cu` - CUDA-модуль агрегации данных на GPU: использует библиотеку CUB для RLE и scan операций, реализует два ядра агрегации (блочное для больших интервалов и простое для множества малых периодов). Исходный текст файла представлен в Приложение Ж;
- `src/intervals.cpp`, `src/intervals.hpp` - модуль параллельного поиска интервалов изменения цены: каждый ранк обрабатывает свою часть периодов, передаёт незавершённые интервалы следующему ранку через MPI, собирает результаты на ранке 0. Исходный текст файла представлен в Приложение З;
- `src/utils.cpp`, `src/utils.hpp` - вспомогательный модуль: чтение переменных окружения, вычисление диапазонов байт для параллельного чтения файла, удаление граничных периодов, получение размера файла. Исходный текст файла представлен в Приложение И;

- `src/period_stats.hpp` - заголовочный файл с определением структуры `PeriodStats`, хранящей агрегированные статистики за один временной период (среднее значение, минимумы и максимумы Open/Close). Исходный текст файла представлен в Приложение К;
- `src/record.hpp` - заголовочный файл с определением структуры `Record` для хранения одной записи из CSV-файла Bitcoin (timestamp, open, high, low, close, volume). Исходный текст файла представлен в Приложение Л;
- `data/data_10s.csv` - текстовый файл с входными данными о стоимости Bitcoin по каждым 10 секундам в формате CSV;
- `result.csv` - выходной файл с найденными интервалами изменения цены не менее чем на 10%.

Также в рамках проекта используется система автоматизированной сборки. Для сборки и запуска проекта необходимо выполнить следующие команды:

```
make
make run
```

ЗАКЛЮЧЕНИЕ

В ходе выполнения лабораторной работы были решены задачи, направленные на освоение параллельных вычислений с использованием разнородных типов вычислительных ресурсов. В результате работы удалось:

- Создать виртуальные машины, обеспечивающие выполнение задач как на GPU-узлах, так и на CPU-узлах;
- Настроить сеть для обеспечения стабильной связи между хост-системой и виртуальными узлами;
- Реализовать параллельное приложение с использованием механизма OpenMPI, задействующее ресурсы разнородных узлов;
- Изучить технологии CUDA, OpenMPI, Slurm.

Были изучены технологии OpenMPI, CUDA Toolkit и библиотека CUB. В рамках работы было разработано параллельное приложение на языке C++, использующее разнородный вид вычислительных ресурсов для анализа временных рядов исторических данных о стоимости Bitcoin.

Разработанная программа выполняет параллельную агрегацию временных рядов и поиск интервалов значительного изменения цены. Ключевые особенности реализации:

- Параллельное чтение CSV-файла размером 2.3 ГБ с использованием техники перекрытия диапазонов для корректной обработки границ строк.
- Гибридная агрегация данных, поддерживающая вычисления как на CPU (последовательная обработка), так и на GPU (параллельная обработка с использованием CUDA).
- Два варианта CUDA-ядер: блочное ядро для больших интервалов агрегации (дни, часы) и простое ядро для малых интервалов (минуты, секунды).
- Динамическая загрузка GPU-плагина через dlopen, позволяющая запускать приложение на узлах без GPU без перекомпиляции.
- Параллельный поиск интервалов изменения цены с передачей незавершённых интервалов между ранками через MPI.
- Гибкая конфигурация через переменные окружения, позволяющая настраивать параметры работы без изменения исходного кода.

Для наглядной демонстрации эффективности параллельных вычислений был разработан скрипт линейной интерполяции данных, увеличивший объём исходного набора данных с 360 МБ до 2.3 ГБ. Это позволило достоверно оценить преимущества параллельной обработки и GPU-ускорения.

В рамках работы получены практические знания о гетерогенных вычислительных системах и реализовано полнофункциональное параллельное приложение, эффективно использующее ресурсы разнородных вычислителей для обработки больших объёмов временных рядов.

Список литературы

[1] Zielak - Bitcoin Historical Data // kaggle URL: https://www.kaggle.com/datasets/mczielinski/bitcoin-historical-data (дата обращения: 10.01.2026).

[2] Hyper-V Documentation // Microsoft URL: https://learn.microsoft.com/en-us/virtualization/hyper-v-on-windows/ (дата обращения: 10.01.2026).

[3] Ubuntu OS Docs // Ubuntu URL: https://ubuntu.com/server/docs (дата обращения: 10.01.2026).

[4] New-VMSwitch: Документация PowerShell // Microsoft 365 URL: https://learn.microsoft.com/en-us/powershell/module/hyper-v/new-vmswitch?view=windowsserver2025-ps (дата обращения: 10.01.2026).

[5] New-NetIPAddress: Документация PowerShell // Microsoft 365 URL: https://learn.microsoft.com/en-us/powershell/module/nettcpip/new-netipaddress?view=windowsserver2025-ps (дата обращения: 10.01.2026).

[6] New-NetNat: Документация PowerShell // Microsoft 365 URL: https://learn.microsoft.com/en-us/powershell/module/netnat/New-NetNat?view=windowsserver2016-ps (дата обращения: 10.01.2026).

[7] Add-NetNatStaticMapping: Документация PowerShell // Microsoft 365 URL: https://learn.microsoft.com/ru-ru/powershell/module/netnat/add-netnatstaticmapping?view=windowsserver2022-ps (дата обращения: 10.01.2026).

[8] Get-VMHostPartitionableGpu: Документация PowerShell // Microsoft 365 URL: https://learn.microsoft.com/en-us/powershell/module/hyper-v/get-vmhostpartitionablegpu?view=windowsserver2025-ps (дата обращения: 10.01.2026).

[9] Set-VM: Документация PowerShell // Microsoft 365 URL: https://learn.microsoft.com/en-us/powershell/module/hyper-v/set-vm?view=windowsserver2025-ps (дата обращения: 10.01.2026).

[10] RFC 1094: NFS: Network File System Protocol Specification // Sun Microsystems, Inc. URL: https://datatracker.ietf.org/doc/html/rfc1094 (дата обращения: 10.01.2026).

ПРИЛОЖЕНИЕ А

```
#!/bin/bash
#SBATCH --job-name=btc
#SBATCH --nodes=4
#SBATCH --ntasks=4
#SBATCH --cpus-per-task=2
#SBATCH --output=out.txt

# Путь к файлу данных (должен существовать на всех узлах)
export DATA_PATH="/mnt/shared/supercomputers/data/data_10s.csv"

# Доли данных для каждого ранка (сумма определяет пропорции)
export DATA_READ_SHARES="10,11,13,14"

# Размер перекрытия в байтах для обработки границ строк
export READ_OVERLAP_BYTES=131072

# Интервал агрегации в секундах (60 = минуты, 600 = 10 минут, 86400 = дни)
export AGGREGATION_INTERVAL=60

# Использовать ли CUDA для агрегации (0 = нет, 1 = да)
export USE_CUDA=1

# Использовать ли блочное ядро (быстрее для больших интервалов, 0 = нет, 1 = да)
export USE_BLOCK_KERNEL=0

cd /mnt/shared/supercomputers/build
mpirun -np $SLURM_NTASKS ./bitcoin_app
```

ПРИЛОЖЕНИЕ Б

```
CXX = mpic++
CXXFLAGS = -std=c++17 -O2 -Wall -Wextra -Wno-cast-function-type -fopenmp

NVCC = nvcc
NVCCFLAGS = -O3 -std=c++17 -arch=sm_86 -Xcompiler -fPIC

SRC_DIR = src
BUILD_DIR = build

SRCS = $(wildcard $(SRC_DIR)/*.cpp)
OBJS = $(patsubst $(SRC_DIR)/%.cpp,$(BUILD_DIR)/%.o,$(SRCS))

TARGET = $(BUILD_DIR)/bitcoin_app

PLUGIN_SRC = $(SRC_DIR)/gpu_plugin.cu
PLUGIN = $(BUILD_DIR)/libgpu_compute.so

all: $(PLUGIN) $(TARGET)

$(BUILD_DIR):
   mkdir -p $(BUILD_DIR)

$(BUILD_DIR)/%.o: $(SRC_DIR)/%.cpp | $(BUILD_DIR)
   $(CXX) $(CXXFLAGS) -c $< -o $@

$(TARGET): $(OBJS)
   $(CXX) $(CXXFLAGS) $^ -o $@ -ldl

$(PLUGIN): $(PLUGIN_SRC) | $(BUILD_DIR)
   $(NVCC) $(NVCCFLAGS) -shared $< -o $@

clean:
   rm -rf $(BUILD_DIR)

run: all
   sbatch run.slurm

.PHONY: all clean run
```

ПРИЛОЖЕНИЕ В

```cpp
#include <mpi.h>
#include <iostream>
#include <vector>
#include <iomanip>

#include "csv_loader.hpp"
#include "record.hpp"
#include "period_stats.hpp"
#include "aggregation.hpp"
#include "intervals.hpp"
#include "utils.hpp"
#include "gpu_loader.hpp"

int main(int argc, char** argv) {
    MPI_Init(&argc, &argv);
    double total_start = MPI_Wtime();

    int rank, size;
    MPI_Comm_rank(MPI_COMM_WORLD, &rank);
    MPI_Comm_size(MPI_COMM_WORLD, &size);

    // Проверяем доступность GPU
    bool use_cuda = get_use_cuda();
    bool have_gpu = gpu_is_available();
    bool use_gpu = use_cuda && have_gpu;

    std::cout << "Rank " << rank
              << ": USE_CUDA=" << use_cuda
              << ", GPU available=" << have_gpu
              << ", using " << (use_gpu ? "GPU" : "CPU")
              << std::endl;

    // Параллельное чтение данных
    double read_start = MPI_Wtime();
    std::vector<Record> records = load_csv_parallel(rank, size);
    double read_time = MPI_Wtime() - read_start;

    std::cout << "Rank " << rank
              << ": read " << records.size() << " records"
              << " in " << std::fixed << std::setprecision(3) << read_time << " sec"
              << std::endl;

    // Агрегация по периодам
    double agg_start = MPI_Wtime();
    std::vector<PeriodStats> periods;

    if (use_gpu) {
        int64_t interval = get_aggregation_interval();
        if (!aggregate_periods_gpu(records, interval, periods)) {
            std::cerr << "Rank " << rank << ": GPU aggregation failed, falling back to CPU" << std::endl;
            periods = aggregate_periods(records);
        }
    } else {
        periods = aggregate_periods(records);
    }

    double agg_time = MPI_Wtime() - agg_start;

    std::cout << "Rank " << rank
              << ": aggregated " << periods.size() << " periods"
              << " [" << (periods.empty() ? 0 : periods.front().period)
```

```
              << ".." << (periods.empty() ? 0 : periods.back().period) << "]"
              << " in " << std::fixed << std::setprecision(3) << agg_time << " sec"
              << std::endl;

    // Удаляем крайние периоды (могут быть неполными из-за параллельного чтения)
    trim_edge_periods(periods, rank, size);

    std::cout << "Rank " << rank
              << ": after trim " << periods.size() << " periods"
              << " [" << (periods.empty() ? 0 : periods.front().period)
              << ".." << (periods.empty() ? 0 : periods.back().period) << "]"
              << std::endl;

    // Параллельное построение интервалов
    IntervalResult iv_result = find_intervals_parallel(periods, rank, size);

    std::cout << "Rank " << rank
              << ": found " << iv_result.intervals.size() << " intervals"
              << ", compute " << std::fixed << std::setprecision(6) << iv_result.compute_time << " sec"
              << ", wait " << iv_result.wait_time << " sec"
              << std::endl;

    // Сбор интервалов на ранке 0
    double collect_wait = collect_intervals(iv_result.intervals, rank, size);

    if (rank == 0) {
        std::cout << "Rank 0: collected " << iv_result.intervals.size() << " total intervals"
                  << ", wait " << std::fixed << std::setprecision(3) << collect_wait << " sec"
                  << std::endl;
    }

    // Запись результатов в файл (только ранк 0)
    if (rank == 0) {
        double write_start = MPI_Wtime();
        write_intervals("result.csv", iv_result.intervals);
        double write_time = MPI_Wtime() - write_start;

        std::cout << "Rank 0: wrote result.csv"
                  << " in " << std::fixed << std::setprecision(3) << write_time << " sec"
                  << std::endl;
    }

    // Вывод общего времени выполнения
    MPI_Barrier(MPI_COMM_WORLD);
    double total_time = MPI_Wtime() - total_start;
    if (rank == 0) {
        std::cout << "Total execution time: "
                  << std::fixed << std::setprecision(3)
                  << total_time << " sec" << std::endl;
    }

    MPI_Finalize();
    return 0;
}
```

ПРИЛОЖЕНИЕ Г

```
#include "csv_loader.hpp"
#include <fstream>
#include <sstream>
#include <iostream>
#include <stdexcept>

bool parse_csv_line(const std::string& line, Record& record) {
    if (line.empty()) {
        return false;
    }

    std::stringstream ss(line);
    std::string item;

    try {
        // timestamp
        if (!std::getline(ss, item, ',') || item.empty()) return false;
        record.timestamp = std::stod(item);

        // open
        if (!std::getline(ss, item, ',') || item.empty()) return false;
        record.open = std::stod(item);

        // high
        if (!std::getline(ss, item, ',') || item.empty()) return false;
        record.high = std::stod(item);

        // low
        if (!std::getline(ss, item, ',') || item.empty()) return false;
        record.low = std::stod(item);

        // close
        if (!std::getline(ss, item, ',') || item.empty()) return false;
        record.close = std::stod(item);

        // volume
        if (!std::getline(ss, item, ',')) return false;
        // Volume может быть пустым или содержать данные
        if (item.empty()) {
            record.volume = 0.0;
        } else {
            record.volume = std::stod(item);
        }

        return true;
    } catch (const std::exception&) {
        return false;
    }
}

std::vector<Record> load_csv_parallel(int rank, int size) {
    std::vector<Record> data;

    // Читаем настройки из переменных окружения
    std::string data_path = get_data_path();
    std::vector<int> shares = get_data_read_shares();
    int64_t overlap_bytes = get_read_overlap_bytes();

    // Получаем размер файла
    int64_t file_size = get_file_size(data_path);

```

```
    // Вычисляем диапазон байт для этого ранка
    ByteRange range = calculate_byte_range(rank, size, file_size, shares, overlap_bytes);

    // Открываем файл и читаем нужный диапазон
    std::ifstream file(data_path, std::ios::binary);
    if (!file.is_open()) {
        throw std::runtime_error("Cannot open file: " + data_path);
    }

    // Переходим к началу диапазона
    file.seekg(range.start);

    // Читаем данные в буфер
    int64_t bytes_to_read = range.end - range.start;
    std::vector<char> buffer(bytes_to_read);
    file.read(buffer.data(), bytes_to_read);
    int64_t bytes_read = file.gcount();

    file.close();

    // Преобразуем в строку для удобства парсинга
    std::string content(buffer.data(), bytes_read);

    // Находим позицию начала первой полной строки
    size_t parse_start = 0;
    if (rank == 0) {
        // Первый ранк: пропускаем заголовок (первую строку)
        size_t header_end = content.find('\n');
        if (header_end != std::string::npos) {
            parse_start = header_end + 1;
        }
    } else {
        // Остальные ранки: начинаем с первого \n (пропускаем неполную строку)
        size_t first_newline = content.find('\n');
        if (first_newline != std::string::npos) {
            parse_start = first_newline + 1;
        }
    }

    // Находим позицию конца последней полной строки
    size_t parse_end = content.size();
    if (rank != size - 1) {
        // Не последний ранк: ищем последний \n
        size_t last_newline = content.rfind('\n');
        if (last_newline != std::string::npos && last_newline > parse_start) {
            parse_end = last_newline;
        }
    }

    // Парсим строки
    size_t pos = parse_start;
    while (pos < parse_end) {
        size_t line_end = content.find('\n', pos);
        if (line_end == std::string::npos || line_end > parse_end) {
            line_end = parse_end;
        }

        std::string line = content.substr(pos, line_end - pos);

        // Убираем \r если есть (Windows line endings)
        if (!line.empty() && line.back() == '\r') {
            line.pop_back();
        }

```

```
        Record record;
        if (parse_csv_line(line, record)) {
            data.push_back(record);
        }

        pos = line_end + 1;
    }

    return data;
}
```

ПРИЛОЖЕНИЕ Д

```cpp
#include "aggregation.hpp"
#include "utils.hpp"

#include <algorithm>
#include <cstdint>
#include <limits>
#include <vector>

std::vector<PeriodStats> aggregate_periods(const std::vector<Record>& records) {
    const int64_t interval = get_aggregation_interval();

    std::vector<PeriodStats> result;
    if (records.empty()) return result;

    struct PeriodAccumulator {
        double avg_sum = 0.0;
        double open_min = std::numeric_limits<double>::max();
        double open_max = std::numeric_limits<double>::lowest();
        double close_min = std::numeric_limits<double>::max();
        double close_max = std::numeric_limits<double>::lowest();
        int64_t count = 0;

        void add(const Record& r) {
            const double avg = (r.low + r.high) / 2.0;
            avg_sum += avg;
            open_min = std::min(open_min, r.open);
            open_max = std::max(open_max, r.open);
            close_min = std::min(close_min, r.close);
            close_max = std::max(close_max, r.close);
            ++count;
        }
    };

    PeriodIndex current_period =
        static_cast<PeriodIndex>(records[0].timestamp) / interval;

    PeriodAccumulator acc;
    acc.add(records[0]);

    for (size_t i = 1; i < records.size(); ++i) {
        const Record& r = records[i];
        const PeriodIndex period =
            static_cast<PeriodIndex>(r.timestamp) / interval;

        if (period != current_period) {
            PeriodStats stats;
            stats.period = current_period;
            stats.avg = acc.avg_sum / static_cast<double>(acc.count);
            stats.open_min = acc.open_min;
            stats.open_max = acc.open_max;
            stats.close_min = acc.close_min;
            stats.close_max = acc.close_max;
            stats.count = acc.count;
            result.push_back(stats);

            current_period = period;
            acc = PeriodAccumulator{};
        }

        acc.add(r);
    }
```

```
62
63      // последний период
64      PeriodStats stats;
65      stats.period = current_period;
66      stats.avg = acc.avg_sum / static_cast<double>(acc.count);
67      stats.open_min = acc.open_min;
68      stats.open_max = acc.open_max;
69      stats.close_min = acc.close_min;
70      stats.close_max = acc.close_max;
71      stats.count = acc.count;
72      result.push_back(stats);
73
74      return result;
75  }
```

ПРИЛОЖЕНИЕ Е

```
#include "gpu_loader.hpp"
#include <dlfcn.h>
#include <iostream>
#include <cstdint>

// Структура результата GPU (должна совпадать с gpu_plugin.cu)
struct GpuPeriodStats {
    int64_t period;
    double avg;
    double open_min;
    double open_max;
    double close_min;
    double close_max;
    int64_t count;
};

// Типы функций из GPU плагина
using gpu_is_available_fn = int (*)();

using gpu_aggregate_periods_fn = int (*)(
    const double* h_timestamps,
    const double* h_open,
    const double* h_high,
    const double* h_low,
    const double* h_close,
    int num_ticks,
    int64_t interval,
    GpuPeriodStats** h_out_stats,
    int* out_num_periods
);

using gpu_free_results_fn = void (*)(GpuPeriodStats*);

static void* get_gpu_lib_handle() {
    static void* h = dlopen("./libgpu_compute.so", RTLD_NOW | RTLD_LOCAL);
    return h;
}

bool gpu_is_available() {
    void* h = get_gpu_lib_handle();
    if (!h) return false;

    auto fn = reinterpret_cast<gpu_is_available_fn>(dlsym(h, "gpu_is_available"));
    if (!fn) return false;

    return fn() != 0;
}

bool aggregate_periods_gpu(
    const std::vector<Record>& records,
    int64_t aggregation_interval,
    std::vector<PeriodStats>& out_stats)
{
    if (records.empty()) {
        out_stats.clear();
        return true;
    }

    void* h = get_gpu_lib_handle();
    if (!h) {
        std::cerr << "GPU: Failed to load libgpu_compute.so" << std::endl;
```

```
        return false;
    }

    auto aggregate_fn = reinterpret_cast<gpu_aggregate_periods_fn>(
        dlsym(h, "gpu_aggregate_periods"));
    auto free_fn = reinterpret_cast<gpu_free_results_fn>(
        dlsym(h, "gpu_free_results"));

    if (!aggregate_fn || !free_fn) {
        std::cerr << "GPU: Failed to load functions from plugin" << std::endl;
        return false;
    }

    int num_ticks = static_cast<int>(records.size());

    // Конвертируем AoS в SoA
    std::vector<double> timestamps(num_ticks);
    std::vector<double> open(num_ticks);
    std::vector<double> high(num_ticks);
    std::vector<double> low(num_ticks);
    std::vector<double> close(num_ticks);

    for (int i = 0; i < num_ticks; i++) {
        timestamps[i] = records[i].timestamp;
        open[i] = records[i].open;
        high[i] = records[i].high;
        low[i] = records[i].low;
        close[i] = records[i].close;
    }

    // Вызываем GPU функцию
    GpuPeriodStats* gpu_stats = nullptr;
    int num_periods = 0;

    int result = aggregate_fn(
        timestamps.data(),
        open.data(),
        high.data(),
        low.data(),
        close.data(),
        num_ticks,
        aggregation_interval,
        &gpu_stats,
        &num_periods
    );

    if (result != 0) {
        std::cerr << "GPU: Aggregation failed with code " << result << std::endl;
        return false;
    }

    // Конвертируем результат в PeriodStats
    out_stats.clear();
    out_stats.reserve(num_periods);

    for (int i = 0; i < num_periods; i++) {
        PeriodStats ps;
        ps.period = gpu_stats[i].period;
        ps.avg = gpu_stats[i].avg;
        ps.open_min = gpu_stats[i].open_min;
        ps.open_max = gpu_stats[i].open_max;
        ps.close_min = gpu_stats[i].close_min;
        ps.close_max = gpu_stats[i].close_max;
        ps.count = gpu_stats[i].count;
```

```
126             out_stats.push_back(ps);
127         }
128 
129         // Освобождаем память
130         free_fn(gpu_stats);
131 
132         return true;
133 }
```

ПРИЛОЖЕНИЕ Ж

```cpp
#include <cuda_runtime.h>
#include <cub/cub.cuh>
#include <cstdint>
#include <cfloat>
#include <cstdio>
#include <cstdlib>
#include <ctime>
#include <string>
#include <sstream>
#include <iomanip>

// ============================================================================
// Структуры данных
// ============================================================================

// Результат агрегации одного периода
struct GpuPeriodStats {
    int64_t period;
    double avg;
    double open_min;
    double open_max;
    double close_min;
    double close_max;
    int64_t count;
};

// ============================================================================
// Вспомогательные функции
// ============================================================================

static double get_time_ms() {
    struct timespec ts;
    clock_gettime(CLOCK_MONOTONIC, &ts);
    return ts.tv_sec * 1000.0 + ts.tv_nsec / 1000000.0;
}

#define CUDA_CHECK(call) do { \
    cudaError_t err = call; \
    if (err != cudaSuccess) { \
        printf("CUDA error at %s:%d: %s\n", __FILE__, __LINE__, cudaGetErrorString(err)); \
        return -1; \
    } \
} while(0)

// ============================================================================
// Kernel: вычисление period_id для каждого тика
// ============================================================================

__global__ void compute_period_ids_kernel(
    const double* __restrict__ timestamps,
    int64_t* __restrict__ period_ids,
    int n,
    int64_t interval)
{
    int idx = blockIdx.x * blockDim.x + threadIdx.x;
    if (idx < n) {
        period_ids[idx] = static_cast<int64_t>(timestamps[idx]) / interval;
    }
}

// ============================================================================
```

```
// Kernel: агрегация одного периода (один блок на период)
// ============================================================================

__global__ void aggregate_periods_kernel(
    const double* __restrict__ open,
    const double* __restrict__ high,
    const double* __restrict__ low,
    const double* __restrict__ close,
    const int64_t* __restrict__ unique_periods,
    const int* __restrict__ offsets,
    const int* __restrict__ counts,
    int num_periods,
    GpuPeriodStats* __restrict__ out_stats)
{
    int period_idx = blockIdx.x;
    if (period_idx >= num_periods) return;

    int offset = offsets[period_idx];
    int count = counts[period_idx];

    // Используем shared memory для редукции внутри блока
    __shared__ double s_avg_sum;
    __shared__ double s_open_min;
    __shared__ double s_open_max;
    __shared__ double s_close_min;
    __shared__ double s_close_max;

    // Инициализация shared memory первым потоком
    if (threadIdx.x == 0) {
        s_avg_sum = 0.0;
        s_open_min = DBL_MAX;
        s_open_max = -DBL_MAX;
        s_close_min = DBL_MAX;
        s_close_max = -DBL_MAX;
    }
    __syncthreads();

    // Локальные аккумуляторы для каждого потока
    double local_avg_sum = 0.0;
    double local_open_min = DBL_MAX;
    double local_open_max = -DBL_MAX;
    double local_close_min = DBL_MAX;
    double local_close_max = -DBL_MAX;

    // Каждый поток обрабатывает свою часть тиков
    for (int i = threadIdx.x; i < count; i += blockDim.x) {
        int tick_idx = offset + i;
        double avg = (low[tick_idx] + high[tick_idx]) / 2.0;
        local_avg_sum += avg;
        local_open_min = min(local_open_min, open[tick_idx]);
        local_open_max = max(local_open_max, open[tick_idx]);
        local_close_min = min(local_close_min, close[tick_idx]);
        local_close_max = max(local_close_max, close[tick_idx]);
    }

    // Редукция с использованием атомарных операций
    atomicAdd(&s_avg_sum, local_avg_sum);
    atomicMin(reinterpret_cast<unsigned long long*>(&s_open_min),
              __double_as_longlong(local_open_min));
    atomicMax(reinterpret_cast<unsigned long long*>(&s_open_max),
              __double_as_longlong(local_open_max));
    atomicMin(reinterpret_cast<unsigned long long*>(&s_close_min),
              __double_as_longlong(local_close_min));
    atomicMax(reinterpret_cast<unsigned long long*>(&s_close_max),
```

```
            __double_as_longlong(local_close_max));

    __syncthreads();

    // Первый поток записывает результат
    if (threadIdx.x == 0) {
        GpuPeriodStats stats;
        stats.period = unique_periods[period_idx];
        stats.avg = s_avg_sum / static_cast<double>(count);
        stats.open_min = s_open_min;
        stats.open_max = s_open_max;
        stats.close_min = s_close_min;
        stats.close_max = s_close_max;
        stats.count = count;
        out_stats[period_idx] = stats;
    }
}

// ============================================================================
// Простой kernel для агрегации (один поток на период)
// Используется когда периодов много и тиков в каждом мало
// ============================================================================

__global__ void aggregate_periods_simple_kernel(
    const double* __restrict__ open,
    const double* __restrict__ high,
    const double* __restrict__ low,
    const double* __restrict__ close,
    const int64_t* __restrict__ unique_periods,
    const int* __restrict__ offsets,
    const int* __restrict__ counts,
    int num_periods,
    GpuPeriodStats* __restrict__ out_stats)
{
    int period_idx = blockIdx.x * blockDim.x + threadIdx.x;
    if (period_idx >= num_periods) return;

    int offset = offsets[period_idx];
    int count = counts[period_idx];

    double avg_sum = 0.0;
    double open_min = DBL_MAX;
    double open_max = -DBL_MAX;
    double close_min = DBL_MAX;
    double close_max = -DBL_MAX;

    for (int i = 0; i < count; i++) {
        int tick_idx = offset + i;
        double avg = (low[tick_idx] + high[tick_idx]) / 2.0;
        avg_sum += avg;
        open_min = min(open_min, open[tick_idx]);
        open_max = max(open_max, open[tick_idx]);
        close_min = min(close_min, close[tick_idx]);
        close_max = max(close_max, close[tick_idx]);
    }

    GpuPeriodStats stats;
    stats.period = unique_periods[period_idx];
    stats.avg = avg_sum / static_cast<double>(count);
    stats.open_min = open_min;
    stats.open_max = open_max;
    stats.close_min = close_min;
    stats.close_max = close_max;
    stats.count = count;
```

```
    out_stats[period_idx] = stats;
}


// ============================================================================
// Проверка доступности GPU
// ============================================================================

extern "C" int gpu_is_available() {
    int n = 0;
    cudaError_t err = cudaGetDeviceCount(&n);
    if (err != cudaSuccess) return 0;
    return (n > 0) ? 1 : 0;
}

// ============================================================================
// Главная функция агрегации на GPU
// ============================================================================

extern "C" int gpu_aggregate_periods(
    const double* h_timestamps,
    const double* h_open,
    const double* h_high,
    const double* h_low,
    const double* h_close,
    int num_ticks,
    int64_t interval,
    GpuPeriodStats** h_out_stats,
    int* out_num_periods)
{
    if (num_ticks == 0) {
        *h_out_stats = nullptr;
        *out_num_periods = 0;
        return 0;
    }

    std::ostringstream output;
    double total_start = get_time_ms();

    // ========================================================================
    // Шаг 1: Выделение памяти и копирование данных на GPU
    // ========================================================================
    double step1_start = get_time_ms();

    double* d_timestamps = nullptr;
    double* d_open = nullptr;
    double* d_high = nullptr;
    double* d_low = nullptr;
    double* d_close = nullptr;
    int64_t* d_period_ids = nullptr;

    size_t ticks_bytes = num_ticks * sizeof(double);

    CUDA_CHECK(cudaMalloc(&d_timestamps, ticks_bytes));
    CUDA_CHECK(cudaMalloc(&d_open, ticks_bytes));
    CUDA_CHECK(cudaMalloc(&d_high, ticks_bytes));
    CUDA_CHECK(cudaMalloc(&d_low, ticks_bytes));
    CUDA_CHECK(cudaMalloc(&d_close, ticks_bytes));
    CUDA_CHECK(cudaMalloc(&d_period_ids, num_ticks * sizeof(int64_t)));

    CUDA_CHECK(cudaMemcpy(d_timestamps, h_timestamps, ticks_bytes, cudaMemcpyHostToDevice));
    CUDA_CHECK(cudaMemcpy(d_open, h_open, ticks_bytes, cudaMemcpyHostToDevice));
    CUDA_CHECK(cudaMemcpy(d_high, h_high, ticks_bytes, cudaMemcpyHostToDevice));
    CUDA_CHECK(cudaMemcpy(d_low, h_low, ticks_bytes, cudaMemcpyHostToDevice));
```

```
254     CUDA_CHECK(cudaMemcpy(d_close, h_close, ticks_bytes, cudaMemcpyHostToDevice));
255
256     double step1_ms = get_time_ms() - step1_start;
257
258     // ========================================================================
259     // Шаг 2: Вычисление period_id для каждого тика
260     // ========================================================================
261     double step2_start = get_time_ms();
262
263     const int BLOCK_SIZE = 256;
264     int num_blocks = (num_ticks + BLOCK_SIZE - 1) / BLOCK_SIZE;
265
266     compute_period_ids_kernel<<<num_blocks, BLOCK_SIZE>>>(
267         d_timestamps, d_period_ids, num_ticks, interval);
268     CUDA_CHECK(cudaGetLastError());
269     CUDA_CHECK(cudaDeviceSynchronize());
270
271     double step2_ms = get_time_ms() - step2_start;
272
273     // ========================================================================
274     // Шаг 3: RLE (Run-Length Encode) для нахождения уникальных периодов
275     // ========================================================================
276     double step3_start = get_time_ms();
277
278     int64_t* d_unique_periods = nullptr;
279     int* d_counts = nullptr;
280     int* d_num_runs = nullptr;
281
282     CUDA_CHECK(cudaMalloc(&d_unique_periods, num_ticks * sizeof(int64_t)));
283     CUDA_CHECK(cudaMalloc(&d_counts, num_ticks * sizeof(int)));
284     CUDA_CHECK(cudaMalloc(&d_num_runs, sizeof(int)));
285
286     // Определяем размер временного буфера для CUB
287     void* d_temp_storage = nullptr;
288     size_t temp_storage_bytes = 0;
289
290     cub::DeviceRunLengthEncode::Encode(
291         d_temp_storage, temp_storage_bytes,
292         d_period_ids, d_unique_periods, d_counts, d_num_runs,
293         num_ticks);
294
295     CUDA_CHECK(cudaMalloc(&d_temp_storage, temp_storage_bytes));
296
297     cub::DeviceRunLengthEncode::Encode(
298         d_temp_storage, temp_storage_bytes,
299         d_period_ids, d_unique_periods, d_counts, d_num_runs,
300         num_ticks);
301     CUDA_CHECK(cudaGetLastError());
302
303     // Копируем количество уникальных периодов
304     int num_periods = 0;
305     CUDA_CHECK(cudaMemcpy(&num_periods, d_num_runs, sizeof(int), cudaMemcpyDeviceToHost));
306
307     cudaFree(d_temp_storage);
308     d_temp_storage = nullptr;
309
310     double step3_ms = get_time_ms() - step3_start;
311
312     // ========================================================================
313     // Шаг 4: Exclusive Scan для вычисления offsets
314     // ========================================================================
315     double step4_start = get_time_ms();
316
317     int* d_offsets = nullptr;
```

```
    CUDA_CHECK(cudaMalloc(&d_offsets, num_periods * sizeof(int)));

    temp_storage_bytes = 0;
    cub::DeviceScan::ExclusiveSum(
        d_temp_storage, temp_storage_bytes,
        d_counts, d_offsets, num_periods);

    CUDA_CHECK(cudaMalloc(&d_temp_storage, temp_storage_bytes));

    cub::DeviceScan::ExclusiveSum(
        d_temp_storage, temp_storage_bytes,
        d_counts, d_offsets, num_periods);
    CUDA_CHECK(cudaGetLastError());

    cudaFree(d_temp_storage);

    double step4_ms = get_time_ms() - step4_start;

    // ========================================================================
    // Шаг 5: Агрегация периодов
    // ========================================================================
    double step5_start = get_time_ms();

    GpuPeriodStats* d_out_stats = nullptr;
    CUDA_CHECK(cudaMalloc(&d_out_stats, num_periods * sizeof(GpuPeriodStats)));

    // Выбор ядра через переменную окружения USE_BLOCK_KERNEL
    const char* env_block_kernel = std::getenv("USE_BLOCK_KERNEL");
    if (env_block_kernel == nullptr) {
        printf("Error: Environment variable USE_BLOCK_KERNEL is not set\n");
        return -1;
    }
    bool use_block_kernel = std::atoi(env_block_kernel) != 0;

    if (use_block_kernel) {
        // Блочное ядро: один блок на период, потоки параллельно обрабатывают тики
        // Лучше для больших интервалов с множеством тиков в каждом периоде
        aggregate_periods_kernel<<<num_periods, BLOCK_SIZE>>>(
            d_open, d_high, d_low, d_close,
            d_unique_periods, d_offsets, d_counts,
            num_periods, d_out_stats);
    } else {
        // Простое ядро: один поток на период
        // Лучше для множества периодов с малым количеством тиков в каждом
        int agg_blocks = (num_periods + BLOCK_SIZE - 1) / BLOCK_SIZE;
        aggregate_periods_simple_kernel<<<agg_blocks, BLOCK_SIZE>>>(
            d_open, d_high, d_low, d_close,
            d_unique_periods, d_offsets, d_counts,
            num_periods, d_out_stats);
    }


    CUDA_CHECK(cudaGetLastError());
    CUDA_CHECK(cudaDeviceSynchronize());

    double step5_ms = get_time_ms() - step5_start;

    // ========================================================================
    // Шаг 6: Копирование результатов на CPU
    // ========================================================================
    double step6_start = get_time_ms();

    GpuPeriodStats* h_stats = new GpuPeriodStats[num_periods];
    CUDA_CHECK(cudaMemcpy(h_stats, d_out_stats, num_periods * sizeof(GpuPeriodStats),
```

```
                          cudaMemcpyDeviceToHost));

    double step6_ms = get_time_ms() - step6_start;

    // ============================================================================
    // Шаг 7: Освобождение GPU памяти
    // ============================================================================
    double step7_start = get_time_ms();

    cudaFree(d_timestamps);
    cudaFree(d_open);
    cudaFree(d_high);
    cudaFree(d_low);
    cudaFree(d_close);
    cudaFree(d_period_ids);
    cudaFree(d_unique_periods);
    cudaFree(d_counts);
    cudaFree(d_offsets);
    cudaFree(d_num_runs);
    cudaFree(d_out_stats);

    double step7_ms = get_time_ms() - step7_start;

    // ============================================================================
    // Итого
    // ============================================================================
    double total_ms = get_time_ms() - total_start;

    // Формируем весь вывод одной строкой
    output << "  GPU aggregation (" << num_ticks << " ticks, interval=" << interval << " sec, kernel=" << (use_block_kernel ? "block" : "simple") << "):\n";
    output << "    1. Malloc + H->D copy:  " << std::fixed << std::setprecision(3) << std::setw(7) << step1_ms << " ms\n";
    output << "    2. Compute period_ids:  " << std::setw(7) << step2_ms << " ms\n";
    output << "    3. RLE (CUB):           " << std::setw(7) << step3_ms << " ms (" << num_periods << " periods)\n";
    output << "    4. Exclusive scan:      " << std::setw(7) << step4_ms << " ms\n";
    output << "    5. Aggregation kernel:  " << std::setw(7) << step5_ms << " ms (" << (use_block_kernel ? "block" : "simple") << ")\n";
    output << "    6. D->H copy:           " << std::setw(7) << step6_ms << " ms\n";
    output << "    7. Free GPU memory:     " << std::setw(7) << step7_ms << " ms\n";
    output << "    GPU TOTAL:              " << std::setw(7) << total_ms << " ms\n";

    // Выводим всё одним принтом
    printf("%s", output.str().c_str());
    fflush(stdout);

    *h_out_stats = h_stats;
    *out_num_periods = num_periods;

    return 0;
}

// ============================================================================
// Освобождение памяти результатов
// ============================================================================

extern "C" void gpu_free_results(GpuPeriodStats* stats) {
    delete[] stats;
}
```

ПРИЛОЖЕНИЕ З

```cpp
#include "intervals.hpp"
#include "utils.hpp"
#include <mpi.h>
#include <algorithm>
#include <cmath>
#include <fstream>
#include <iomanip>
#include <sstream>
#include <ctime>
#include <limits>

// Вспомогательная структура для накопления min/max в интервале
struct IntervalAccumulator {
    PeriodIndex start_period;
    double start_avg;
    double open_min;
    double open_max;
    double close_min;
    double close_max;

    void init(const PeriodStats& p) {
        start_period = p.period;
        start_avg = p.avg;
        open_min = p.open_min;
        open_max = p.open_max;
        close_min = p.close_min;
        close_max = p.close_max;
    }

    void update(const PeriodStats& p) {
        open_min = std::min(open_min, p.open_min);
        open_max = std::max(open_max, p.open_max);
        close_min = std::min(close_min, p.close_min);
        close_max = std::max(close_max, p.close_max);
    }

    Interval finalize(const PeriodStats& end_period, double change) const {
        Interval iv;
        iv.start_period = start_period;
        iv.end_period = end_period.period;
        iv.start_avg = start_avg;
        iv.end_avg = end_period.avg;
        iv.change = change;
        iv.open_min = std::min(open_min, end_period.open_min);
        iv.open_max = std::max(open_max, end_period.open_max);
        iv.close_min = std::min(close_min, end_period.close_min);
        iv.close_max = std::max(close_max, end_period.close_max);
        return iv;
    }
};

// Упакованная структура PeriodStats для MPI передачи (8 doubles)
struct PackedPeriodStats {
    double period;   // PeriodIndex as double
    double avg;
    double open_min;
    double open_max;
    double close_min;
    double close_max;
    double count;    // int64_t as double
    double valid;    // флаг валидности (1.0 = valid, 0.0 = invalid)
```

```
62
63      void pack(const PeriodStats& ps) {
64          period = static_cast<double>(ps.period);
65          avg = ps.avg;
66          open_min = ps.open_min;
67          open_max = ps.open_max;
68          close_min = ps.close_min;
69          close_max = ps.close_max;
70          count = static_cast<double>(ps.count);
71          valid = 1.0;
72      }
73
74      PeriodStats unpack() const {
75          PeriodStats ps;
76          ps.period = static_cast<PeriodIndex>(period);
77          ps.avg = avg;
78          ps.open_min = open_min;
79          ps.open_max = open_max;
80          ps.close_min = close_min;
81          ps.close_max = close_max;
82          ps.count = static_cast<int64_t>(count);
83          return ps;
84      }
85
86      bool is_valid() const { return valid > 0.5; }
87      void set_invalid() { valid = 0.0; }
88  };
89
90  IntervalResult find_intervals_parallel(
91      const std::vector<PeriodStats>& periods,
92      int rank, int size,
93      double threshold)
94  {
95      IntervalResult result;
96      result.compute_time = 0.0;
97      result.wait_time = 0.0;
98
99      if (periods.empty()) {
100         if (rank < size - 1) {
101             PackedPeriodStats invalid;
102             invalid.set_invalid();
103             MPI_Send(&invalid, 8, MPI_DOUBLE, rank + 1, 0, MPI_COMM_WORLD);
104         }
105         return result;
106     }
107
108     double compute_start = MPI_Wtime();
109
110     size_t process_until = (rank == size - 1) ? periods.size() : periods.size() - 1;
111
112     IntervalAccumulator acc;
113     size_t start_idx = 0;
114     bool have_pending_interval = false;
115
116     if (rank > 0) {
117         double wait_start = MPI_Wtime();
118
119         PackedPeriodStats received;
120         MPI_Recv(&received, 8, MPI_DOUBLE, rank - 1, 0, MPI_COMM_WORLD, MPI_STATUS_IGNORE);
121
122         result.wait_time = MPI_Wtime() - wait_start;
123         compute_start = MPI_Wtime();
124
125         if (received.is_valid()) {
```

```
            PeriodStats prev_period = received.unpack();

            for (start_idx = 0; start_idx < periods.size(); start_idx++) {
                if (periods[start_idx].period > prev_period.period) {
                    break;
                }
            }

            if (start_idx < process_until) {
                acc.init(prev_period);
                have_pending_interval = true;

                for (size_t i = start_idx; i < process_until; i++) {
                    acc.update(periods[i]);

                    double change = std::abs(periods[i].avg - acc.start_avg) / acc.start_avg;

                    if (change >= threshold) {
                        result.intervals.push_back(acc.finalize(periods[i], change));
                        have_pending_interval = false;

                        start_idx = i + 1;
                        if (start_idx < process_until) {
                            acc.init(periods[start_idx]);
                            have_pending_interval = true;
                        }
                    }
                }
            }
        } else {
            if (process_until > 0) {
                acc.init(periods[0]);
                have_pending_interval = true;
                start_idx = 0;
            }
        }
    } else {
        if (process_until > 0) {
            acc.init(periods[0]);
            have_pending_interval = true;
            start_idx = 0;
        }
    }

    if (rank == 0 && have_pending_interval) {
        for (size_t i = 1; i < process_until; i++) {
            acc.update(periods[i]);

            double change = std::abs(periods[i].avg - acc.start_avg) / acc.start_avg;

            if (change >= threshold) {
                result.intervals.push_back(acc.finalize(periods[i], change));
                have_pending_interval = false;

                start_idx = i + 1;
                if (start_idx < process_until) {
                    acc.init(periods[start_idx]);
                    have_pending_interval = true;
                }
            }
        }
    }

    if (rank == size - 1 && have_pending_interval && !periods.empty()) {
```

```
        const auto& last_period = periods.back();
        double change = std::abs(last_period.avg - acc.start_avg) / acc.start_avg;
        result.intervals.push_back(acc.finalize(last_period, change));
    }

    result.compute_time = MPI_Wtime() - compute_start;

    if (rank < size - 1) {
        PackedPeriodStats to_send;

        if (have_pending_interval) {
            PeriodStats start_period;
            start_period.period = acc.start_period;
            start_period.avg = acc.start_avg;
            start_period.open_min = acc.open_min;
            start_period.open_max = acc.open_max;
            start_period.close_min = acc.close_min;
            start_period.close_max = acc.close_max;
            start_period.count = 0;
            to_send.pack(start_period);
        } else if (periods.size() >= 2) {
            to_send.pack(periods[periods.size() - 2]);
        } else {
            to_send.set_invalid();
        }

        MPI_Send(&to_send, 8, MPI_DOUBLE, rank + 1, 0, MPI_COMM_WORLD);
    }

    return result;
}

double collect_intervals(
    std::vector<Interval>& local_intervals,
    int rank, int size)
{
    double wait_time = 0.0;

    if (rank == 0) {
        for (int r = 1; r < size; r++) {
            double wait_start = MPI_Wtime();

            int count;
            MPI_Recv(&count, 1, MPI_INT, r, 1, MPI_COMM_WORLD, MPI_STATUS_IGNORE);

            if (count > 0) {
                std::vector<double> buffer(count * 9);
                MPI_Recv(buffer.data(), count * 9, MPI_DOUBLE, r, 2, MPI_COMM_WORLD, MPI_STATUS_IGNORE);

                for (int i = 0; i < count; i++) {
                    Interval iv;
                    iv.start_period = static_cast<PeriodIndex>(buffer[i * 9 + 0]);
                    iv.end_period = static_cast<PeriodIndex>(buffer[i * 9 + 1]);
                    iv.open_min = buffer[i * 9 + 2];
                    iv.open_max = buffer[i * 9 + 3];
                    iv.close_min = buffer[i * 9 + 4];
                    iv.close_max = buffer[i * 9 + 5];
                    iv.start_avg = buffer[i * 9 + 6];
                    iv.end_avg = buffer[i * 9 + 7];
                    iv.change = buffer[i * 9 + 8];
                    local_intervals.push_back(iv);
                }
            }

```

```
            wait_time += MPI_Wtime() - wait_start;
        }

        std::sort(local_intervals.begin(), local_intervals.end(),
            [](const Interval& a, const Interval& b) {
                return a.start_period < b.start_period;
            });
    } else {
        int count = static_cast<int>(local_intervals.size());
        MPI_Send(&count, 1, MPI_INT, 0, 1, MPI_COMM_WORLD);

        if (count > 0) {
            std::vector<double> buffer(count * 9);
            for (int i = 0; i < count; i++) {
                const auto& iv = local_intervals[i];
                buffer[i * 9 + 0] = static_cast<double>(iv.start_period);
                buffer[i * 9 + 1] = static_cast<double>(iv.end_period);
                buffer[i * 9 + 2] = iv.open_min;
                buffer[i * 9 + 3] = iv.open_max;
                buffer[i * 9 + 4] = iv.close_min;
                buffer[i * 9 + 5] = iv.close_max;
                buffer[i * 9 + 6] = iv.start_avg;
                buffer[i * 9 + 7] = iv.end_avg;
                buffer[i * 9 + 8] = iv.change;
            }
            MPI_Send(buffer.data(), count * 9, MPI_DOUBLE, 0, 2, MPI_COMM_WORLD);
        }
    }

    return wait_time;
}

std::string period_index_to_datetime(PeriodIndex period) {
    int64_t interval = get_aggregation_interval();
    time_t ts = static_cast<time_t>(period) * interval;
    struct tm* tm_info = gmtime(&ts);

    std::ostringstream oss;
    oss << std::setfill('0')
        << (tm_info->tm_year + 1900) << "-"
        << std::setw(2) << (tm_info->tm_mon + 1) << "-"
        << std::setw(2) << tm_info->tm_mday << " "
        << std::setw(2) << tm_info->tm_hour << ":"
        << std::setw(2) << tm_info->tm_min << ":"
        << std::setw(2) << tm_info->tm_sec;

    return oss.str();
}

void write_intervals(const std::string& filename, const std::vector<Interval>& intervals) {
    std::ofstream out(filename);

    out << std::fixed << std::setprecision(2);
    out << "start_datetime,end_datetime,open_min,open_max,close_min,close_max,start_avg,end_avg,change\n";

    for (const auto& iv : intervals) {
        out << period_index_to_datetime(iv.start_period) << ","
            << period_index_to_datetime(iv.end_period) << ","
            << iv.open_min << ","
            << iv.open_max << ","
            << iv.close_min << ","
            << iv.close_max << ","
            << iv.start_avg << ","
            << iv.end_avg << ","
```

```
318             << std::setprecision(6) << iv.change << "\n";
319     }
320 }
```

ПРИЛОЖЕНИЕ И

```cpp
#include "utils.hpp"
#include <fstream>
#include <sstream>
#include <stdexcept>
#include <numeric>

int get_num_cpu_threads() {
    const char* env_threads = std::getenv("NUM_CPU_THREADS");
    int num_cpu_threads = 1;
    if (env_threads) {
        num_cpu_threads = std::atoi(env_threads);
        if (num_cpu_threads < 1) num_cpu_threads = 1;
    }
    return num_cpu_threads;
}

std::string get_env(const char* name) {
    const char* env = std::getenv(name);
    if (!env) {
        throw std::runtime_error(std::string("Environment variable not set: ") + name);
    }
    return std::string(env);
}

std::string get_data_path() {
    return get_env("DATA_PATH");
}

std::vector<int> get_data_read_shares() {
    std::vector<int> shares;
    std::stringstream ss(get_env("DATA_READ_SHARES"));
    std::string item;
    while (std::getline(ss, item, ',')) {
        shares.push_back(std::stoi(item));
    }
    return shares;
}

int64_t get_read_overlap_bytes() {
    return std::stoll(get_env("READ_OVERLAP_BYTES"));
}

int64_t get_aggregation_interval() {
    return std::stoll(get_env("AGGREGATION_INTERVAL"));
}

bool get_use_cuda() {
    return std::stoi(get_env("USE_CUDA")) != 0;
}

int64_t get_file_size(const std::string& path) {
    std::ifstream file(path, std::ios::binary | std::ios::ate);
    if (!file.is_open()) {
        throw std::runtime_error("Cannot open file: " + path);
    }
    return static_cast<int64_t>(file.tellg());
}

ByteRange calculate_byte_range(int rank, int size, int64_t file_size,
                               const std::vector<int>& shares, int64_t overlap_bytes) {
    std::vector<int> effective_shares;
```

```
    if (shares.size() == static_cast<size_t>(size)) {
        effective_shares = shares;
    } else {
        effective_shares.assign(size, 1);
    }

    int total_shares = std::accumulate(effective_shares.begin(), effective_shares.end(), 0);
    int64_t bytes_per_share = file_size / total_shares;

    int64_t base_start = 0;
    for (int i = 0; i < rank; i++) {
        base_start += bytes_per_share * effective_shares[i];
    }

    int64_t base_end = base_start + bytes_per_share * effective_shares[rank];

    ByteRange range;

    if (rank == 0) {
        range.start = 0;
        range.end = std::min(base_end + overlap_bytes, file_size);
    } else if (rank == size - 1) {
        range.start = std::max(base_start - overlap_bytes, static_cast<int64_t>(0));
        range.end = file_size;
    } else {
        range.start = std::max(base_start - overlap_bytes, static_cast<int64_t>(0));
        range.end = std::min(base_end + overlap_bytes, file_size);
    }

    return range;
}

void trim_edge_periods(std::vector<PeriodStats>& periods, int rank, int size) {
    if (periods.empty()) return;

    if (rank == 0) {
        periods.pop_back();
    } else if (rank == size - 1) {
        periods.erase(periods.begin());
    } else {
        periods.pop_back();
        periods.erase(periods.begin());
    }
}
```

ПРИЛОЖЕНИЕ К

```
#pragma once
#include <cstdint>

using PeriodIndex = int64_t;

// Агрегированные данные за один период
struct PeriodStats {
    PeriodIndex period;   // индекс периода (timestamp / AGGREGATION_INTERVAL)
    double avg;           // среднее значение (Low + High) / 2 по всем записям
    double open_min;      // минимальный Open за период
    double open_max;      // максимальный Open за период
    double close_min;     // минимальный Close за период
    double close_max;     // максимальный Close за период
    int64_t count;        // количество записей, по которым агрегировали
};
```

ПРИЛОЖЕНИЕ Л

```
#pragma once
#include <cstdint>

struct Record {
    double timestamp;
    double open;
    double high;
    double low;
    double close;
    double volume;
};
```