Министерство образования Республики Беларусь
Учреждение образования
«Белорусский государственный университет
информатики и радиоэлектроники»

Факультет компьютерных систем и сетей

Кафедра электронных вычислительных машин

ИНТЕЛЛЕКТУАЛЬНАЯ ОБРАБОТКА БОЛЬШИХ ОБЪЕМОВ ДАННЫХ НА ОСНОВЕ ТЕХНОЛОГИЙ MPI И CUDA. ЛАБОРАТОРНЫЙ ПРАКТИКУМ

Рекомендовано УМО по образованию в области информатики и радиоэлектроники в качестве пособия для специальности 1-40 02 01 «Вычислительные машины, системы и сети»

Минск БГУИР 2017

УДК 004.032.2(076.5)
ББК 32.973.26-018.2я73
И73

А в т о р ы:

А. И. Демидчук, Д. Ю. Перцев, М. М. Татур, Д. В. Кришталь

Р е ц е н з е н т ы:

кафедра информационных систем и технологий
Белорусского национального технического университета
(протокол №3 от 28.11.2015);

заведующий лабораторией идентификации систем государственного научного учреждения «Объединенный институт проблем информатики Национальной академии наук Беларуси»,
доктор технических наук А. А. Дудкин

Интеллектуальная обработка больших объемов данных на основе технологий MPI и CUDA. Лабораторный практикум : пособие / А. И. Демидчук [и др.]. – Минск : БГУИР, 2017. – 60 с. : ил.
И73 ISBN 978-985-543-274-7.

Содержит минимальные необходимые теоретические сведения для выполнения лабораторных работ по дисциплине «Архитектура высокопроизводительных процессоров».

УДК 004.032.2(076.5)
ББК 32.973.26-018.2я73

ISBN 978-985-543-274-7

СОДЕРЖАНИЕ

ВВЕДЕНИЕ

В пособии студенту предлагается выполнить лабораторные работы, в рамках которых он познакомится с методологией разработки программ для сложных гетерогенных вычислительных систем. К каждой лабораторной работе приводятся краткие теоретические сведения, необходимые для выполнения задания.

Комплекс работ предполагает последовательное изучение технологии программирования параллельных архитектур и включает следующие этапы:

– разработка базовых версий для CPU и GPU на языке программирования C/C++ с применением технологии CUDA. При этом студент разрабатывает простейшие версии алгоритмов для двух вычислительных архитектур;

– выполнение оптимизации разработанных алгоритмов для работы на GPU, изучение основных методов оптимизации: механизма транзакций, дивергенции потоков и т. д.;

– знакомство с расширенными методами оптимизации: технология CUDA Stream. На данном этапе студент получает возможность запустить программу на кластере;

– внедрение технологии MPI, позволяющей распределить вычисления между всеми узлами кластера.

1. ОБЩИЕ ТЕОРЕТИЧЕСКИЕ СВЕДЕНИЯ

1.1. Формальные задачи интеллектуальной обработки данных

Ключевым понятием, с которым оперируют задачи интеллектуальной обработки данных, является **образ**. Образ, или объект, характеризуется упорядоченным набором параметров (вектором информативных признаков), над которыми осуществляется обработка. Число признаков (или размерность вектора) конечно и в общем случае равно *M*, тогда в геометрической интерпретации образ – это точка в *M*-мерном (числовом) пространстве признаков.

Примечание. Информативные признаки в общем случае могут быть не только числовыми, но и логическими (да, нет), качественными (серый, красный, зеленый, соленый, кислый, горький, веселый, грустный, задумчивый), порядковыми (первый, второй, десятый), структурными и т. п. Далее в пособии будем оперировать только с числовыми признаками. Для наглядности в учебном примере количество признаков ограничено двумя (рис. 1.1). Например, три образа x^1, x^2, x^3 представлены точками в двумерном пространстве признаков x_1 и x_2.

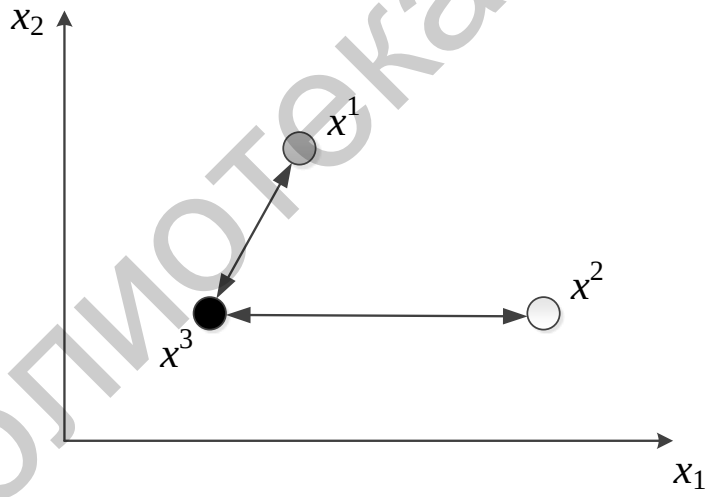

Рис. 1.1. Графическое представление трех образов в двумерном пространстве признаков

Образы могут быть сопоставлены или сравнены с использованием некоторой меры или метрики. Интуитивно понятной является метрика Евклидова расстояния (см. рис. 1.1), по которой образ x^3 более похож на образ x^1, т. к. расстояние между образами x^3 и x^1 меньше, чем расстояние между образами x^3 и x^2:

$$\rho_{\text{E}}(x^i, x^j) = \sqrt{\sum_{k=1}^{M} (x_k^i - x_k^j)^2}. \quad (1.1)$$

Наряду с евклидовым расстоянием существуют и другие меры сходства образов:

– манхэттенское расстояние:

$$\rho_{\text{M}}(x^i, x^j) = \sqrt{\sum_{k=1}^{M} |x_k^i - x_k^j|}; \quad (1.2)$$

– расстояние Чебышева:

$$\rho_{\text{Ч}}(x^i, x^j) = \max_k(|x_k^i - x_k^j|); \quad (1.3)$$

– расстояние Хэмминга:

$$\rho_{\text{X}}(x^i, x^j) = \sum_{k=1}^{M} \oplus(x_k^i, x_k^j), \quad (1.4)$$

где $(x_k^i, x_k^j) = \sum_{k=1}^{M} x_k^i \times x_k^j$,

M – число информативных признаков.

Выбор метрики сравнения образов осуществляют исходя из специфики решаемых задач. В данном пособии используется евклидова метрика, поскольку его основной целью является изучение методов и программно-аппаратных средств распараллеливания, а не тонкости интеллектуального анализа данных.

Различают следующие типовые задачи интеллектуальной обработки данных:

– *классификация* (дифференциация, разделение) – отнесение неизвестного образа к одному из предопределенных классов;

– *кластеризация* (группирование) – объединение образов в кластеры по заданному критерию;

– *ассоциативный поиск* – установление связей между образами;

– *регрессия* – выявление закономерности распределения группы образов.

Каждая из задач может быть решена различными алгоритмами, которые достаточно хорошо изучены и описаны в литературе. Большинство алгоритмов интеллектуального анализа данных связано с рекуррентными вычислениями расстояний (или весов) и перебором признаков, образов, классов и др., что и является предметом для распараллеливания.

В настоящем пособии мы остановимся на одном из известных и наиболее очевидных алгоритмов кластеризации – *k*-средних – и на его примере будем осваивать основы параллельного программирования с использованием технологии CUDA.

1.2. Алгоритм *k*-средних

Исходными данными для кластеризации алгоритмом *k*-средних является набор образов и число кластеров – *k*, в которые необходимо сгруппировать образы. Алгоритмы кластеризации (в том числе и алгоритм *k*-средних) стремятся сгруппировать образы так, чтобы расстояние между образами было минимальным, а расстояние между центрами кластеров – максимальным.

Проиллюстрируем работу алгоритма на простом графическом примере. Пусть задано множество из семи образов в двумерном пространстве признаков x_1 и x_2, которые необходимо сгруппировать в два кластера, т. е. $k = 2$ (рис. 1.2).

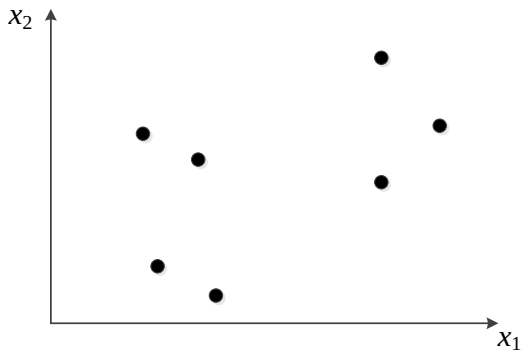

Рис. 1.2. Графическое представление семи образов в двумерном признаковом пространстве

Шаг 1. Для начала работы (инициализации алгоритма) необходимо задать центры кластеров.

Существуют различные способы инициализации, которые могут повлиять как на время работы алгоритма, так и на результат. Один из вариантов – выбрать случайные значения в пределах от минимального до максимальных чисел из исходных данных. Пусть в качестве центров будут точки, отмеченные квадратами, как показано на рис. 1.3. Другой способ – случайная выборка образов из анализируемого множества в качестве начальных центров.

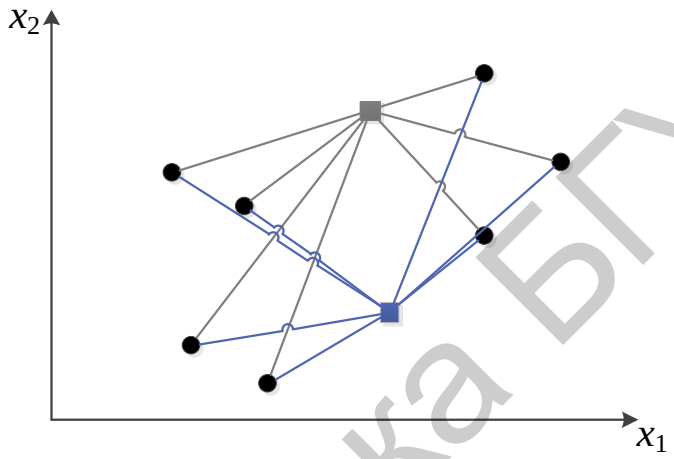

Рис. 1.3. Задание начальных центров кластеров и определение расстояний

Шаг 2. Для каждого образа x^j необходимо вычислить расстояние к каждому центру кластера d^j (в примере используется метрика евклидова расстояния):

$$d^j = \sqrt{\sum_{i=0}^{M} \left(x_i^j - k_i^l\right)^2}, \qquad (1.5)$$

где M – число признаков объекта, равное 2;

x_i^j – признак i исследуемого образа j;

k_i^l – признак i текущего центра кластера l.

Шаг 3. Из каждой пары полученных расстояний выбирается минимальное ($d_{\min}$) и к данному кластеру приписывается образ x^j. В результате первой итерации образы будут сгруппированы, как показано на рис. 1.4.

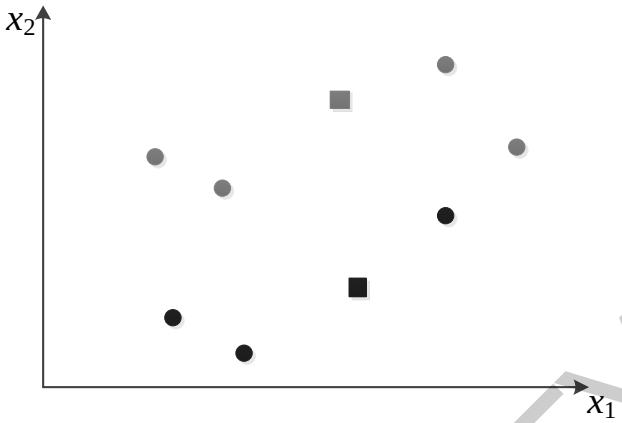

Рис. 1.4. Результат после первой итерации алгоритма *k*-средних

Шаг 4. Определяются новые центры кластеров как среднее арифметическое по каждому признаку объектов, отнесенных к одному и тому же кластеру k (рис. 1.5):

$$k_i^{new} = \frac{1}{N}\sum_{j=0}^{N} \mathrm{x}_i^j, \tag{1.6}$$

где N – число объектов, отнесенных к кластеру k.

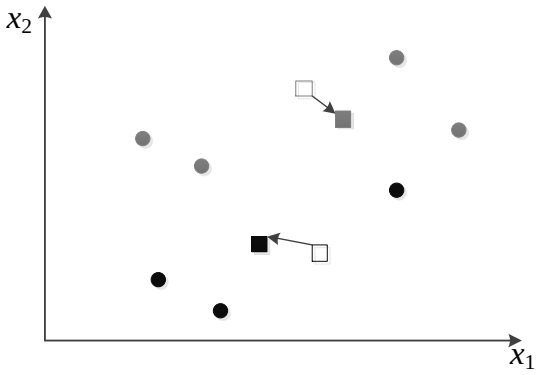

Рис. 1.5. Вычисление новых центров кластеров

Описанная последовательность действий (шаги 2, 3, 4) повторяется либо до полного совпадения значений центров кластеров, либо до достижения заданного порога разности центров в предыдущей и текущей итерациях алгоритма.

В результате работы нескольких итераций алгоритма кластеризации часть образов объединится в кластер «серых», другая часть – в кластер «черных», а центры кластеров пройдут путь от начальных значений к конечным, как показано на рис. 1.6.

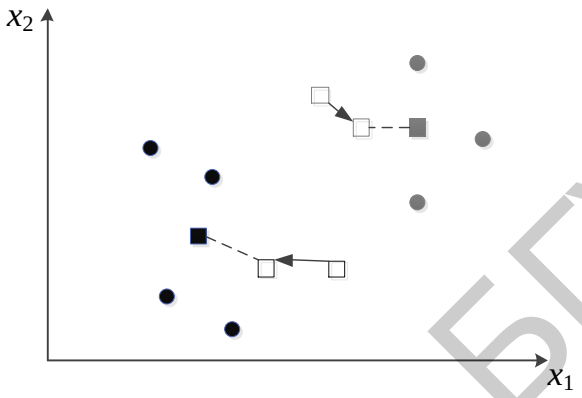

Рис. 1.6. Окончание группирования образов в два кластера

1.3. Вычислительный пример

Проиллюстрируем работу алгоритма на вычислительном примере.

Пусть задана совокупность из девяти образов, представленных векторами в пятимерном пространстве числовых признаков, которые необходимо сгруппировать в три кластера (рис. 1.7).

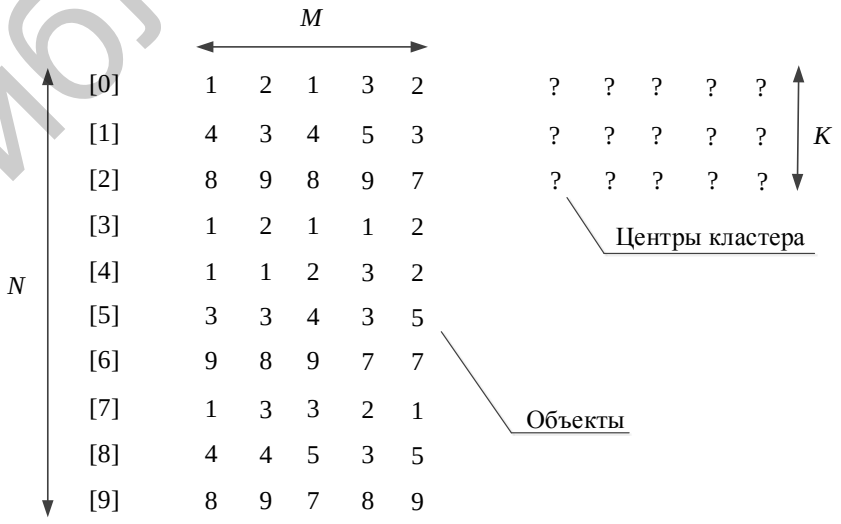

Рис. 1.7. Структура исходных данных

Шаг 1. Инициализируем алгоритм (выбираем центры начальных кластеров). Для этого заполняем случайными значениями элементы «?» в пределах от минимального до максимального числа из исходных данных (рис. 1.8).

Шаг 2. Для каждого образа x^j по формуле (1.5) вычисляется расстояние к каждому центру кластера.

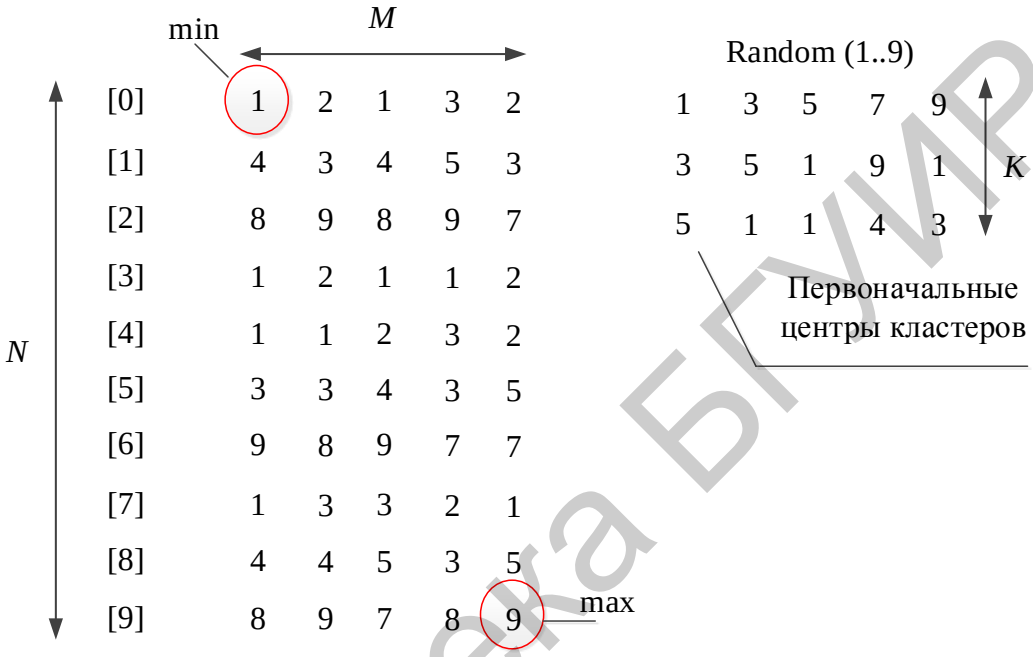

Рис. 1.8. Заполнение начальных центров

Шаг 3. Из полученных расстояний выбирается минимальное, и к данному кластеру относят образ. В примере на рис. 1.8 для рассматриваемого образа расстояния к центрам составляют:

$$D_1 = \sqrt{\left((1-1)+(2-3)+(1-5)+(3-7)+(2-9)\right)^2} = 16,$$

$$D_2 = \sqrt{\left((1-3)+(2-5)+(1-1)+(3-9)+(2-1)\right)^2} = 10,$$

$$D_3 = \sqrt{\left((1-5)+(2-1)+(1-1)+(3-4)+(2-3)\right)^2} = 5.$$

Минимальное расстояние к третьему кластеру $D_3 = 5$, следовательно, данный объект на текущей итерации относится к кластеру 3. Все остальные объекты вычисляются по аналогии.

Шаг 4. Затем значения центров кластера обновляются. Для этого вычисляется среднее арифметическое по каждому признаку объектов, отнесенных к одному и тому же кластеру. Объекты, которые принадлежат нулевому кластеру из рассматриваемого примера, – это объекты с индексами 2, 6 и 9 (рис. 1.9).

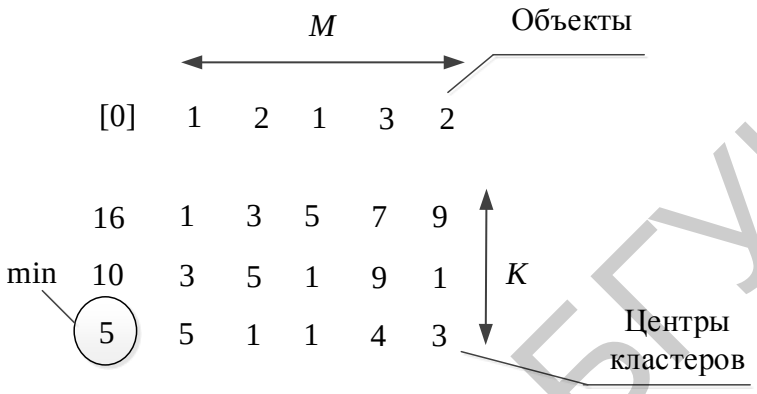

Рис. 1.9. Вычисление расстояний к центрам кластеров

Полученное множество значений средних сохраняется в качестве новых значений центра кластера (рис. 1.10).

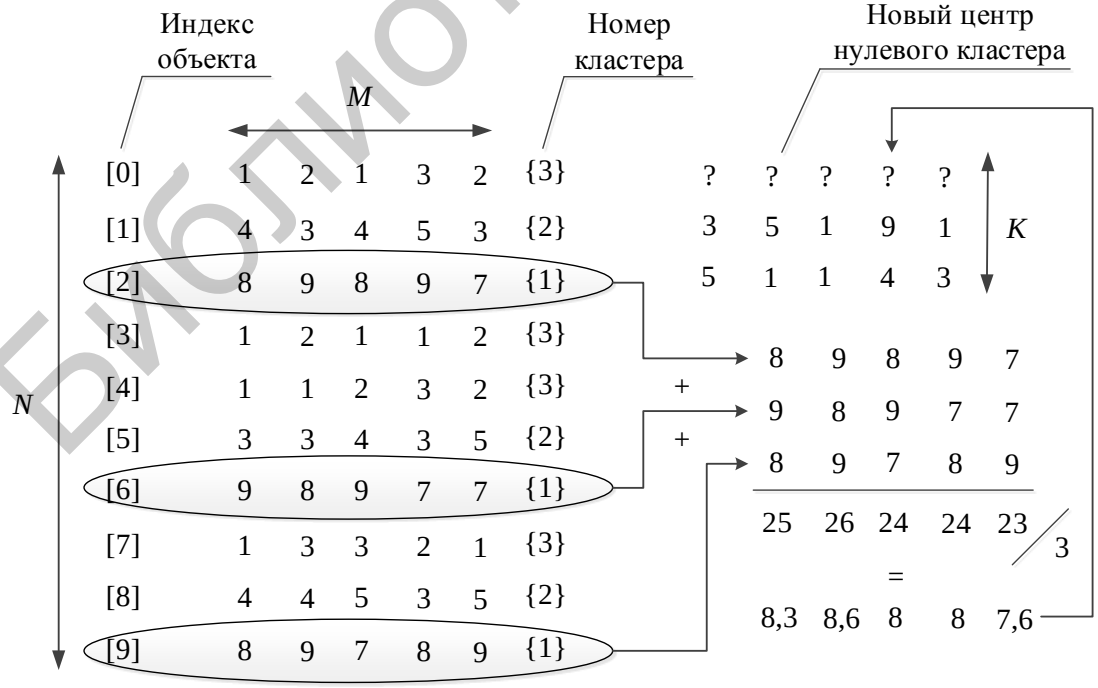

Рис. 1.10. Вычисление нового центра кластера

Описанная последовательность действий повторяется либо до полного совпадения начального и рассчитанного значений центров кластеров, либо до достижения заданного порога разности центров с предыдущей и текущей итерациями алгоритма.

2. ВВЕДЕНИЕ В ПРОГРАММИРОВАНИЕ GPU

2.1. Базовые понятия в CUDA

В упрощенном виде вычислительный блок графического процессора представляет собой совокупность потоковых мультипроцессоров (streaming multiprocessor, SMX), управляемых с помощью GigaThread Engine. Каждый SMX состоит из множества вычислительных CUDA-ядер (рис. 2.1).

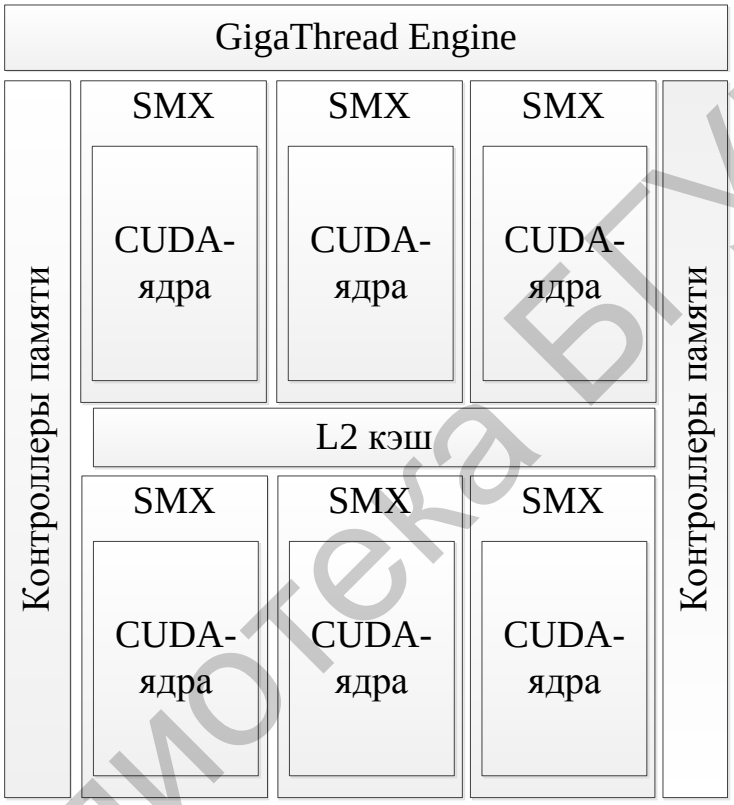

Рис. 2.1. Упрощенная структура графического процессора

Такая архитектура легла в основу следующих базовых понятий в CUDA: нить (thread), блок (block) и решетка блоков (grid).

Нить (thread) – базовая абстракция, представляющая собой легковесный поток, отвечающий за исполнение инструкций. Нить соответствует одному исполняющему ядру CUDA мультипроцессора. Максимальное число нитей, которое можно определить, – 2048 на один блок (для архитектуры NVIDIA Kepler). Легковесность нити заключается в том, что время, необходимое на его создание, разрушение и переключение между нитями, ничтожно мало. Для упрощения аппаратной схемы каждые 32 CUDA-ядра соединены с одним счетчиком

команд, поэтому данные блоки ядер всегда выполняют одну и ту же инструкцию. Данные 32 нити называются *варпом* (*warp*), который выполняется на потоковом мультипроцессоре (SMX). Учитывая, что в SMX включено более 32 нитей, это позволяет выполнять несколько варпов одновременно.

Множество нитей объединяется в **блок (block)**, который проецируется на мультипроцессор. Это означает, что все нити блока всегда будут выполняться на выделенном для него мультипроцессоре.

Решетка блоков (grid) представляет собой самый высокий уровень абстракции и, как правило, является параллельным вычислителем, который занимается решением поставленной задачи.

Каждая из абстракций является трехмерной. Для ее определения в NVIDIA CUDA используется специальная структура dim3. Инициализация значений структуры осуществляется следующим образом:

```
dim3 blocks (16, 16); // эквивалентно blocks(16, 16, 1)
dim3 grid (256); // эквивалентно grid(256, l, 1)
```

Для того чтобы определить текущие координаты, в решаемой задаче введены следующие константы:

```
// размерность решетки блоков при запуске ядра
dim3 gridDim;
// координаты текущего блока внутри решетки блоков
uint3 blockIdx;
// размерность блока при запуске ядра
dim3 blockDim;
// координаты текущей нити внутри блока
uint3 threadIdx;
```

Например, требуется сложить два вектора размерностью $\text{size} = N \cdot N$ элементов между собой:

```
void sum(float *A, float *B, float *C, int size) {
     for (int i = 0; i < size; i++) {
          C[i] = A[i] + B[i];
     }
}
```

При решении данной задачи на GPU можно распределить нагрузку следующим образом: решетка блоков – вся задача, число нитей – 1024, число блоков – size/<число нитей>. На GPU ядро (функция, описывающая последовательность операций, выполняемых каждой нитью параллельно), описывающее данное суммирование, выглядит следующим образом:

```
__global__ void MatAdd(
     float *A, float *B, float *C, int size
) {
     int idx = blockIdx.x * blockDim.x + threadIdx.x;
     if (idx < size) {
          C[idx] = A[idx] + B[idx];
     }
}
```

Для определения, на каком типе устройства может исполняться функция, введены следующие ключевые слова:

– *__host__* – функция может быть вызвана и выполнена только на стороне хоста (на центральном процессоре). Если не указывается спецификатор, функция считается объявленной как *__host__*;

– *__global__* – ядро предназначено для выполнения на устройстве (графическом процессоре) и может быть вызвано с хоста. На устройствах с Compute Capability 3.5 (способ описания версии архитектуры графического процессора и поддерживаемого CUDA API) или более новыми версиями допускается вызов с устройства. Ядра данного типа не возвращают никаких данных (тип void);

– *__device__* – ядро выполняется на устройстве, вызывается из кода, выполняемого на устройстве.

Запуск ядра выглядит следующим образом:

```
int main(){
     // ...
     const int N = 1024 * 1024;
     dim3 threadsPerBlock(1024);
     dim3 numBlocks((N + 1023) / threadsPerBlock.x);
     MatAdd<<<numBlocks, threadsPerBlock>>>(d_A, d_B, d_C, N);
     // ...
}
```

Специальное расширение языка C <<<...>>> определяет конфигурацию запуска нитей. Первый параметр задает количество блоков по каждому из трех измерений (тип dim3), а второй – количество нитей в блоке также по каждому измерению.

Все запуски CUDA-ядер являются асинхронными: CPU запрашивает разрешение на запуск ядра путем записи в специальный буфер команд без проверки, выполнен код или нет, после этого продолжает выполнять код в соответствии с алгоритмом для хоста.

2.2. Типы памяти GPU

Технология CUDA использует следующие типы памяти (рис. 2.2):

- регистры;
- локальная;
- глобальная;
- разделяемая;
- константная;
- текстурная.

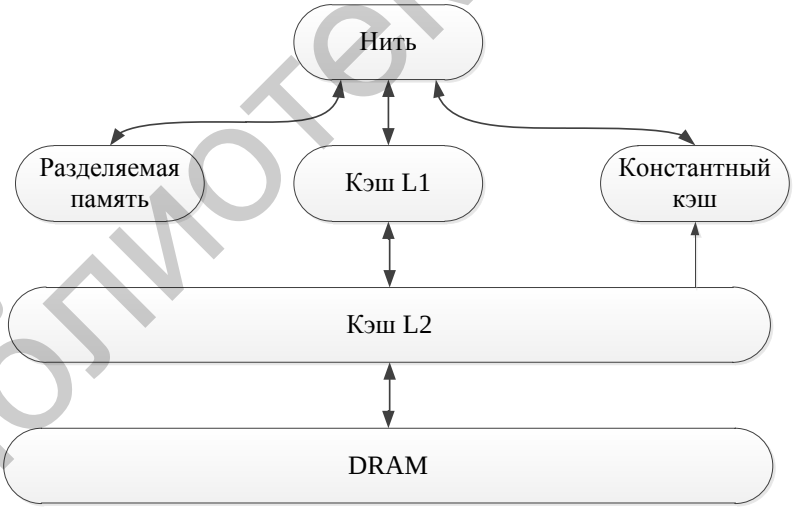

Рис. 2.2. Иерархия памяти

Однако разработчику недоступны к управлению регистры и локальная память. Все остальные типы он вправе использовать по своему усмотрению.

Характеристики каждого типа памяти представлены в табл. 2.1. Для локальной и глобальной памяти был добавлен кэш, начиная с Compute Capability 2.0.

Таблица 2.1

Типы памяти CUDA

Тип памяти	Расположение	Кэш	Доступ	Видимость	Время жизни
Регистры	Мультипроцессор	–	R/W	Нить	Нить
Локальная	DRAM GPU	Есть	R/W	Нить	Нить
Разделяемая	Мультипроцессор	–	R/W	Блок	Блок
Глобальная	DRAM GPU	Есть	R/W	Решетка блоков	–
Константная	DRAM GPU	Есть	R	Решетка блоков	–
Текстурная	DRAM GPU	Есть	R	Решетка блоков	–

Глобальная память (global memory) – тип памяти с самой высокой латентностью из доступных на GPU. Переменные в данном типе памяти можно выделить с помощью спецификатора *__global__*, а также динамически с помощью функций из семейства cudaMalloc():

```
cudaError_t cudaMalloc(
     void** devPtr,
     size_t size
);
```

где devPtr – выделенный указатель в глобальной памяти;

size – размер выделяемой памяти, байт.

Если память была успешно выделена, возвращаемое значение – cudaSuccess.

Глобальная память в основном служит для хранения больших объемов данных, над которыми осуществляется обработка. В нее загружаются данные из ОЗУ центрального процессора, результаты вычислений считываются обратно в оперативную память. Данные перемещения осуществляются с использованием функции cudaMemcpy():

```
cudaError_t cudaMemcpy (
     void* dst,
     const void* src,
     size_t count,
     cudaMemcpyKind kind
);
```

где dst – адрес, по которому данные будут скопированы;

src – адрес, начиная с которого данные будут скопированы;

count – размер копируемых данных, байт;

kind – направление копирования.

При этом в качестве направления копирования kind могут быть следующие значения:

- cudaMemcpyHostToHost – копирование с хоста на хост (аналог классической функции memcpy);
- cudaMemcpyHostToDevice – копирование с хоста в память GPU;
- cudaMemcpyDeviceToHost – копирование из памяти GPU в память хоста;
- cudaMemcpyDeviceToDevice – копирование в памяти GPU;
- cudaMemcpyDefault – копирование по умолчанию (работает только при использовании режима Unified Virtual Address и в данном пособии не рассматривается).

В алгоритмах, требующих высокой производительности, количество операций с глобальной памятью необходимо свести к минимуму, что связано с низкой пропускной способностью шины данных.

Разделяемая память (shared memory) относится к типу памяти с низкой латентностью. Физически она располагается на кристалле графического процессора и совмещена с кэшем L1, имеющим суммарный размер 64 Кб. При этом программист может выбрать одну из следующих конфигураций: 16 Кб для разделяемой памяти и 48 Кб для кэша либо 48 Кб разделяемой памяти и 16 Кб для кэша L1.

Данный тип памяти рекомендуется использовать для минимизации обращения к глобальной памяти. Адресация разделяемой памяти осуществляется между нитями одного блока, что может быть использовано для обмена данными между ними в пределах блока. Для размещения данных в разделяемой памяти используется спецификатор *__shared__*.

Константная память (constant memory) является одной из самых быстрых из доступных с хоста на GPU. Отличительной особенностью данного типа памяти является возможность записи данных с хоста, но при этом в пределах GPU возможно лишь чтение из этой памяти, что и обуславливает ее название. Константная память обладает собственным кэшем, что обеспечивало выигрыш на платформах с Compute Capability 1.x, однако на более новых архитектурах из-за ввода кэшей L1 и L2 константная память частично потеряла свою актуальность. Для размещения данных в константной памяти предусмотрен спецификатор *__constant__*.

Регистровая память (register) является самой быстрой из всех типов памяти. Определить общее количество регистров, доступных GPU, можно с помощью функции cudaGetDeviceProperties:

```
cudaError_t cudaGetDeviceProperties(
     cudaDeviceProp* prop,
     int device
);
```

где prop – указатель на структуру параметров;

device – индекс GPU, для которого требуется получить данные.

В структуре содержится два поля, позволяющих определить число доступных регистров:

– regsPerBlock – число 32-битных регистров, доступных в пределах блока;

– regsPerMultiprocessor – число 32-битных регистров, доступных на мультипроцессоре.

Количество регистров, доступных одной нити, определяется по формуле

$$RegsPerThread = \frac{MaxRegsCount}{ThreadsPerBlock \cdot BlocksPerGrid}, \quad (2.1)$$

где $MaxRegsCount$ – общее количество регистров, доступных GPU (определяется по спецификации для используемой архитектуры);

$ThreadsPerBlock$ – число нитей в блоке;

$BlocksPerGrid$ – число блоков в решетке блоков.

Все регистры GPU 32-разрядные. При этом в технологии CUDA нет явных способов использования регистровой памяти, всю работу по размещению данных в регистрах берет на себя компилятор.

Локальная память (local memory) может быть использована компилятором, если все локальные переменные не могут быть размещены в регистровой памяти. По скоростным характеристикам локальная память значительно медленнее, чем регистровая. Для проверки использования локальной памяти рекомендуется при компиляции включить опцию PTXAS Output в свойствах проекта в Microsoft Visual Studio (опция --ptxas-options=-v при компиляции через командную строку) и проанализировать вывод информации о сборке проекта.

Более подробная информация по особенностям работы с памятью на GPU представлена в официальной документации на сайте компании NVIDIA [1].

2.3. Задание

Разработать алгоритм кластеризации в соответствии с предложенной теорией. Разработка выполняется на языке программирования C/C++ для CPU и для GPU с размещением данных в глобальной памяти. Тестовый набор данных предоставляется преподавателем и соответствует формату .arff [2] (см. прил. 2). В качестве эталона при тестировании правильности работы алгоритмов рекомендуется использовать программу Weka 3 [3]. Для последующих лабораторных работ рекомендуется использовать реализацию на CPU в качестве эталонной, что позволит автоматизировать процесс тестирования.

В качестве отчетного материала предлагается использовать:

– описание CPU- и GPU-реализаций;

– описание типов переменных, размеры выделенной памяти с обоснованием, для чего используется каждый массив;

– сравнение времени работы CPU- и GPU-реализаций.

3. ТИПЫ ПАМЯТИ В CUDA. ОПТИМИЗАЦИЯ ДОСТУПА

3.1. Оптимизация работы с глобальной памятью

Как уже отмечалось, операции копирования между ОЗУ и памятью видеокарты обладают существенной латентностью.

Все современные операционные системы работают в режиме с виртуальной адресацией памяти. Для каждого запущенного процесса выделяется свое адресное пространство, в рамках которого процесс может осуществлять чтение и запись данных. Сторонний процесс не может получить доступ на чтение/запись в области памяти текущего процесса без специального разрешения со стороны ОС.

В большинстве операционных систем виртуальное адресное пространство делится на страницы по 4096 байт. Для доступа к физическому адресу данных в процессор встроена таблица PTE (Page Table Entry), осуществляющая трансляцию между виртуальным и физическим адресами. В случае если запрашиваемая страница не загружена, генерируется исключение, которое должна обработать ОС. Схема работы показана на рис. 3.1.

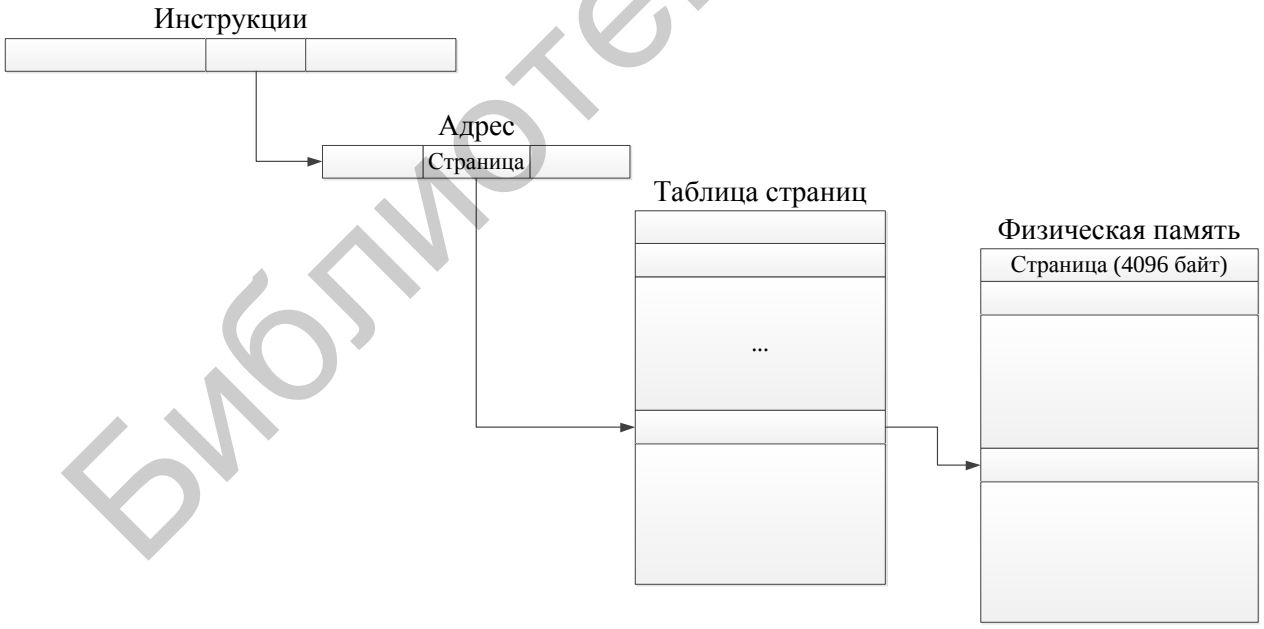

Рис. 3.1. Виртуальное адресное пространство

Для организации доступа к глобальной памяти на видеокарте используется упрощенный механизм виртуальной адресации – сохраняется безопасность и локальность в памяти для конкретной CUDA-программы, однако убрана страничная адресация. Различия в механизме адресации для CPU и GPU привели к тому, что доступ к ОЗУ центральным процессором является относительно сложной задачей:

1) определяется номер страницы, содержащей необходимые данные в ОЗУ, и ее наличие в памяти;

2) осуществляется копирование страницы в специальный pinned-буфер – область памяти, защищенная от сбрасывания на диск (swap);

3) выполняется синхронное копирование страницы в память GPU с использованием DMA;

4) освобождаются запрошенные ресурсы.

Данный алгоритм повторяется для каждой последующей страницы.

Очевидно, что из-за большого числа операций приблизиться к пиковой пропускной способности шины PCI Express при копировании данных между ОЗУ и памятью GPU не представляется возможным.

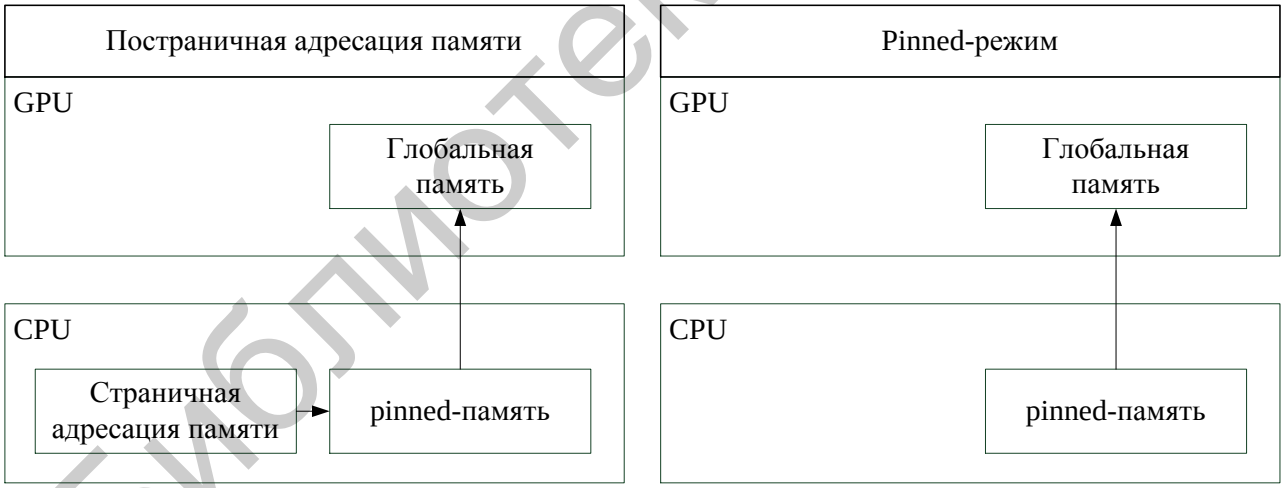

Рис. 3.2. Сравнение передачи данных в постраничном и pinned-режимах

Для решения данной проблемы в CUDA предложен способ выделения памяти в ОЗУ непосредственно в pinned-буфере в обход страничной адресации. С одной стороны, ОС теряет в объеме доступной динамической памяти ОЗУ и возможности кэшировать данные на диск, а с другой – существенно упрощается механизм копирования из ОЗУ в память GPU, что позволяет приблизиться к

пиковой пропускной способности PCI Express. Для выделения памяти в pinned-режиме достаточно выделить память в ОЗУ с применением следующей функции:

```
cudaError_t cudaMallocHost (
     void** ptr,     // указатель на выделяемую память в ОЗУ
     size_t size     // объем выделяемой памяти
);
```

Заранее выделенная системная память может быть также зарегистрирована в pinned-режиме с использованием CUDA API:

```
cudaError_t cudaHostRegister (
     void * ptr,     // указатель на память ОЗУ
     size_t size,    // размер регистрируемой памяти в байтах
     unsigned int flags
);
```

К параметрам управления (флагам) относятся следующие допустимые значения:

– cudaHostRegisterPortable – зарегистрированная память переводится в pinned-режим для всех контекстов, а не только для текущего;

– cudaHostRegisterMapped – проецирует выделенную память в адресное пространство CUDA. Указатель на память в GPU может быть получен с помощью cudaHostGetDevicePointer().

Одним из основных способов оптимизации доступа к глобальной памяти внутри ядра является механизм объединения запросов (coalescing) и формирования транзакций. Для формирования транзакций требуется, чтобы нити обращались к одному выравненному 128-байтному сегменту данных при считывании/записи 4-байтового слова (типы int, float) или 8-байтового (тип double) (рис. 3.3). Для данных, не кратных указанным размерам, рекомендуется вводить структуры с заполняющими полями либо использовать спецификатор выравнивания __align(16).

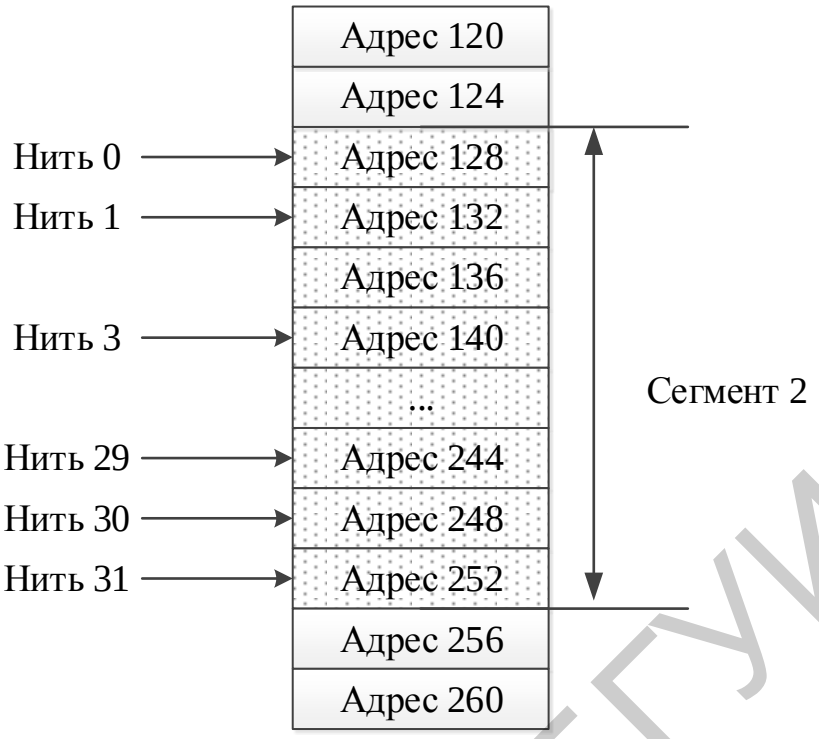

Рис. 3.3. Пример выравненного обращения к глобальной памяти

3.2. Оптимизация работы с разделяемой памятью

Разделяемая память делится на 32 блока (банка) фиксированного размера (размер слова – 32 бита), доступ к которым осуществляется одновременно. Все записываемые в разделяемую память данные распределяются по этим банкам циклически, т. е. каждое 33-е слово из массива типа int (32 бита) будет записываться в один банк разделяемой памяти. Если нити обращаются к 32-битным словам в разных банках, латентность доступа будет минимальна. Если хотя бы одна из нитей не получила данные из-за конфликта доступа к банку (одновременное обращение к различным адресам одного банка), осуществляется повторный доступ к памяти, и данные догружаются. Данная процедура будет повторяться до тех пор, пока не будут разрешены все конфликты и загружены или сохранены все данные. В результате время доступа будет произведением порядка конфликта на латентность доступа к разделяемой памяти. Пример возникновения конфликта банков показан на рис. 3.4.

В случае обращения к одному и тому же 32-битному слову в разделяемой памяти множеством нитей или непоследовательным обращением к разным банкам конфликт возникать не будет. Это продемонстрировано на рис. 3.5.

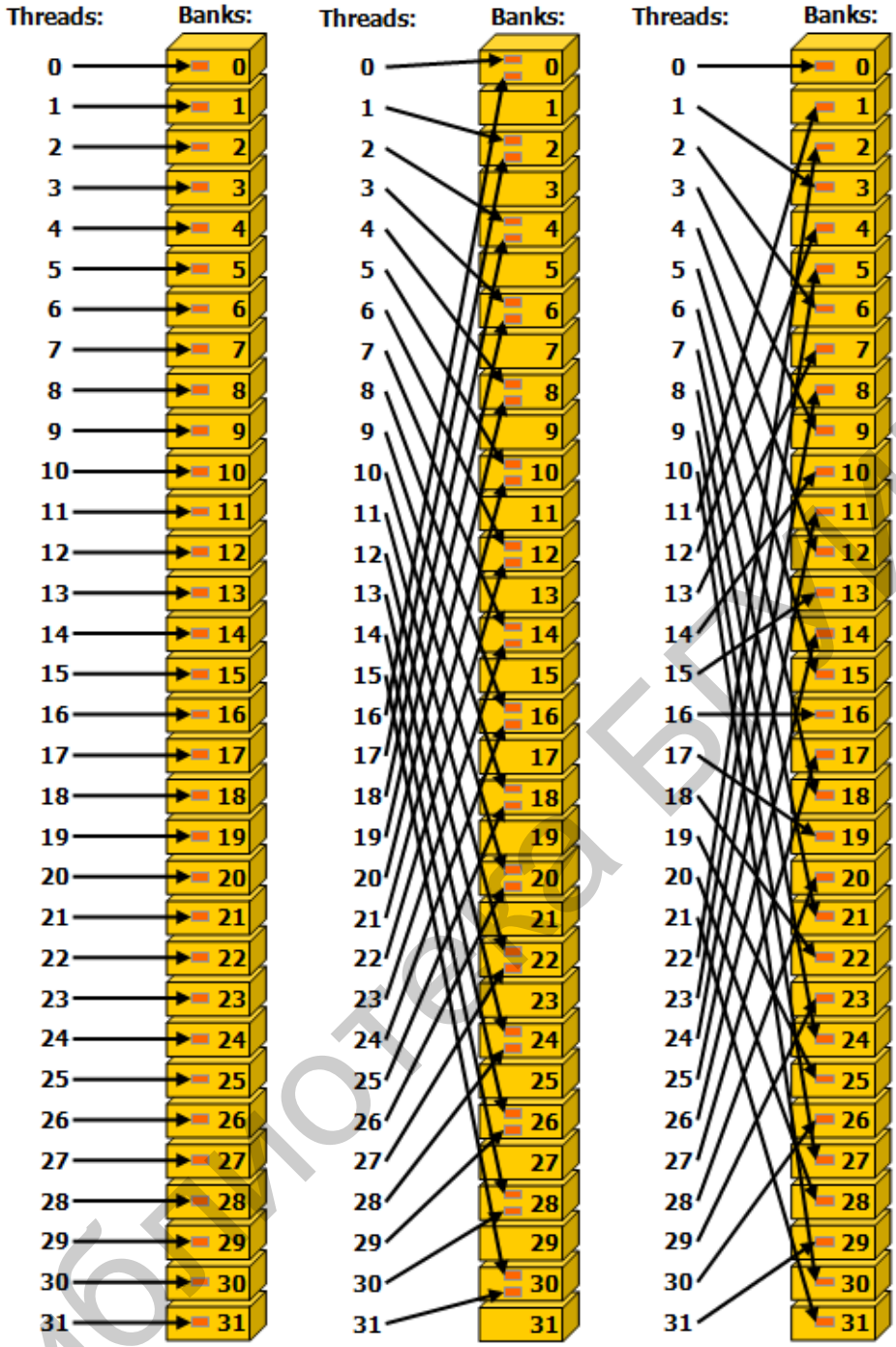

Рис. 3.4. Доступ к разделяемой памяти, линейная адресация с последовательным доступом:

а – к одному 32-битному слову (нет конфликта банков);

б – к двум 32-битным словам (конфликт банков 2-го порядка);

в – к трем 32-битным словам (нет конфликта банков)

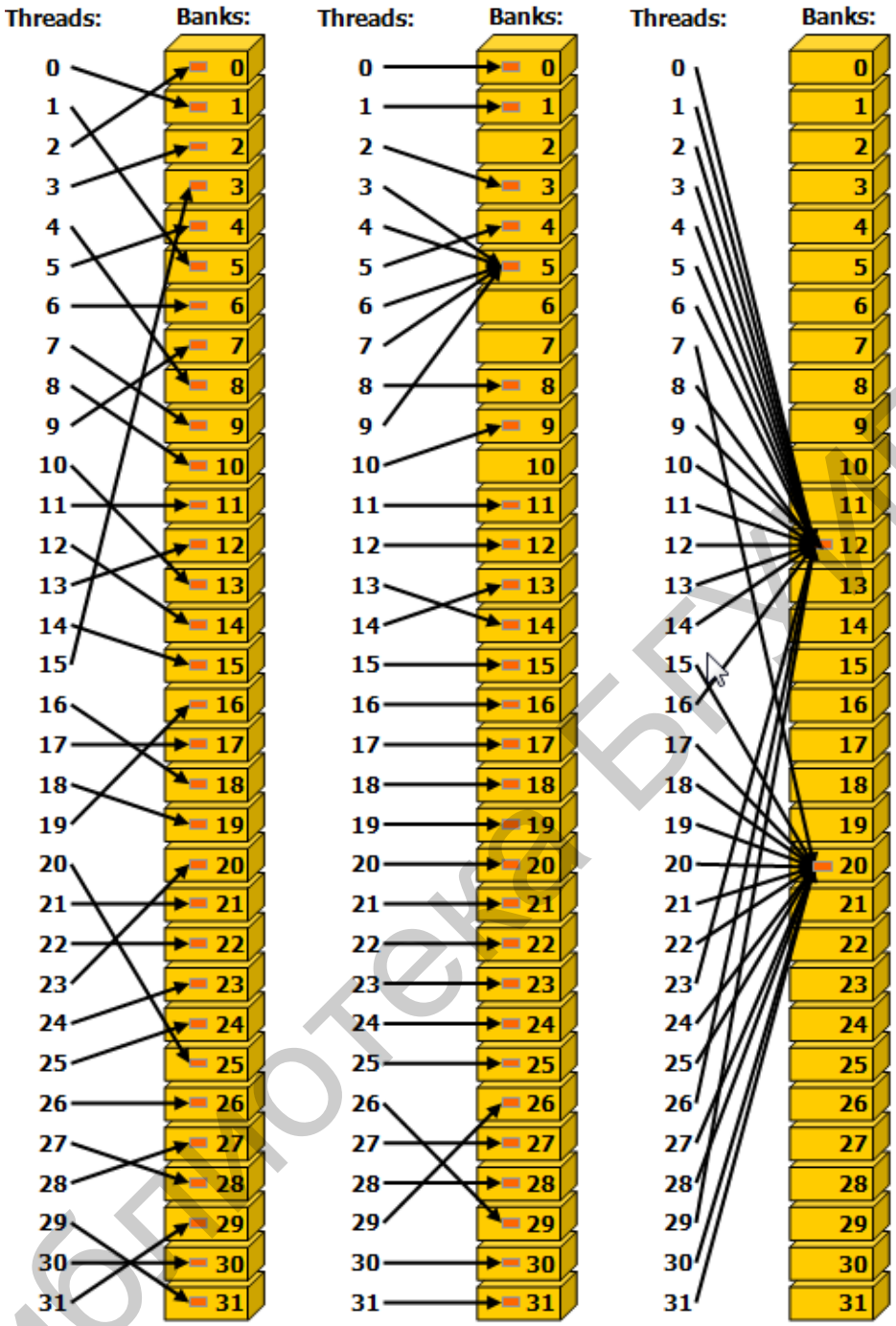

Рис. 3.5. Непоследовательное обращение нитей к разделяемой памяти (слова по 32 бита) без возникновения конфликта банков:

а – обращение к словам в случайном порядке;

б – обращение нитей 3, 4, 6, 7, 9 к одному слову в 5-м банке;

в – широковещательное обращение нитей к двум словам

Также при работе с разделяемой памятью необходимо помнить, что нити выполняются параллельно только в пределах одного варпа (32 нити с последовательными индексами), но не блока. Поэтому следует синхронизировать работу нитей с разделяемой памятью в случае, когда нить должна использовать данные, которые будут получены в результате работы другой нити в блоке. Синхронизация выполняется после любого действия или набора действий с разделяемой памятью с помощью вызова функции __syncthreads(). Эта функция служит барьером для всех нитей блока: пока каждая нить не придет к этой функции, остальные будут ждать.

3.3. Обзор структуры данных для реализации алгоритма

Для работы алгоритма кластеризации требуются следующие массивы (рис. 3.6):

– *objects* для объектов размером $N \cdot M$;

– *centers* для центров кластеров размером $K \cdot M$;

– *clusters* для номеров кластеров для объектов размерностью N;

– *objcount* для количества объектов каждого кластера размерностью K.

Здесь и далее N – число объектов по M признаков, K – число кластеров.

Все эти массивы располагаются в памяти GPU линейно и непрерывно.

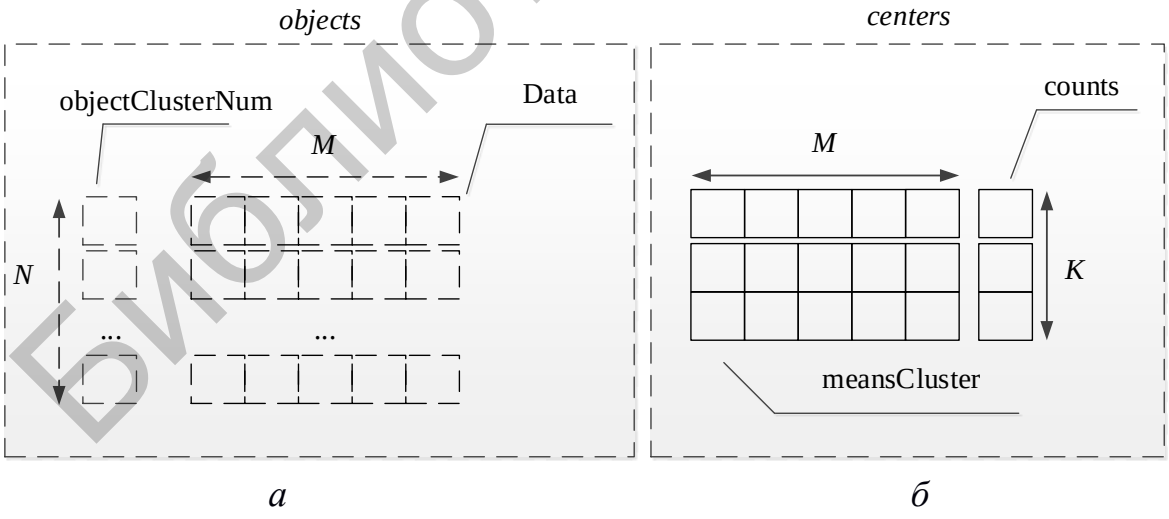

Рис. 3.6. Представление данных:
а – массив *objects*; *б* – массив *centers*

3.4. Конфигурация запуска ядра и распределение нагрузки между ядрами

Существует два варианта решения алгоритма *k*-средних на GPU:

– ядро, для которого одна нить полностью обрабатывает один объект и определяет кластер, к которому данный объект относится;

– ядро, для которого одна нить обрабатывает один признак и формирует сумму признаков объектов каждого из кластеров.

Конфигурация первого ядра:

– dim3 threadsFunc1(1, 64, 1);

– dim3 blocksFunc1(1, N / threadsFunc1.y, 1).

В результате данных действий формируется одномерный столбец, позволяющий обработать весь объем объектов (рис. 3.7).

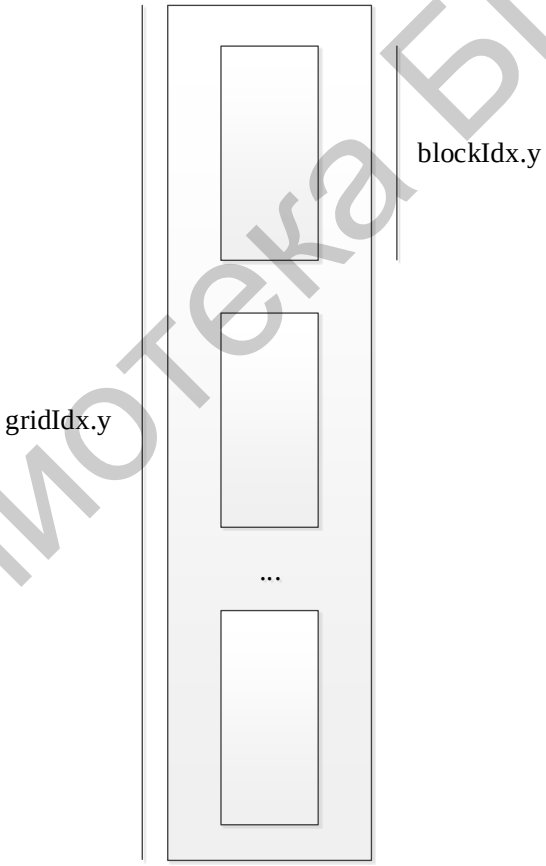

Рис. 3.7. Графическое представление обрабатываемых объектов

Конфигурация второго ядра:

– dim3 threadsFunc2(M, 1024 / M, 1);

– dim3 blocksFunc2(1, N / threadsFunc2.y, 1).

В результате формируется одномерная решетка блоков с двумерными блоками в нем. Блок строится исходя из максимального числа нитей (в данном случае – 1024 нити) для максимальной загрузки видеокарты. В результате деления 1024 на количество признаков M для одного объекта получается число объектов, которое будет обрабатывать один блок. Решетка блоков строится в зависимости от того, сколько всего объектов и сколько из них сможет обработать один блок (рис. 3.8).

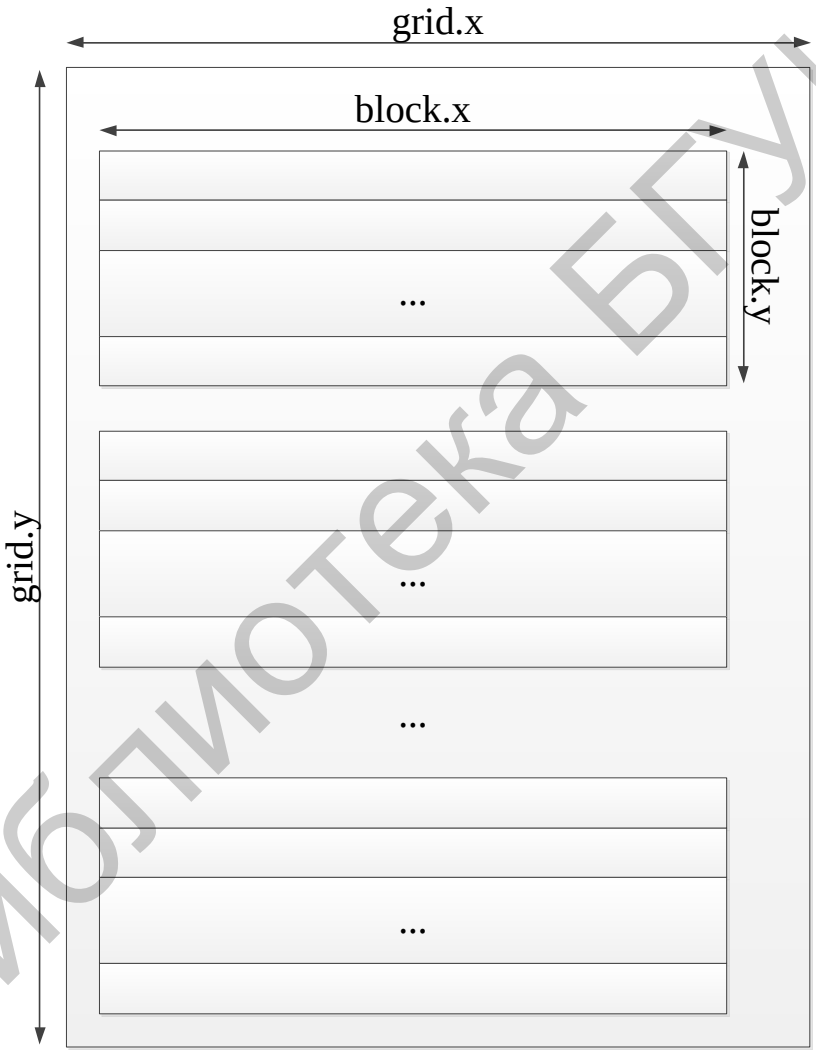

Рис. 3.8. Графическое представление обрабатываемых объектов

В данной реализации следует учитывать архитектуру GPU, на которой осуществляется запуск. Для архитектуры Fermi ограничение по оси *Y* для решетки блоков составляет 65535, что при обработке больших наборов данных

приводит к тому, что запуск будет некорректен. К примеру, если число признаков $M = 100$, блок будет сконфигурирован как $100 \cdot 10$ нитей. В результате один блок будет обрабатывать 10 объектов, таким образом, $65535 \cdot 10 = 655350$ объектов сможет обработать одна решетка блоков, что немного для задач в области обработки больших объемов данных. В этом случае для новых архитектур (Kepler, Maxwell) следует поменять оси *X* и *Y* местами, что позволит получить ограничение не в 65535, а $2 \cdot 10^{32}$.

Для архитектуры Fermi задача решается следующим образом: в цикле последовательно запускается несколько ядер, а в качестве параметров указывается дополнительное смещение по данным. При этом следует учитывать, что запуск ядра на последней итерации может быть неэффективен. Поэтому на последних итерациях рекомендуется дополнительно проанализировать имеющиеся данные и найти оптимальную конфигурацию.

3.5. Описание работы ядра, выполняющего кластеризацию объектов

Базовое ядро findMinDist() выполняет следующие задачи:

– подсчет расстояний от объекта к каждому из кластеров по формуле (1.1);

– определение минимального расстояния;

– подсчет количества объектов, принадлежащих каждому из кластеров.

Результатом работы функции являются (см. рис. 3.6):

– массив *clusters* – ключ, сопоставляющий номер кластера и объект;

– массив *objcount* – количество объектов, принадлежащих каждому из кластеров.

Данные, считанные из глобальной памяти, на время вычислений рекомендуется размещать на регистрах. Блок-схема алгоритма представлена на рис. 3.9.

Рис. 3.9. Блок-схема базового ядра

3.6. Алгоритм работы ядра, выполняющего частичное суммирование

Задача ядра calcSums() заключается в формировании суммы признаков объектов каждого из кластеров. В дальнейшем, после формирования вектора сумм расстояний, данные делятся на количество объектов, и формируются новые центры кластеров (ядро готовит данные для вычисления среднего арифметического в соответствии с алгоритмом).

Результатом работы функции является массив partSum – частичные суммы (рис. 3.10).

Поскольку ядро читает данные из глобальной памяти последовательно, дополнительных манипуляций с чтением через разделяемую память не требуется. В то же время рекомендуется использовать разделяемую память для частичного суммирования нескольких объектов в пределах блока в связи со следую-

щим условием. В алгоритме один блок состоит из максимально близкого к 1024 числа нитей. В соответствии с конфигурацией в зависимости от числа признаков M блок может обрабатывать $n = 1024/M$ объектов. Допустим, $M = 50$, $K = 5$ (число кластеров), тогда $n = 1024/50 = 20$ (в целочисленной арифметике). Таким образом, один блок будет обрабатывать 20 объектов, у каждого из которых по 50 признаков. Результатом работы является сумма всех 20 объектов по ключу их номера кластера (массив objectClusterNum), т. е. 20 объектов в зависимости от того, какому из кластеров они принадлежат, суммируются в одну из строк массива partSum. Таким образом, доступ к $K = 5$ строкам массива partSum будет осуществлен $M \cdot n = 50 \cdot 20 = 1000$ раз. В конце обработки блок данных из разделяемой памяти размером $K \cdot M = 5 \cdot 50 = 250$ чисел копируется в глобальную память. При этом объем данных, хранимый в глобальной памяти, для последующей обработки сокращается на 75 %. Графически это представлено на рис. 3.10.

Рис. 3.10. Графическое представление работы с разделяемой памятью

Все операции в данном ядре должны происходить атомарно. Это значит, что каждая операция сложения блокирует участок памяти, чтобы другая нить, которая также может попытаться записать свое значение, не получила доступ и не испортила значение. Блок-схема алгоритма ядра представлена на рис. 3.11.

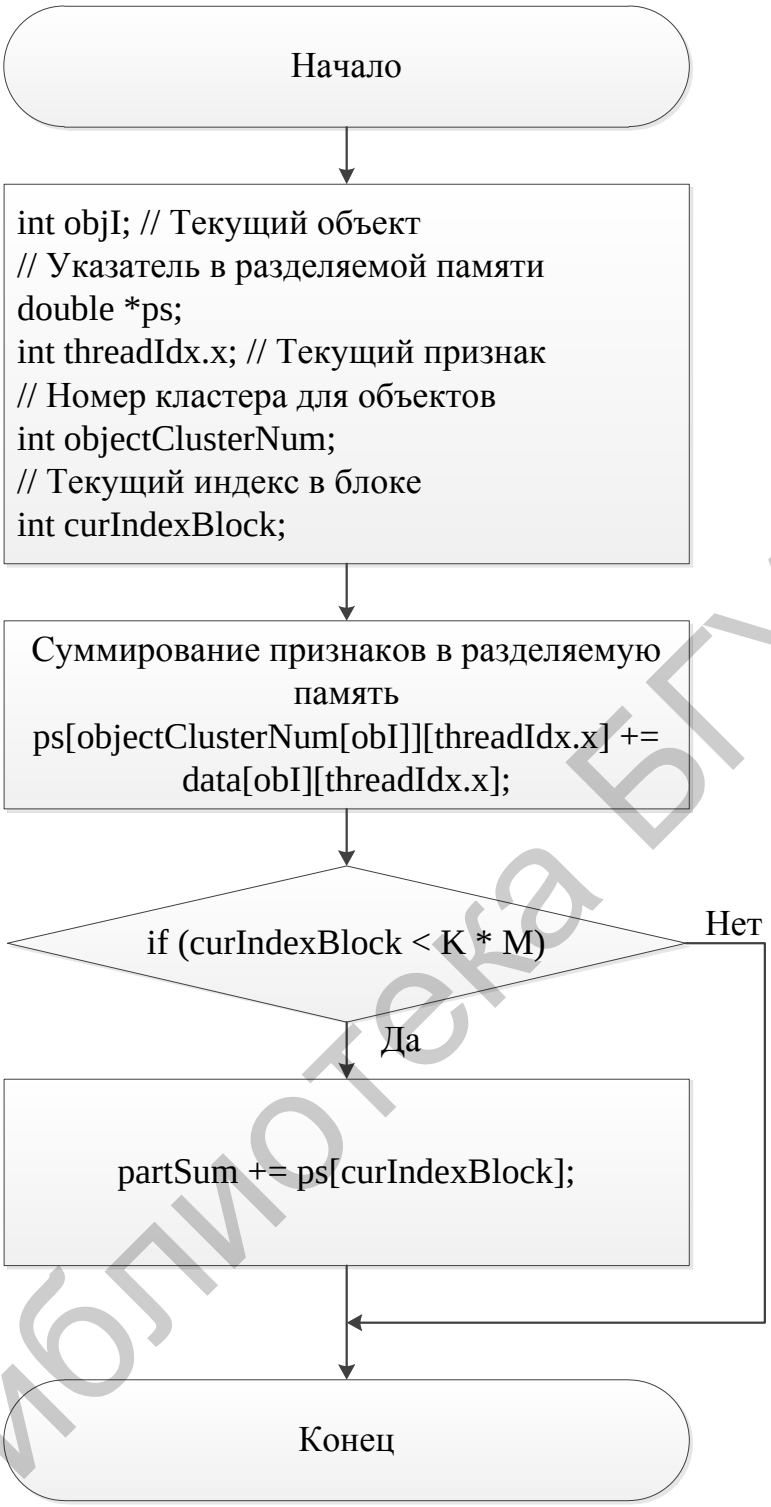

Рис. 3.11. Блок-схема ядра, формирующего частичные суммы

Найти сумму массива чисел можно также используя алгоритм параллельной редукции [4].

3.7. Задание

Модифицировать алгоритм для GPU в соответствии с рекомендациями по работе с памятью, представленными в теории. При этом студент должен определить и теоретически обосновать преимущества и недостатки предлагаемого подхода. По желанию студента допускается разработка собственной версии алгоритма, устраняющей найденные недостатки.

4. ТЕХНОЛОГИЯ CUDA STREAM

4.1. Описание технологии

Максимальная производительность возможна только при полной утилизации всех имеющихся ресурсов (в случае с технологией CUDA – CPU и GPU). Данный принцип лежит в основе управления нитями на GPU: как только нити варпа переходят в состояние ожидания получения результата, планировщик пытается найти варп, который может начать либо продолжить выполнение следующих операций, и переключается на него. В результате при правильном проектировании конфигурации системы возможно достигнуть идеального баланса производительности для GPU, когда задержки из-за ожидания выполнения операции сводятся к минимуму.

Кроме того, асинхронный запуск ядра позволяет одновременно с расчетами задействовать ресурсы CPU. Однако данный подход не позволяет полностью перекрыть операции копирования с хоста на устройство и наоборот. Для решения данной проблемы NVIDIA предлагает использовать CUDA Stream. Эта технология представляет собой очередь команд для GPU, где справедливы следующие правила:

– команды в рамках одного стрима (от англ. stream – «поток») выполняются строго последовательно;

– команды из разных стримов могут выполняться по мере освобождения необходимых ресурсов, т. е. возможно параллельное исполнение команд из разных стримов.

По умолчанию все команды выполняются в нулевом стриме, который создается автоматически системой. Стандартная последовательность команд (копирование с хоста на устройство, выполнение ядра, копирование результата с устройства на хост) при использовании только одного стрима показана на рис. 4.1.

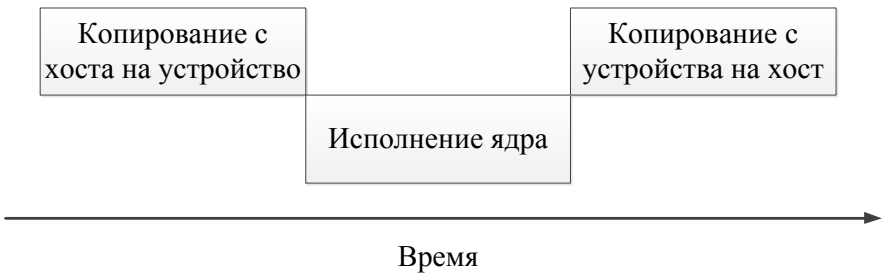

Рис. 4.1. Исполнение при использовании стрима по умолчанию

Используемый при этом код выглядит следующим образом:

```
cudaMemcpy(d_a, a, numBytes, cudaMemcpyHostToDevice);
kernel<<<1, N>>>(d_a);
cpuFunction();
cudaMemcpy(a, d_a, numBytes, cudaMemcpyDeviceToHost);
```

Основная проблема в данной ситуации – простаивание GPU при копировании данных и шины при исполнении ядра.

Для использования разных стримов требуется реорганизовать код следующим образом:

– ядро kernel для каждого стрима должно обрабатывать свой локальный участок данных независимо по отношению к остальным стримам (например, при обработке изображений каждый стрим обрабатывает свой рисунок или его часть);

– функции копирования заменить на их асинхронные версии;

– инициализировать необходимое число стримов.

Инициализация стрима выглядит следующим образом (все представленные команды будут относиться к одному стриму):

```
cudaStream_t stream1;
// создаем stream1
cudaStreamCreate(&stream1);
// асинхронное копирование на устройство
cudaMemcpyAsync(d_a, a, N, cudaMemcpyHostToDevice, stream1);
// запуск ядра
kernelStream<<<1, N, 0, stream1>>>(d_a);
… // дальнейшая работа со stream1, CPU
// разрушаем определенный stream1
cudaStreamDestroy(stream1);
```

Для определения того, что использование CUDA Stream может обеспечить повышение производительности, на этапе инициализации рекомендуется выполнить следующие действия:

– проверить флаг deviceOverlap в структуре cudaDeviceProp. В соответствии с документацией все устройства с Compute Capability версии 1.1 и новее данный флаг установлен в истинное значение;

– память на хосте должна быть определена как pinned.

Для синхронизации, например, со stream1 введена функция cudaStreamSynchronize(stream1), которая блокирует хост до тех пор, пока не будет полностью выполнены все команды для stream1. Для того чтобы узнать, завершены ли команды стрима без блокировки хоста, используется функция cudaStreamQuery(stream1).

Новая версия кода с использование nStreams стримов показана ниже (на рис. 4.2 показан результат перекрытия операций в идеальных условиях):

```
for (int i = 0; i < nStreams; ++i) {
  int offset = i * streamSize;
  cudaMemcpyAsync(
        &d_a[offset],
        &a[offset],
        streamBytes,
        cudaMemcpyHostToDevice,
        stream[i]
  );
  kernel<<<…, stream[i]>>>(d_a, offset);
  ...
  cudaMemcpyAsync(
        &a[offset],
        &d_a[offset],
        streamBytes,
        cudaMemcpyDeviceToHost,
        stream[i]
  );
}
```

Для отслеживания прироста производительности программы на CUDA NVidia предоставляет в составе CUDA Toolkit специальную программу-профилировщик. Для ОС Windows профилировщик встраивается в IDE Visual Studio, в ОС Linux его можно запустить командой nvvp.

Рис. 4.2. Перекрытие операций с использование различных стримов

4.2. Задание

Для закрепления материала выполнить следующие действия:

– добавить поддержку технологии CUDA Stream в программу;

– продемонстрировать с помощью профилировщика CUDA перекрытие операций копирования.

5. МЕТОДЫ ВЗАИМОДЕЙСТВИЯ CUDA И MPI

5.1. Введение в MPI

MPI (англ. Message Passing Interface) – программный интерфейс для передачи информации, который позволяет обмениваться сообщениями между процессами, выполняющими одну задачу. Процессы могут находиться как на одном физическом хосте, так и на отдаленных хостах, связанных по сети.

Базовым механизмом связи между MPI-процессами является передача-прием сообщений. Сообщение несет в себе информацию, которая позволяет принимающей стороне осуществлять выборочный прием.

Основные понятия MPI:

– **ранг** – идентификатор процесса, участвующего в вычислениях, задается внутри системы MPI автоматически;

– **коммуникатор** – идентификатор некоторой группы процессов, с помощью которого можно посылать сообщения для этой группы процессов. Стандартный коммуникатор, в который изначально входят все процессы, участвующие в вычислениях, – MPI_COMM_WORLD.

Перед началом работы с MPI требуется выполнить инициализацию интерфейса:

```
int MPI_Init( int *argc, char ***argv );
```

где argc – указатель на число аргументов;
agrv – указатель на вектор аргументов.

Определение ранга процесса:

```
int MPI_Comm_rank( MPI_Comm comm, int *rank );
```

где comm – коммуникатор;
rank – ранг процесса.

Определение числа процессов, относящихся к коммуникатору:

```
int MPI_Comm_size( MPI_Comm comm, int *size );
```

где size – число процессов в группе коммуникатора comm.

Завершение работы MPI:

```
int MPI_Finalize( void );
```

Пример программы на MPI:

```
int main(int argc, char **argv) {
     int MPIprocRank, MPIprocCount;
     MPI_Init(&argc, &argv);

     MPI_Comm_rank(MPI_COMM_WORLD, &MPIprocRank);
     MPI_Comm_size(MPI_COMM_WORLD, &MPIprocCount);

     printf("Proc rank [%d] say'Hello world!'\n", MPIpro-
cRank);
     if (MPIprocRank == 0)
          printf("Proc count: %d\n", MPIprocCount);

     MPI_Finalize();
}
```

Запуск программы с параметрами

```
mpirun -np 4 mpiHelloWorld.exe
```

приведет к выводу следующей информации:

```
Proc rank [3] say 'Hello world!'
Proc rank [1] say 'Hello world!'
Proc rank [2] say 'Hello world!'
Proc rank [0] say 'Hello world!'
Proc count: 4
```

5.2. Парные операции MPI

Стандарт MPI включает две основные парные операции для взаимодействия между процессами – MPI_Send и MPI_Recv. При этом взаимодействие между процессами осуществляется с помощью сообщений по схеме «точка – точка». Данные операции могут быть как синхронными, так и асинхронными.

Функция для передачи сообщения определенному процессу:

```
int MPI_Send(
     void * message,
     int count,
     MPI_Datatype mpi_type,
     int rankDest,
     int tag,
     MPI_comm comm
);
```

где message – буфер для отправки данных;
count – количество отправляемых данных;
mpi_type – тип отправляемых данных;
rankDest – идентификатор процесса, которому отправляются данные;
tag – маркер сообщения;
comm – коммуникатор.

Функция для приема сообщения определенным процессом:

```
int MPI_ Recv(
     void *message,
     int count,
     MPI_Datatype mpi_type,
     int rankSource,
     int tag,
     MPI_comm comm  ,
     MPI_Status *status
);
```

где status – дополнительная информация о принимаемом сообщении и его статусе.

5.3. Коллективные операции MPI

Стандарт MPI предоставляет коллективные операции в пределах определенных коммуникаторов. Данный тип операций исполняется всеми процессами без исключения. Пока все процессы не исполнят коллективную операцию, остальные ожидают. Например, MPI предоставляет для синхронизации процессов функцию

```
int MPI_Barrier( MPI_Comm comm );
```

которая задерживает все процессы, относящиеся к коммуникатору comm, не вызовут данную функцию.

Кроме того, могут быть полезны следующие MPI-функции:

– функция рассылки сообщения всем процессам, входящим в группу коммуникатора:

```
int MPI_Bcast(
     void *buffer,
     int count,
     MPI_Datatype datatype,
     int root,
     MPI_Comm comm
);
```

где buffer – буфер с сообщением для рассылки;

count – число отправляемых данных;

– функция рассылки равных фрагментов сообщения всем процессам, входящим в группу коммуникатора:

```
int MPI_Scatter(
     const void *sendbuf,
     int sendcount,
     MPI_Datatype sendtype,
            void *recvbuf,
     int recvcount,
     ...
     MPI_Datatype recvtype,
     int root,
     MPI_Comm comm
);
```

– функция сборки равных фрагментов сообщения процессами, входящими в группу коммуникатора (параметры аналогичны функции MPI_Scatter):

```
int MPI_Gather(
     const void *sendbuf,
     int sendcount,
     MPI_Datatype sendtype,
```

```
    void *recvbuf,
    int recvcount,
    MPI_Datatype recvtype,
    int root,
    MPI_Comm comm
);
```

– функция, выполняющая заданное действие *op* над каждым элементом из sendbuf от всех процессов, входящих в группу коммуникатора comm (результат сохраняется в recvbuf):

```
int MPI_Reduce(
    const void *sendbuf,
    void *recvbuf,
    int count,
    MPI_Datatype datatype,
    MPI_Op op,
    int root,
    MPI_Comm comm
);
```

В качестве допустимых операций *op* могут быть следующие:

– MPI_SUM – суммирование значений;

– MPI_MIN – поиск минимального значения;

– MPI_MAX – поиск максимального значения.

5.4. Реализация алгоритма на MPI

При обработке алгоритмом *k*-средних данные могут быть разделены на множество частей, что дает возможность дополнительного распараллеливания алгоритма. В предлагаемой реализации вычисления могут осуществляться на n узлах ($n \leq N$). При этом следует учитывать, что слишком маленькие объемы данных для одного узла, наоборот, приведут к замедлению вычислений, т. к. время расчетов будет гораздо меньше времени коммуникации узлов.

Для алгоритма *k*-средних количество объектов для каждого из узлов вычисляется как $N_{mpi} = N/n$, центры кластеров загружаются в полном объеме на каждый узел в количестве $K_{mpi} = K$, аналогично загружается в полном объеме число признаков $M_{mpi} = M$ (рис. 5.1).

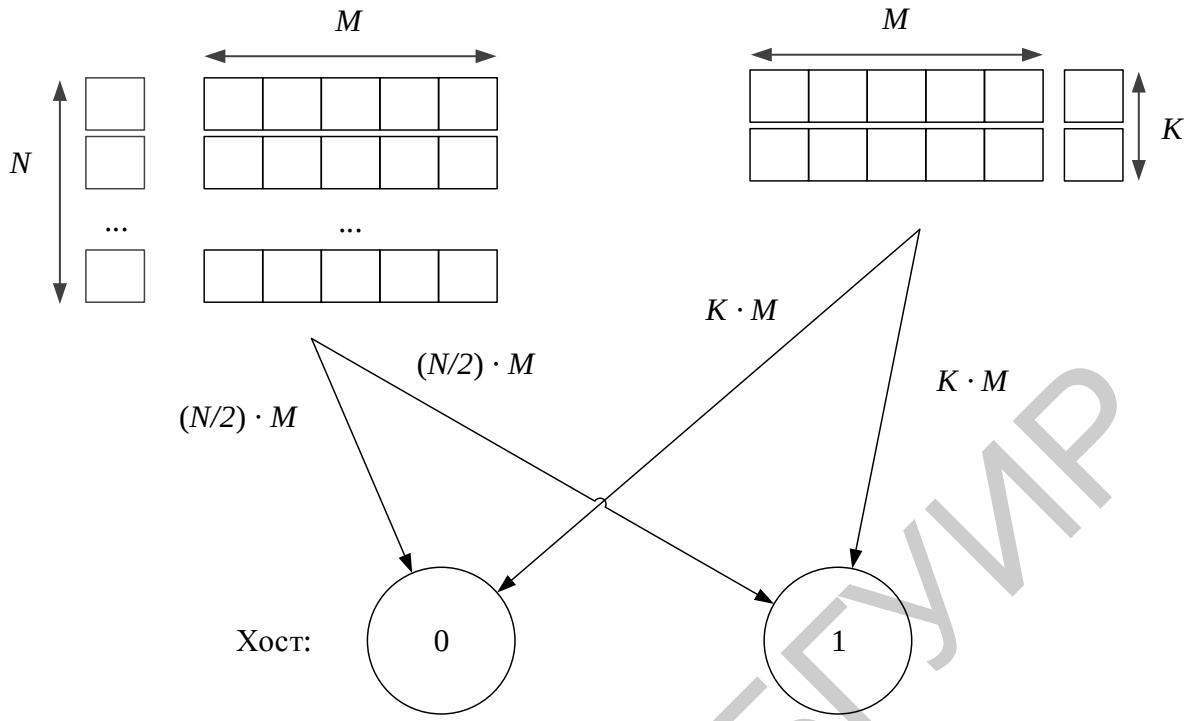

Рис. 5.1. Графическое представление разбиения данных

Дополнительно при разбиении на объекты требуется учитывать, что деление осуществляется в целочисленной арифметике, поэтому возможна ситуация, когда один из узлов будет обрабатывать больше данных, чем другие. К примеру, если на 100 объектов осуществляется 7 процессов, для равномерной загрузки каждый узел должен обработать 100/7 = 14,2 объекта, что предлагаемым алгоритмом не предусмотрено. Поэтому одному из процессов передается не 14, а 16 объектов.

Для распределения данных между процессами рекомендуется использовать расширенную функцию MPI_Scatterv:

```
int MPI_Scatterv(
      const void *sendbuf,
      const int *sendcounts,
      const int *displs,
      MPI_Datatype sendtype,
      void *recvbuf, int recvcount,
      MPI_Datatype recvtype,
      int root,
      MPI_Comm comm
);
```

Основное отличие данной функции от стандартной версии MPI_Scatter заключается в возможности указать массив со смещениями по данным и массив с количеством элементов для пересылки.

Пример использования функции MPI_Scatterv:

```
int main(int argc, char **argv) {
     int MPIprocRank, MPIprocCount;

     MPI_Init(&argc, &argv);

     MPI_Comm_rank(MPI_COMM_WORLD, &MPIprocRank);
     MPI_Comm_size(MPI_COMM_WORLD, &MPIprocCount);

     int counts[] = { 3, 1, 2 };
     int offsets[] = { 0, 3, 4 };

     char * data;
     if (MPIprocRank == 0) {
          data = (char *)malloc(7 * sizeof(char));
          sprintf(data, "123456");
     }

     char * recv = (char *)malloc((counts[MPIprocRank] + 1) *
                    sizeof(char));
     MPI_Scatterv(data, counts, offsets, MPI_CHAR, recv,
               counts[MPIprocRank], MPI_CHAR,     0,
              MPI_COMM_WORLD);

     recv[counts[MPIprocRank]] = '\0';
     printf("[%d] %s\n", MPIprocRank, recv);
     MPI_Finalize();
     return 0;
}
```

Массив counts определяет число элементов исходного массива, который получит каждый процесс, при этом ранг процесса соответствует индексу массива. Массив offsets содержит смещения по исходному массиву, начиная с которого будет осуществляться пересылка данных. После выполнения программы с аргументами –np 3 получается следующий вывод:

```
[1] 4
[2] 56
[0] 123
```

5.5. Синхронизация нескольких GPU на одном хосте

На одном физическом хосте может быть размещено более 1 видеокарты, поэтому ситуация с распределением объектов в рамках хоста также должна быть обработана.

Один из вариантов распределения нагрузки на GPU – переложить работу на уровень MPI. В этом случае в файле конфигурации указывается, что на данном хосте может быть исполнено до двух процессов (рис. 5.2).

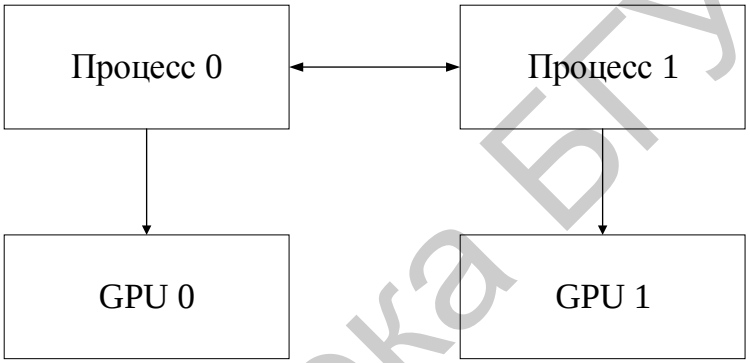

Рис. 5.2. Один процесс на один GPU

Данная реализация отличается простотой разработки, однако предполагает, что для дальнейшей обработки этим двум процессам нужно будет собрать данные между собой, что влечет дополнительные расходы на коммуникации.

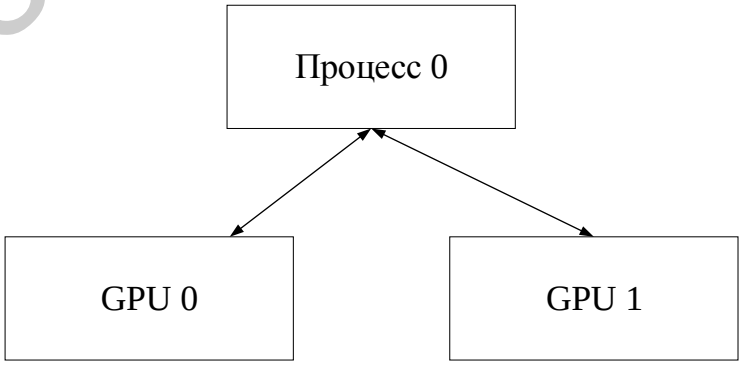

Рис. 5.3. Один процесс для всех GPU

Второй вариант распределения нагрузки на GPU – разработка в рамках одного процесса, который самостоятельно разбивает данные и синхронизирует видеокарты между собой (рис. 5.3).

Один из возможных примеров синхронизаций GPU в рамках одного процесса выглядит следующим образом:

```
cudaGetDeviceCount(&GPU_count);
for (int i = 0; i < countGPU; i++) {
     cudaSetDevice(i);

     // запуск ядра на исполнение в i-м stream
}

for (int i = 0; i < countGPU; i++) {
     cudaSetDevice(i);
     cudaDeviceSynchronize();
     /*Тут копирование результатов с GPU на хост*/
}
```

В данном примере в первом цикле по очереди выбираются GPU, и в отдельном стриме запускается ядро. Во втором цикле выполняется синхронизация выбранного устройства. Это гарантирует, что ядро, выполняющееся на данном GPU, завершило свою работу, и можно получать с него данные.

Полная версия блок-схемы алгоритма, предлагаемая для реализации, представлена на рис. 5.4.

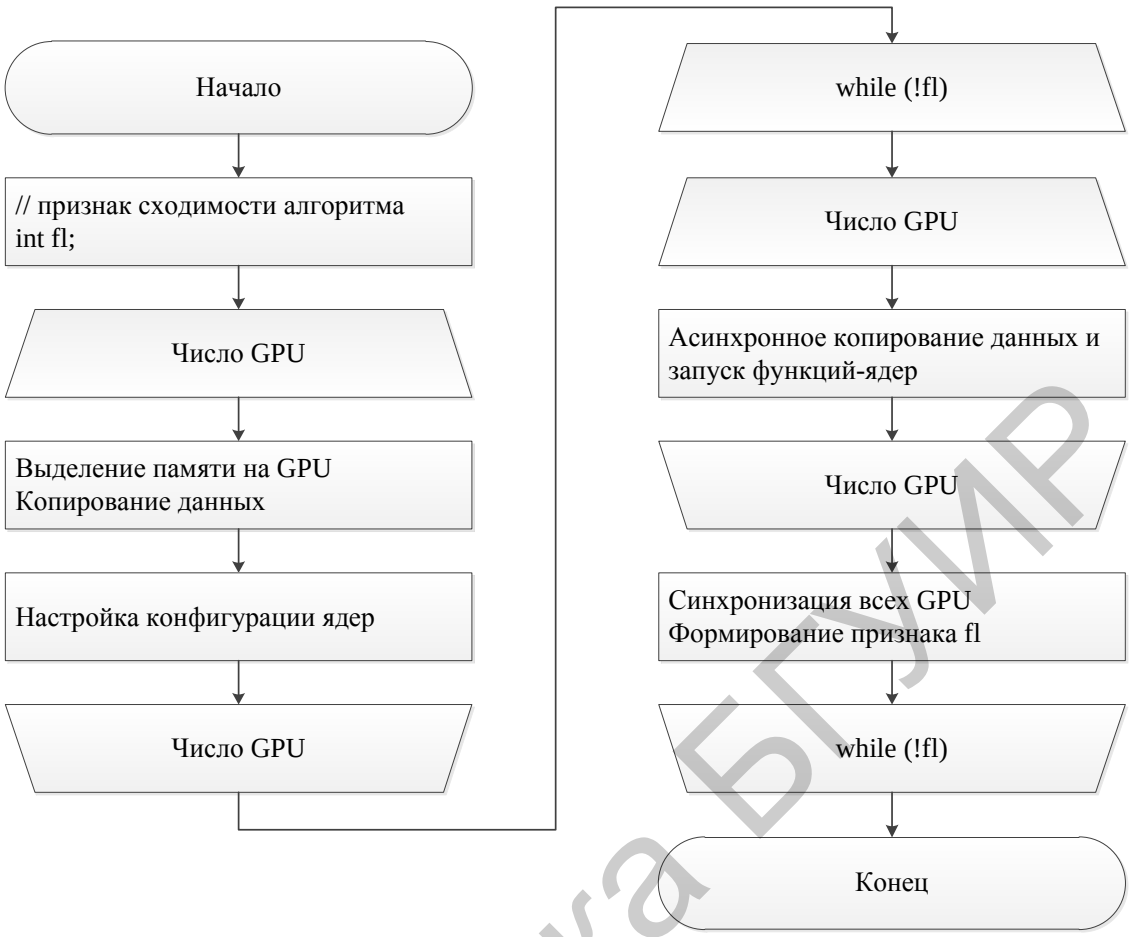

Рис. 5.4. Блок-схема алгоритма на GPU

5.6. Чтение исходных данных из файла с помощью MPI

Обзор структуры входных данных представлен в прил. 2.

При параллельном чтении файла всеми процессами каждый процесс открывает файл в режиме чтения и читает нужную ему часть. Поскольку файл текстовый, необходимо применить алгоритм смещения по строкам. Допустим, суммарное число процессов 4. И нулевой, и все остальные процессы вычисляют два смещения: откуда прочитать данные текущему и последующему процессам. Однако в отличие от нулевого процесса все остальные могут попасть не в начало строки, т. к. в файле строки имеют разную длину, и вычислить точные смещения не представляется возможным. Поэтому каждый процесс, попадающий не в начало строки, смещается на начало следующей и начинает читать с нее. Следовательно, необходимо определять конечную границу считываемой последовательности: если при завершении считывания для данного процесса оказывается не достигнут конец строки, продолжается считывание до достижения

символа перевода каретки. Таким образом, все четыре процесса параллельно вычитают примерно одинаковый объем данных для обработки. В этом случае функция MPI_Scatterv не нужна, т. к. каждый процесс сразу читает данные в своем ОЗУ.

Допустим, имеется файл *N*, *M*, *K* и четыре вычислительных узла (процесса) n, размер файла S. Разделим файл поровну на количество узлов $S_n = S/n$. Смещение, с которого начнет читать каждый процесс, равно произведению $S_n \cdot P_r$, где P_r – ранг процесса. Смещение, на котором закончится чтение, равно $S_n \cdot (P_r + 1)$, или будет достигнут конец файла. Прочитав данные до окончания смещения, каждый процесс проверяет, является ли это концом строки. Если нет – строка будет дочитана до конца. Графически процесс представлен на рис. 5.5. На рис. 5.6 отображена ситуация, когда процесс попадает не в начало строки.

Также при параллельном чтении был задействован MPI_Reduce с операциями MPI_MIN и MPI_MAX для сбора с узлов максимальных и минимальных признаков в их частях файла и вычисления этих чисел для всего файла.

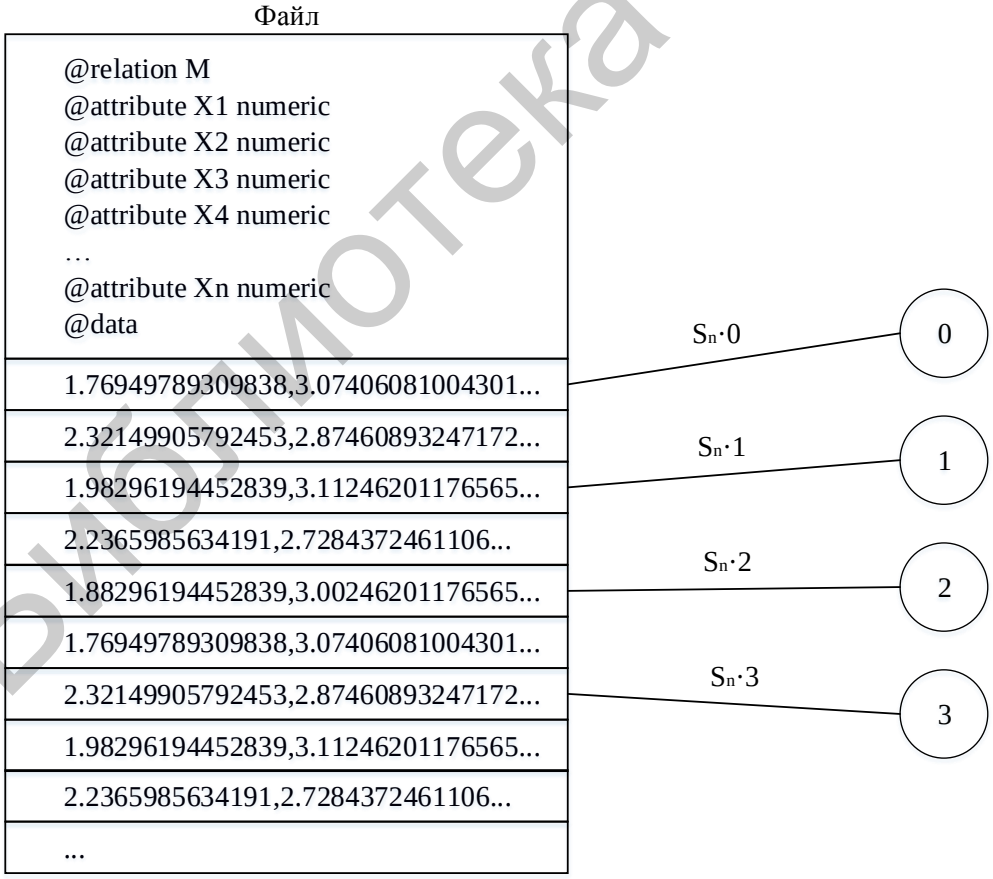

Рис. 5.5. Параллельное чтение файла четырьмя процессами

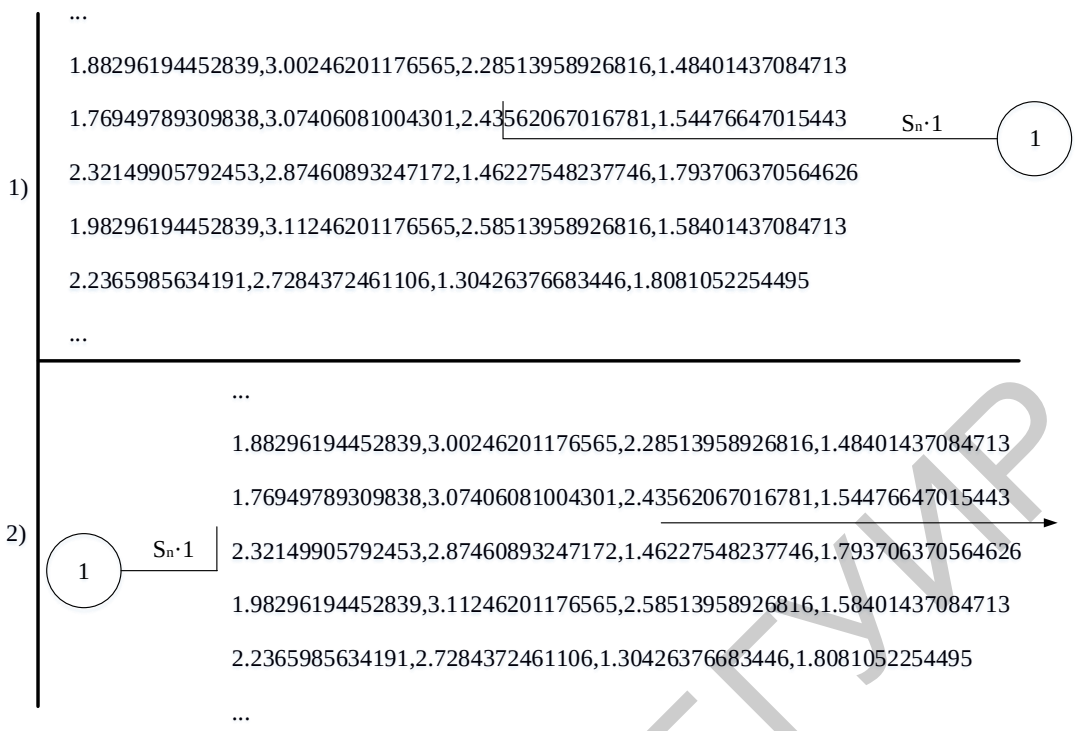

Рис. 5.6. Переход на новую строку при попадании не в начало строки

5.7. Задание

Для закрепления материала выполните следующие действия:

1) реализовать распределение кластеризации с использованием средств MPI;

2) учесть при реализации возможность использования нескольких GPU на хосте;

3) организовать параллельное считывание файла данных всеми MPI-процессами;

4) замерить время выполнения расчета кластеризации и время чтения данных из файла;

5) построить графики зависимости времени вычисления:

– от количества процессов MPI;

– от количества объектов;

– от количества признаков;

– от количества кластеров;

6) построить график зависимости времени доступа к данным от количества MPI-процессов. Для сравнения вывести на графики аналогичные показатели для CPU-реализации алгоритма. Пример графика представлен в прил. 3.

ПРИЛОЖЕНИЕ 1
Запуск программы на кластере

В лаборатории БГУИР на кластере установлен механизм PBS Torque для запуска задач на кластере. Запуск на вычисление возможен при получении SSH-доступа к кластеру, затем выполняется распределение на *n* процессов с применением механизма Torque (система взаимодействует с Open MPI). При этом Torque контролирует, чтобы задача не попала на хост, у которого нет нужных ресурсов.

Таким образом, для запуска потребуется:

– SSH-доступ к кластеру, который выдает преподаватель;

– для работы под Windows – утилита Putty (или аналог) и файловый менеджер с возможностью подключения по sftp; для работы с кластером из Linux достаточно утилит ssh и scp.

В окне Putty вводится IP-адрес кластера в строку Host Name (рис. П.1.1), например, 192.168.11.180.

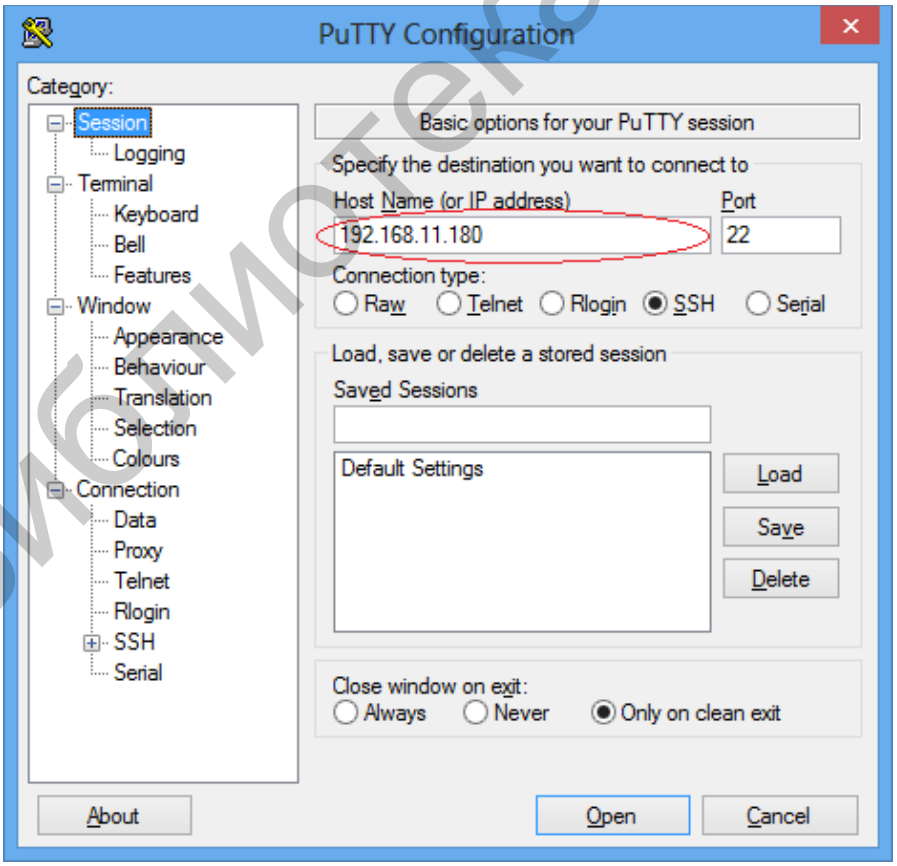

Рис. П.1.1. Ввод IP-адреса кластера в Putty

При этом в меню Window → Translation следует изменить Remote character set на UTF-8 (рисунок П.1.2) для корректного отображения кириллицы.

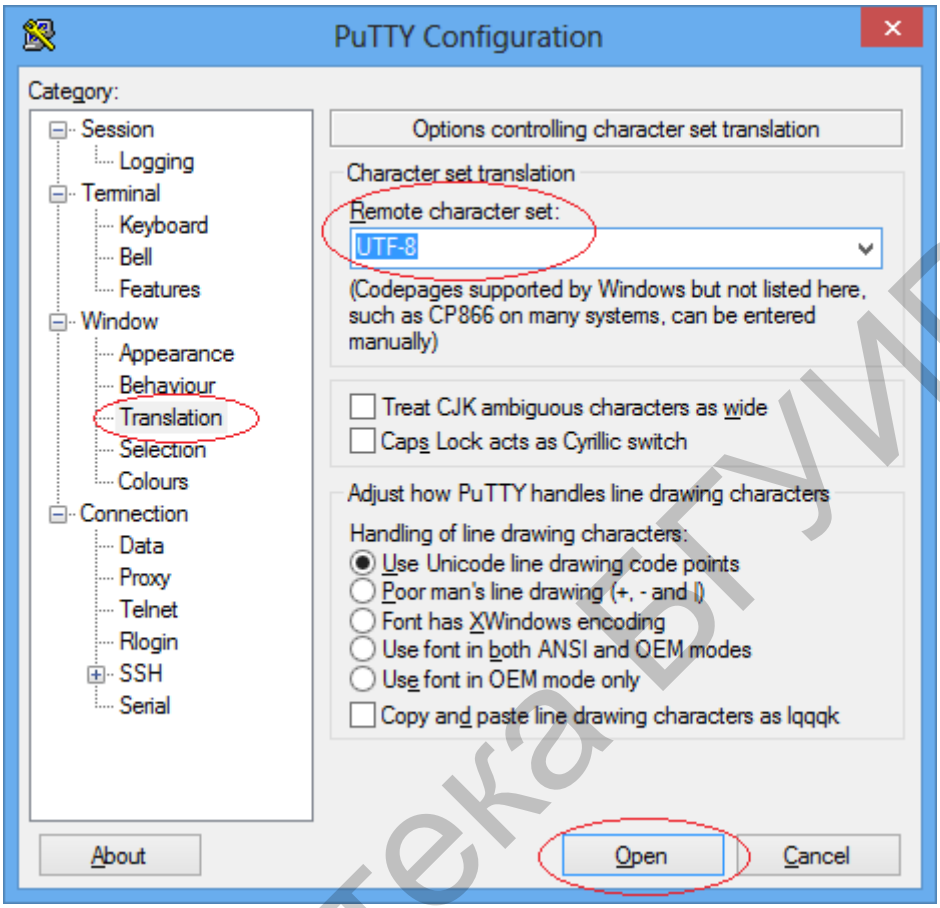

Рисунок П.1.2. Настройка кодировки и подключение

После нажатия кнопки Open вводятся имя пользователя и пароль для доступа. Для Linux в терминале вводится

```
ssh -l ваше_имя_пользователя 192.168.11.180
```

Перед началом работы с кластером следует сгенерировать SSH-ключи для доступа к семи узлам кластера:

```
[test@cluster508 ~]$ sh ~/pbs_run_sample/gen-keys.sh
```

Для запуска программы на кластере требуется скопировать код программы в домашнюю директорию, скомпилировать проект, после чего настраивается

конфигурация для Torque с целью запуска на нескольких узлах. Для этого создается файл mpi-run.pbs с настройкой прав доступа:

```
[test@cluster508 work]$ > mpi_run.pbs && chmod a+x mpi_run.pbs
```

Настройка конфигурации запуска выглядит следующим образом:

```
#!/bin/bash
#PBS -N run
#PBS -l nodes=3:ppn:=1,walltime=24:00:00
#PBS -l mem=1gb
#PBS -j oe
cd $PBS_O_WORKDIR
/usr/lib64/openmpi/bin/mpirun -hostfile $PBS_NODEFILE \
 ~/work/kernel
```

Вторая строка содержит имя задачи. Третья строка указывает, что задача будет исполняться на трех хостах по одному процессу на каждом, максимальное время исполнения – 24 часа. Четвертая строка указывает, сколько выделить ОЗУ для задачи. В пятой строке указывается, что следует перенаправить stderr в stdout. В шестой строке переходим в рабочую директорию PBS Torque, а в седьмой запускаем тестовую программу kernel из папки ~/work (рис. П.1.3).

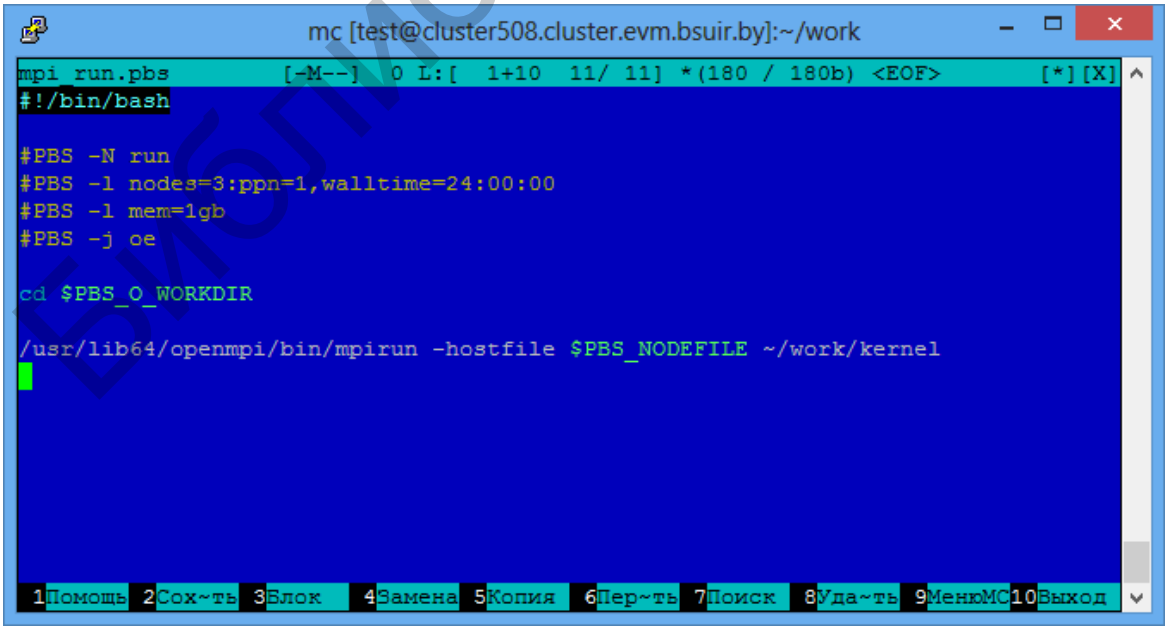

Рисунок П.1.3. Пример конфигурационного файла запуска задачи на кластере

Чтобы запустить задачу, необходимо в консоли набрать команду запуска конфигурационного файла:

```
[test@cluster508 work]$ qsub mpi_run.pbs
```

Затем управление передается PBS Torque. В ответ он начинает работу и выдает идентификатор задачи – «1090.pbs.cluster.evm.bsuir.by» (число 1090 приведено для примера).

Посмотреть, в каком статусе находится запущенная задача, можно с помощью команды qstat:

```
[test@cluster508 work]$ qstat

Job id          Name      User      Time Use S Queue
----------------------------------------- - -----
1090.ps         run       test             0 R batch
```

Если задача завершилась, вывод qstat будет пустым. Удалить задачу из очереди или прекратить ее выполнение можно по идентификатору задачи, выполнив команду

```
[test@cluster508 work]$ qdel 1090.pbs
```

После завершения задачи в папке ~/work появится файл run.o1090 – это выходной файл, в который были перенаправлены потоки stderr и stdout.

ПРИЛОЖЕНИЕ 2
Структура входных данных

Входной файл для алгоритма *k*-средних имеет текстовый формат .arff. В первых строках указаны признаки, затем располагаются объекты – каждая строка один объект, в строке через запятую перечислены признаки в формате с плавающей запятой двойной точности.

Пример содержимого файла $N = 10$, $M = 4$, $K = 2$:

```
@relation M
@attribute X1 numeric
@attribute X2 numeric
@attribute X3 numeric
@attribute X4 numeric
@data
1.76949789309838,3.07406081004301,2.43562067016781,1.54476647015443
2.32149905792453,2.87460893247172,1.46227548237746,1.793706370564626
1.98296194452839,3.11246201176565,2.58513958926816,1.58401437084713
2.2365985634191,2.7284372461106,1.30426376683446,1.8081052254495
1.88296194452839,3.00246201176565,2.28513958926816,1.48401437084713
1.76949789309838,3.07406081004301,2.43562067016781,1.54476647015443
2.32149905792453,2.87460893247172,1.46227548237746,1.793706370564626
1.98296194452839,3.11246201176565,2.58513958926816,1.58401437084713
2.2365985634191,2.7284372461106,1.30426376683446,1.8081052254495
1.88296194452839,3.00246201176565,2.28513958926816,1.48401437084713
```

ПРИЛОЖЕНИЕ 3
Пример графика

На рисунке П.3.1 представлен пример графика, иллюстрирующий зависимость времени вычисления на CPU и GPU от количества узлов (хостов), участвующих в вычислениях, для набора тестовых данных $N = 5 \cdot 10^6$, $M = 100$, $K = 50$.

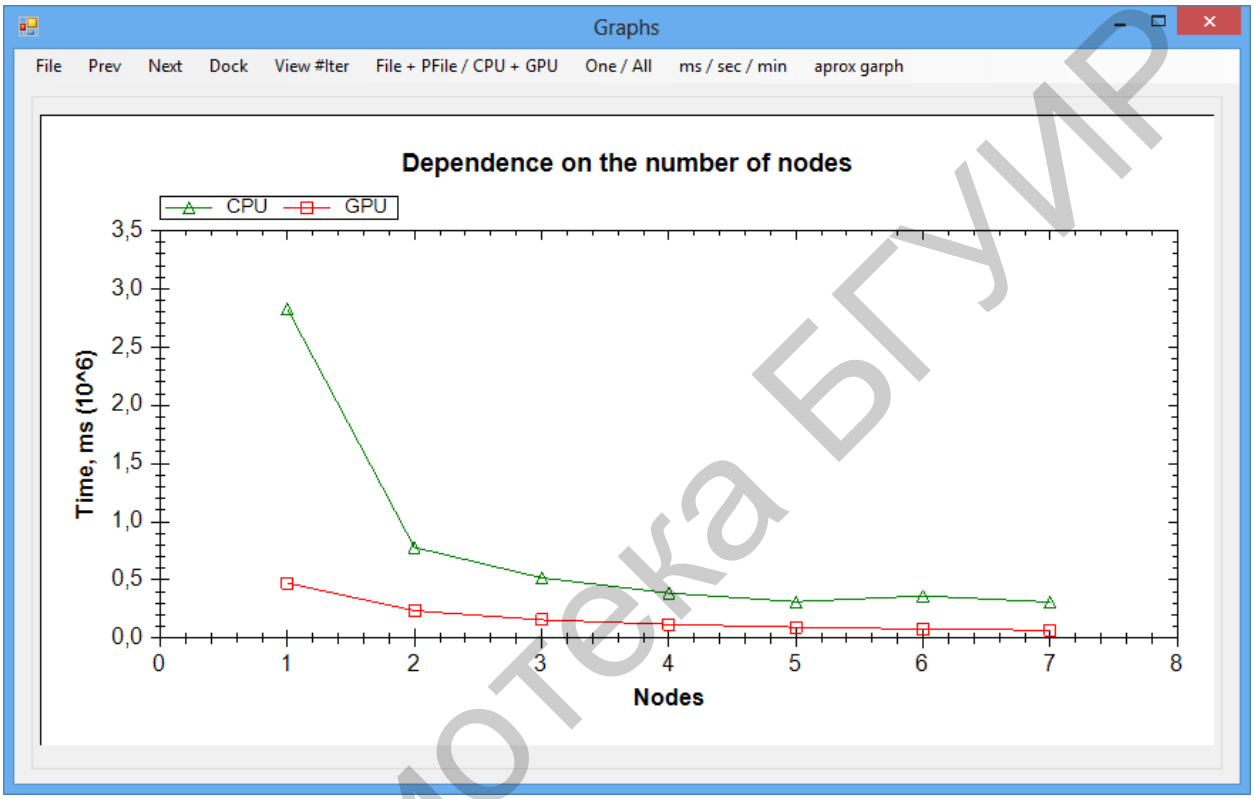

Рис. П.3.1. График зависимости времени от количества хостов, участвующих в вычислениях

ЛИТЕРАТУРА

1. CUDA C Programming Guide [Электронный ресурс]. – 2015. – Режим доступа : http://docs.nvidia.com/cuda/cuda-c-programming-guide/.

2. CUDA C Best Practices Guide [Электронный ресурс]. – 2015. – Режим доступа : http://docs.nvidia.com/cuda/ cuda-c-best-practices-guide/.

3. Параллельные вычисления на GPU. Архитектура и программная модель CUDA / А. Боресков [и др.]. – М. : Изд-во МГУ, 2015.

4. Attribute-Relation File Format (ARFF) [Электронный ресурс]. – 2015. – Режим доступа : http://www.cs.waikato.ac.nz/ml/weka/arff.html.

5. Weka 3: Data Mining Software in Java [Электронный ресурс]. – 2015. – Режим доступа : http://www.cs.waikato.ac.nz/ml/weka/.

6. Parallel Reduction in CUDA [Электронный ресурс]. – 2015. – Режим доступа : http://docs.nvidia.com/cuda/samples/6_Advanced/reduction/doc/reduction.pd

Св. план 2016, поз. 39

Учебное издание

Демидчук Алексей Иванович
Перцев Дмитрий Юрьевич
Татур Михаил Михайлович
Кришталь Дмитрий Викторович

ИНТЕЛЛЕКТУАЛЬНАЯ ОБРАБОТКА БОЛЬШИХ ОБЪЕМОВ ДАННЫХ НА ОСНОВЕ ТЕХНОЛОГИЙ MPI И CUDA. ЛАБОРАТОРНЫЙ ПРАКТИКУМ

ПОСОБИЕ

Редактор *А. К. Петрашкевич*
Корректор *Е. Н. Батурчик*
Компьютерная правка, оригинал-макет *А. В. Бас*

Подписано в печать 24.05.2017. Формат 60×84 1/16. Бумага офсетная. Гарнитура «Таймс».
Отпечатано на ризографе. Усл. печ. л. 3,6. Уч.-изд. л. 4,0. Тираж 100 экз. Заказ 111.

Издатель и полиграфическое исполнение: учреждение образования
«Белорусский государственный университет информатики и радиоэлектроники».
Свидетельство о государственной регистрации издателя, изготовителя,
распространителя печатных изданий №1/238 от 24.03.2014,
№2/113 от 07.04.2014, №3/615 от 07.04.2014.
ЛП №02330/264 от 14.04.2014.
220013, Минск, П. Бровки, 6