МИНОБРНАУКИ РОССИИ
ФЕДЕРАЛЬНОЕ ГОСУДАРСТВЕННОЕ БЮДЖЕТНОЕ ОБРАЗОВАТЕЛЬНОЕ УЧРЕЖДЕНИЕ
ВЫСШЕГО ПРОФЕССИОНАЛЬНОГО ОБРАЗОВАНИЯ
«НОВОСИБИРСКИЙ НАЦИОНАЛЬНЫЙ ИССЛЕДОВАТЕЛЬСКИЙ ГОСУДАРСТВЕННЫЙ УНИВЕРСИТЕТ» (НОВОСИБИРСКИЙ ГОСУДАРСТВЕННЫЙ УНИВЕРСИТЕТ, НГУ)

Кафедра систем информатики

ВЫПУСКНАЯ КВАЛИФИКАЦИОННАЯ РАБОТА БАКАЛАВРА

Сумбатянц Илья Ильич

Оптимизация производительности алгоритма избыточного кодирования данных

Направление подготовки 230100.62 ИНФОРМАТИКА И ВЫЧИСЛИТЕЛЬНАЯ ТЕХНИКА

Руководитель

Иртегов Д. В.

б/с, доцент ФИТ НГУ

……..
(подпись, дата)

Автор

Сумбатянц И. И.

ФИТ, 9201

..……..
(подпись, дата)

Новосибирск, 2013г.

МИНОБРНАУКИ РОССИИ

ФЕДЕРАЛЬНОЕ ГОСУДАРСТВЕННОЕ БЮДЖЕТНОЕ ОБРАЗОВАТЕЛЬНОЕ УЧРЕЖДЕНИЕ
ВЫСШЕГО ПРОФЕССИОНАЛЬНОГО ОБРАЗОВАНИЯ
«НОВОСИБИРСКИЙ НАЦИОНАЛЬНЫЙ ИССЛЕДОВАТЕЛЬСКИЙ ГОСУДАРСТВЕННЫЙ УНИВЕРСИТЕТ» (НОВОСИБИРСКИЙ ГОСУДАРСТВЕННЫЙ УНИВЕРСИТЕТ, НГУ)

Кафедра систем информатики
(название кафедры)

УТВЕРЖДАЮ

Зав. Кафедрой М. М. Лаврентьев

……
(подпись, дата)

ЗАДАНИЕ
НА ВЫПУСКНУЮ КВАЛИФИКАЦИОННУЮ РАБОТУ БАКАЛАВРА

Студенту: Сумбатянцу Илье Ильичу
(фамилия, имя, отчество)

Направление подготовки 230100.62 ИНФОРМАТИКА И ВЫЧИСЛИТЕЛЬНАЯ ТЕХНИКА

ФАКУЛЬТЕТ ИНФОРМАЦИОННЫХ ТЕХНОЛОГИЙ

Тема: Оптимизация производительности алгоритма избыточного кодирования данных
(полное название темы выпускной квалификационной работы бакалавра)

Исходные данные (или цель работы): оптимизация производительности алгоритма избыточного кодирования данных на основе матриц Коши над полем Галуа.

Структурные части работы: исследование прототипа системы распределенного резервного копирования на основе избыточного кодирования данных; тестирование модулей, реализующих и использующих алгоритм избыточного кодирования данных; создание реализаций алгоритма, транслирующихся в машинно-зависимый код; интеграция реализаций в прототип системы; анализ полученных результатов.

Содержание

ВВЕДЕНИЕ

В рамках проекта “Distributed storage” лаборатории НГУ Parallels был разработан прототип системы распределенного резервного копирования данных на основе алгоритма избыточного кодирования данных.

В ходе исследования прототипа было выявлено несколько существенных недостатков: ограниченный размер данных, резервную копию которых можно сделать, и низкая производительность алгоритма избыточного кодирования данных. Даже если не учитывать тот факт, что размер данных, резервную копию которых можно сделать, ограничен объемом оперативной памяти, низкая производительность приводит к трудностям при использовании программного продукта. Данные недостатки определили основную цель работы.

Целью данной работы является оптимизация алгоритма избыточного кодирования данных. После определения цели были определены задачи, которые необходимо выполнить для достижения поставленной цели:

а) Детальное изучение модуля разбиения и сборки данных, его связи с системой
б) Тестирование необходимых частей системы для возможности свободно изменять код прототипа и проверять корректность оптимизированных реализаций
в) Реализация оптимизированных нативных версий модуля
г) Интеграция нативного кода
д) Анализ полученных результатов

1 Система

1.1 Идея создания

Основная идея данной системы – использование свободного дискового пространства участников сети для резервного копирования данных. Отличительной чертой системы является тот факт, что отсутствие возможности получить от некоторых участников корректные данные (отключение от сети, модификация данных, и т. д.) не приводит систему в неработоспособное состояние.

1.2 Абстрактное описание системы

1.2.1 Архитектура системы

Систему можно рассматривать как совокупность четырех абстрактных модулей:

а) NK Algorithm
б) DHT
в) Share Manager
г) Download Manager

Далее пойдет речь о спецификациях, которым должны соответствовать эти модули.

NK Algorithm

NK Algorithm – основа всей системы. Задача этого модуля – разделение и сборка данных.

Описать его можно единственным требованием – данные должны делиться таким образом на N частей, чтобы по любым K частям можно было восстановить исходные данные. В дальнейшем такую конфигурацию я буду называть "*(N, K)*-схема".

DHT

Distributed Hash Table (Распределенная хеш таблица) – модуль, необходимый для организации сети, в которой будет происходить разбиение данных. Требования к модулю соответствуют определению интерфейса DHT:

а) Организация одноранговой сети
б) Обеспечение поискового сервиса по принципу ассоциативного массива для каждого узла сети

В прототипе системы в качестве реализации DHT используется Kademlia [8][9].

Share Manager и Download Manager

Эти менеджеры должны передавать части данных, которые были разделены, узлам сети и, соответственно, получать необходимые части из сети.

1.3 Взаимодействие между модулями

Ниже представлена диаграмма, которая показывает зависимости между модулями:

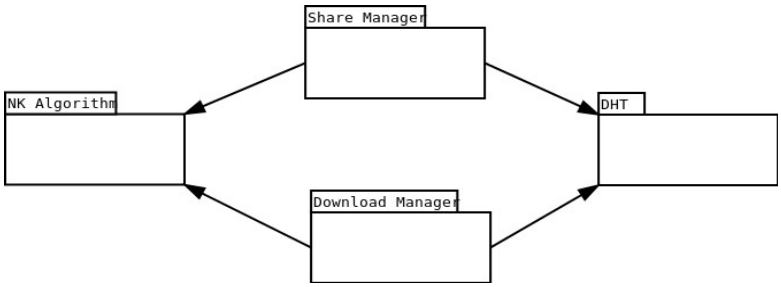

Рисунок 1: Диаграмма модулей

Модули DHT и NK Algorithm являются утилитными модулями, которые используются менеджерами для разбиения/сборки данных и раздачи/получения частей участников сети. Исходя из задач менеджеров и спецификаций DHT и NK Algorithm подробное описание Share Manager и Download Manager не требуется, потому что их реализация в большинстве своем зависит от NK Algorithm и DHT.

1.4 Поведение системы

Процесс создания резервной копии:

а) На вход подаются данные

б) Данные разбиваются в соответствии с *(N, K)*-схемой

в) Части полученные при разбиении раздаются *N-1* узлам сети

Процесс восстановления копии:

а) Любые *K-1* частей принимаются обратно из сети

б) Части собираются с помощью NK Algorithm в исходные данные

2 Модуль NK Algorithm

2.1 Реализация

Модуль NK Algorithm состоит из двух абстрактных функций, которые должны реализовывать алгоритмы разбиения и сборки данных с учетом спецификации описанной в первой главе.

Идея алгоритма состоит в следующем: берется матрица $N \times K$, у которой все подматрицы $K \times K$ невырожденные; исходное сообщение разбивается на K элементов – получается вектор длины K; вектор умножается на эту матрицу – получается вектор длины N. Это и есть избыточно закодированная форма сообщения. Взяв любые K элементов этого вектора и умножив их на обратную матрицу соответствующей подматрицы, можно получить исходное сообщение.

Формальное описание алгоритма требует введения понятий, которые определены далее.

2.2 Поле Галуа

2.2.1 Определение

Поле Галуа – это поле, состоящее из конечного числа элементов. Обозначается следующим образом: $GF(n)$, где n – число элементов поля.

Характеристика поля $char\,GF$ – это наименьшее из таких чисел c, что $c \cdot a = 0,\ \forall a \in GF(n)$.

Как и любое поле, поле Галуа обладает двумя бинарными операциями: аддитивной и мультипликативной; и образует коммутативное ассоциативное кольцо с единицей, все ненулевые элементы которого обратимы.

Обобщая все вышеизложенное, можно определить поле следующим образом:

а) $a + b = b + a,\ \forall a, b \in GF(n)$

б) $a + (b + c) = (a + b) + c,\ \forall a, b, c \in GF(n)$

в) $a \cdot b = b \cdot a,\ \forall a, b \in GF(n)\ a \neq 0\ b \neq 0$

г) $\exists 0 \in GF(n), a + 0 = 0 + a = 0,\ \forall a \in GF(n)$

д) $\exists 1 \in GF(n), a \cdot 1 = 1 \cdot a = a,\ \forall a \in GF(n)$

е) $(\forall a \in GF(n)\ \exists -a \in GF(n)),\ a + -a = -a + a = 0$

ж) $(\forall a \in GF(n),\ a \neq 0, \exists i \in GF(n)),\ a \cdot i = i \cdot a = 1$

з) $a \cdot (b + c) = a \cdot b + a \cdot c,\ \forall a, b, c \in GF(n)$

и) $(a+b)\cdot c=a\cdot c+b\cdot c,\ \forall a,b,c\in GF(n)$

к) $(a+b)\in GF(n),\ \forall a,b\in GF(n)$

л) $(a\cdot b)\in GF(n),\ \forall a,b\in GF(n)$

2.2.2 Арифметика в поле Галуа

Элементы поля Галуа $GF(p^n)$ удобно интерпретировать как полиномы порядка меньше n с коэффициентами при членах меньше p.

В данной реализации используется поле Галуа $GF(2^8)$, а это значит, что речь будет идти о полиномах следующего вида: $a_7x^7+a_6x^6+\ \ldots\ +a_1x^1+a_0,\ a_i\in\{0,\ 1\}$.

Такие полиномы достаточно просто записать в следующем виде:

$a_7a_6a_5a_4a_3a_2a_1a_0,\ a_i\in\{0,1\}$, что является битовой строкой.

Характеристикой поля $GF(2^8)$ был взят неприводимый полином

$x^8+x^4+x^3+x^2+1$

Аддитивная операция

Сложение элементов поля Галуа $GF(2^8)$ в полиномиальной интерпретации выглядит следующим образом:

$(a_7x^7+a_6x^6+\ \ldots\ +a_1x^1+a_0)+(b_7x^7+b_6x^6+\ \ldots\ +b_1x^1+b_0)=(c_7x^7+c_6x^6+\ \ldots\ +c_1x^1+c_0)$, где

$c_i=(a_i+b_i)\,mod\,2$

Что в свою очередь является “исключающим или” (xor) двух битовых строк, соответствующих данным полиномам.

Исходя из этого можно сделать вывод, что обратным элементом в поле $GF(2^n)$ является сам элемент.

Мультипликативная операция

Умножение в поле Галуа является обычным умножением двух полиномов по модулю характеристики $charGF$, где аддитивная операция определена выше.

$$(a_7x^7+a_6x^6+\ \ldots\ +a_1x^1+a_0)(b_7x^7+b_6x^6+\ \ldots\ +b_1x^1+b_0)=$$
$$=(c_{14}x^{14}+c_{13}x^{13}+\ \ldots\ +c_1x^1+c_0)\,mod\,(charGF)$$

Есть несколько вариантов реализации умножения в поле Галуа: непосредственно умножать или вычислять таблицу умножения заранее. Таблица может быть полезной, если умножение происходит в поле небольшой размерности.

Исходный код умножения двух элементов поля и вычисления таблицы умножения приведен в Приложении А.

Обратный элемент

В общем случае, обращение элемента поля представляет из себя перебор элементов до тех пор, пока не будет найден такой элемент, который при умножении на исходный даст единицу. Тем не менее можно сконструировать таблицу обратных элементов по ходу вычисления таблицы умножения.

Исходный код вычисления таблицы обратных элементов приведен в Приложении А.

2.3 Матрица Коши

2.3.1 Определение

Матрица Коши – это матрица размером $n \times k$, элементы которой имеют следующий вид: $a_{ij} = \frac{1}{x_i + y_i}$, где $x_1, \ldots, x_n, y_1, \ldots, y_k$ – элементы некоторого поля F. Тогда в общем случае матрица Коши будет выглядеть следующим образом:

$$C = \begin{pmatrix} \frac{1}{x_1 + y_1} & \frac{1}{x_1 + y_2} & \cdots & \frac{1}{x_1 + y_k} \\ \frac{1}{x_2 + y_1} & \frac{1}{x_2 + y_2} & \cdots & \frac{1}{x_2 + y_k} \\ \vdots & \vdots & \ddots & \vdots \\ \frac{1}{x_n + y_1} & \frac{1}{x_n + y_2} & \cdots & \frac{1}{x_n + y_k} \end{pmatrix}$$

2.3.2 Детерминант Коши

Известно, что детерминант квадратной матрицы Коши можно записать следующим образом:

$$det(C) = \frac{\prod_{1 \le i < j \le n} (x_j - x_i)(y_j - y_i)}{\prod_{1 \le i, j \le n} (x_i + y_j)}$$

Исходя из такого определения детерминанта, можно наложить несколько ограничений на элементы, порождающие матрицу Коши. Для того, чтобы определитель матрицы был определен и не равен нулю, необходимо следующее:

а) Элементы $x_1, \ldots, x_n$ должны быть различными

б) Элементы $y_1, \ldots, y_k$ должны быть различными

Если элементы являются элементами поля Галуа, определенного выше, то должно выполняться еще одно условие, которое исключает неопределенность детерминанта: все элементы $x_1, \ldots, x_n, y_1, \ldots, y_k$ должны быть попарно различными. Это условие исключает ситуацию, когда $(x_i + y_j)$ равняется нулю.

Отсюда вытекает важное свойство матриц Коши – если матрица размером $n \times k$ была сгенерирована из порождающих элементов, которые удовлетворяют условиям, описанным выше, то любая квадратная подматрица этой матрицы будет невырожденной.

2.3.3 Генерация матрицы Коши

В дальнейшем будет идти речь о матрицах Коши над полем Галуа $GF(2^8)$. То есть поле состоит их 256 элементов, что накладывает ограничение на размер матриц: $(n + k) \leq 256$.

Для того, чтобы сгенерировать матрицу Коши размера $n \times k$, необходимо взять 2 набора попарно различных элементов $x_1, \ldots, x_n, y_1, \ldots, y_k$, после чего по определению вычислить элементы матрицы Коши.

2.4 Алгоритм разбиения данных

Все дальнейшие действия будут происходить в поле Галуа $GF(2^8)$, в том числе и все матрицы Коши будут над этим же полем.

Для работы алгоритму необходимы данные и схема разбиения (n, k).

Данные разбиваются на блоки длиной k. Генерируется матрица Коши C размера $n \times k$. Для каждого блока будет выполняться следующая операция: блок интерпретируется как вектор inv и умножается на матрицу C, и результатом будет вектор $outv$ размером n, который можно интерпретировать как набор элементов, каждый i-ый из которых необходимо положить в i-ое хранилище части (в качестве хранилища может использоваться файл или массив). Если последний блок не кратен k, то он дополняется нулями.

Пусть исходные данные имеют размер $dataSize$, тогда размер всех частей будет равен $n \cdot [dataSize / k]$.

Наглядно это можно записать следующим образом:

$$data \rightarrow \begin{pmatrix} inv_1 \\ inv_2 \\ \vdots \\ inv_k \end{pmatrix}_1 \dots \begin{pmatrix} inv_1 \\ inv_2 \\ \vdots \\ inv_k \end{pmatrix}_{[dataSize/k]}$$ – разбиение входных данных на части

$$\begin{pmatrix} \frac{1}{x_1+y_1} & \frac{1}{x_1+y_2} & \dots & \frac{1}{x_1+y_k} \\ \frac{1}{x_2+y_1} & \frac{1}{x_2+y_2} & \dots & \frac{1}{x_2+y_k} \\ \vdots & \vdots & \ddots & \vdots \\ \frac{1}{x_n+y_1} & \frac{1}{x_n+y_2} & \dots & \frac{1}{x_n+y_k} \end{pmatrix} \cdot \begin{pmatrix} inv_1 \\ inv_2 \\ \vdots \\ inv_k \end{pmatrix}_i = \begin{pmatrix} outv_1 \\ outv_2 \\ outv_3 \\ \vdots \\ outv_n \end{pmatrix}_i, \; 1 \le i \le [dataSize/k]$$ – вычисление данных частей

$$\begin{pmatrix} outv_1 \\ outv_2 \\ outv_3 \\ \vdots \\ outv_n \end{pmatrix}_i \begin{matrix} \rightarrow part_1 \\ \rightarrow part_2 \\ \rightarrow part_3 \\ \vdots \\ \rightarrow part_n \end{matrix}$$ – перемещение полученных данных в хранилища частей

2.5 Алгоритм сборки данных

Алгоритму необходимы k частей, полученных при разбиении, и соответствующие строки матрицы Коши. Квадратная подматрица размером $k \times k$ получается сразу после сборки строк в матрицу, после чего находится обратная матрица C (свойство о невырожденности квадратных подматриц матрицы Коши гарантирует существование матрицы C). Составим вектор длины k, выбирая по одному очередному элементу из *i*-ой части и помещая на *i*-ое место в вектор. Умножаем C на полученный вектор – результатом будет очередной блок исходного файла. Объединяя полученные таким образом блоки, получим исходные данные.

2.6 Определение ложных данных

Пусть есть N частей, которые были получены в результате разбиения данных. Если части не были изменены после разбиения, то сборка по любым K частям будет в результате давать одни и те же данные.

Чтобы определить, что среди $K+1$ частей есть хотя бы одна модифицированная, достаточно произвести две сборки с различными наборами из K частей – результаты не будут совпадать.

После этого можно провести K сборок, по очереди исключая одну из частей. Если

при этом получатся K одинаковых результатов и один отличающийся, то можно однозначно определить единственную часть с искаженными данными. Иначе можно сделать вывод, что среди $K+1$ частей присутствует больше одной искаженной части.

3 Оптимизация модуля NK Algorithm

Алгоритм избыточного кодирования данных был реализован, как и вся система, на языке Java. На вычислительных задачах Java показывает достаточно низкую производительность. Поэтому было решено сделать нативную реализацию алгоритма и оптимизировать её, используя аппаратные средства. Это решение определило языки и технологии, которые будут использоваться для оптимизации.

Языками, которые можно использовать для достижения цели, были выбраны C и C++.

Библиотеки и технологии, которые подходят для данной цели и позволяют использовать аппаратные средства:

а) SSE – потоковое SIMD-расширение процессора
б) OpenCL – фреймворк, для написания параллельных программ, исполняемых на графических или центральных процессорах.
в) Boost ublas – библиотека из набора C++ библиотек boost, реализующая интерфейс BLAS (Basic Linear Algebra Subroutine).

В качестве инструментов для компиляции, отладки и профилирования были выбраны:

а) GNU Compiler Collection (GCC). Из данного набора компиляторов использовались gcc (компилятор языка C) и g++ (компилятор языка C++).
б) GNU debugger (gdb) – отладчик
в) Valgrind – набор инструментов для отладки и профлирования. В данном случае использовались: memcheck (инструмент для анализа работы с памятью) и cachegrind (инструмент для профилирования кеша)
г) GNU profiler (gprof) – утилита для профилирования. Позволяет выяснить, сколько времени исполняется каждая функция, а так же построить граф вызовов (call graph).

Для интеграции нативного кода в систему был выбран стандартный интерфейс JNI (Java Native Interface).

3.1 Анализ прототипа

После постановки цели работы, возникла необходимость в детальном изучении модуля NK Algorithm, его взаимодействии с другими частями системы. Результат исследования системы, а так же данного модуля был представлен в главах 1 и 2.

Изучение модуля привело к пониманию того, в каком направлении надо двигаться при написании нативного кода и какие проблемы можно решить на этапе разработки.

3.2 Тестирование

После анализа системы были протестированы модуль NK Algorithm и весь код, который зависит от модуля NK Algorithm. Тестирование необходимо для двух задач: рефакторинга кода (для повышения читаемости, простоты изменения и устранения ограничения размера входных данных) и проверки интегрированного нативного кода.

Так же был написан набор тестов, необходимый для проверки правильности работы нативных реализаций. Этот набор сравнивает результат исполнения реализации алгоритма исходной системы и результат исполнения нативной или альтернативной реализации в самой системе.

В качестве инструмента для тестирования была выбрана библиотека Junit.

В итоге было покрыто около 30 процентов всего кода системы, что включает в себя все необходимые модули.

3.3 Измерение производительности

Пусть есть *(n, k)*-схема, в соответствии с которой необходимо разбить файл длиной *filesize*. Тогда размер выходных данных равен $n \cdot \frac{filesize}{k}$. В случае со сборкой данных речь идет о схемах вида *(k, k)*, что определяет размер аналогичным образом.

Тогда производительность можно оценивать следующим образом: $\left(n \cdot \frac{filesize}{k} \right) / time$. Иными словами – это размер продуцируемых данных в секунду.

С другой стороны, можно измерять количество входных обработанных данных в секунду, но такая информация не будет отражать затраты на получение одного элемента продукции. То есть, будет видно, что данные обрабатываются быстрее с ростом k, но скорость получения продукции будет уменьшаться. По этой причине я остановился на измерении размера продуцируемых данных в единицу времени.

Исходя из поставленной проблемы (низкая производительность алгоритма при работе системы), необходимо учитывать передачу данных через JNI (в случае с измерением производительности нативного кода), загрузку данных и запись данных.

Исходя из вышеизложенного, дополнительно была написана программа на Java, которая измеряет время работы реализаций алгоритма и учитывает все необходимые

затраты на работу алгоритма.

Для получения более точных результатов проводится измерение времени трех проходов алгоритма и делится на три, а в случае с реализациями на Java учитываются так же оптимизации JIT-компилятора, для этого, дополнительно перед измерением времени, делается три холостых прогона необходимого участка кода.

Все измерения были произведены на компьютере с процессором Intel Core 2 Quad (4 ядра, 2336 МГц, L1-кеш: 32 Кб, L2-кеш: 2048 Кб) и видеокартой NVIDIA GeForce GTX 650Ti (768 ядер, 928 МГц).

3.4 Рефакторинг

В первую очередь, подвергся рефакторингу модуль NK Algorithm, так же частично были изменены модули, зависящие от него. В ходе рефакторинга был отделен код, который необходимо менять, и производительность которого необходимо измерить, что упростило будущий процесс интеграции.

В ходе рефакторинга был создан единый интерфейс алгоритма избыточного кодирования, который не фиксирует процесс работы с данными (то есть чтение и запись данных скрыта в реализации интерфейса). Данный интерфейс должен быть реализован как в нативном коде, так и в исходной системе.

Реализация этого интерфейса в системе позволила устранить ограничение размера файла, что повлекло за собой изменение семантики методов, реализующих алгоритмы разбиения и сборки, что в свою очередь повлекло изменения в модулях, которые использовали эти методы.

После тестирования было измерено время разбиения и сборки исходной и измененной реализации. Так как исходная реализация может обрабатывать ограниченный объем данных, то измерение времени было произведено на данных размером 1 мегабайт.

Исходная реализация демонстрировала скорость получения продукции от 1 до 26 МБ/с. Измененная же реализация демонстрирует такую же скорость, и разница времени их работы колеблется в рамках погрешности. Поэтому в дальнейших сравнениях будет использоваться измененная реализация.

3.5 Интеграция

Набор C++ библиотек Boost предлагает высокоуровневую библиотеку ublas для манипуляций с матрицами, поэтому он удобен для быстрой реализации алгоритма, но по производительности не оптимален. Для отладки механизмов интеграции нативного кода с

Java, первой была выбрана реализация на boost.

3.6 Оптимизация

После проделанной работы, процесс создания и оптимизации нативных реализаций стал выглядеть следующим образом:

а) Создание отлаженной нативной реализации
б) Интеграция в исходную систему
в) Тестирование
г) Измерение времени исполнения в системе
д) Оптимизация (изменение реализации)

Шаги в, г, д образуют цикл, который должен проводиться после каждого законченного изменения реализации.

3.6.1 Реализация на C

Основой, необходимой для дальнейших оптимизаций стала реализация арифметики над полем Галуа и алгоритм избыточного кодирования, определенные в разделе 2.2, на языке C. Для сравнения были реализованы оба варианта умножения: чистое умножение без вспомогательной таблицы, и умножение с использованием заранее вычисленной таблицы.

Разбиение данных

Чтобы не повторять ошибку исходного прототипа (, количество выделенной памяти контролируется объемом части, которая читается из файла на каждой итерации. Данный прием описан в приложении Б.

Входные данные не требуют трансформации, так как уже представляют из себя набор векторов размера k.

Выходные данные же требуют изменения: каждый выходной вектор размера n содержит в себе по одному элементу соответствующих частей. Имея r таких векторов на выходе, достаточно интерпретировать их как матрицу и транспонировать. Таким образом получится матрица размером $n \times r$, строки которой содержат по r элементов каждой части.

Сборка данных

Выделенная память контролируется аналогичный образом, как и при разбиении данных.

Ситуация с входными и выходными данными становится обратной: то есть

выходные данные – набор векторов, который необходимо записать в файл; входные данные (так же набор векторов) можно представить в виде матрицы $k \times r$, в таком случае можно читать данные из частей блоками длины r, но векторы не будут расположены в памяти линейно. Поэтому, как и в случае с выходными данными при разбиении, достаточно транспонировать матрицу, состоящую из входных векторов.

Анализ

Утилитой gprof было проверено, что большая часть времени исполнения тратится на работу алгоритма (без чтения и записи данных). В случае с умножением по таблице также было проверено количество кеш-промахов, которое составило 1-2 процента.

Реализация алгоритма с умножением без заранее посчитанной таблицы демонстрирует низкую производительность, но она была необходима для дальнейшей векторизации и сравнения с векторизованной реализацией.

Реализация алгоритма, использующая заранее посчитанную таблицу умножения, работает в 2.5-3.7 раз быстрее реализации на Java.

Ниже представлены результаты работы реализации. На *(n, k)*-схемах с *n* равным 8 данная реализация работает в 3.5-3.7 раз быстрее. А при *n* равным 120-128 в 2.5-2.6 раз быстрее.

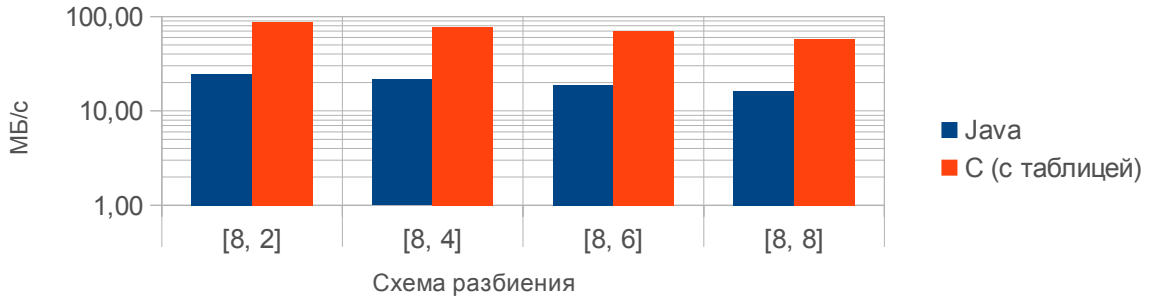

Рисунок 2: Лучший результат работы реализации на C

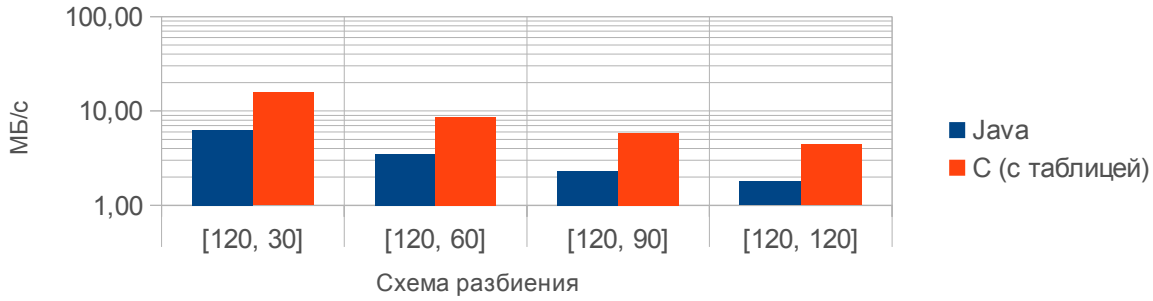

Рисунок 3: Худший результат работы реализации на C

Общие результаты представлены в приложении В.

3.6.2 Реализация на языке С с использованием SSE

Современные ЦПУ содержат набор SSE инструкций, которые работают со скалярными и упакованными данными, размещенными на 128-битных регистрах. Библиотечные типы данных могут отображаться на эти регистры и интерпретироваться как массив чисел меньшего размера. Например, как массив из шестнадцати восьмибитных чисел.

В определении реализации алгоритмов разбиения и сборки используется поле Галуа $GF(2^8)$, то есть все элементы поля умещаются в один байт, а значит в одной такой 128-битной переменной можно уместить 16 элементов поля Галуа. Таким образом алгоритм можно оптимизировать путем векторизации, умножая и складывая сразу по 16 элементов.

Векторизация алгоритма избыточного кодирования была проведена в два этапа.

Первым было векторизовано сложение. Умножение производилось по таблице, поэтому результат умножения матрицы на вектор вычисляется не за один проход, а за два прохода: на первом проходе вычисляется матрица, в которой все элементы просто умножаются на элементы вектора (то есть остается только сложить элементы в каждой строке и получить результирующий вектор). После чего полученная матрица транспонируется и делится следующим образом: каждая строка делится на блоки по 16 элементов (если число столбцов в матрице не кратно 16, то необходимо расширить матрицу, то есть сделать количество столбцов в транспонированной матрице кратным 16 или заранее расширить исходную матрицу, добавив такое число строк, что их число будет кратным 16), последние элементы разбиения содержат данные, которые не следует учитывать после умножения.

Основной причиной расширения матрицы стала необходимость в выровненных данных для загрузки в 128-битные регистры. По той же причине необходимо расширение контейнера для части выходных данных.

Второй этап – векторизация умножения. Умножение производилось следующим образом: умножение блока размером 16 элементов поля Галуа на один элемент. Такая реализация определяет умножение матрицы на вектор входных данных – каждый столбец матрицы умножается на соответствующий элемент из вектора и складывается с результирующим вектором. Это определяет отображение матрицы на память – столбцы матрицы должны располагаться линейно. Стоит заметить, что матрица должна быть

расширена, как и контейнер, по тем же соображениям, что и на первом этапе.

На небольших схемах разбиения данная реализация работает медленнее исходной реализации на C с заранее посчитанной таблицей умножения, но, тем не менее, работает быстрее реализации на C без таблицы умножения в 2-9 раз, в зависимости от схемы разбиения. Это объясняется тем, что в реализации используются расширенные матрица и выходные данные.

На *(n, k)*-схемах, где *n* превышает 60 данная реализация демонстрирует производительность до в 1.5 раза превышающую производительность реализации на C, и, как и ожидалось, она производительнее в 10-15 раз реализации на C без таблицы умножения.

В сравнении с реализацией на Java данная реализация производительнее в 1.5-4 раза.

На представленных ниже гистограммах можно видеть худший и лучший результаты работы векторизованной реализации.

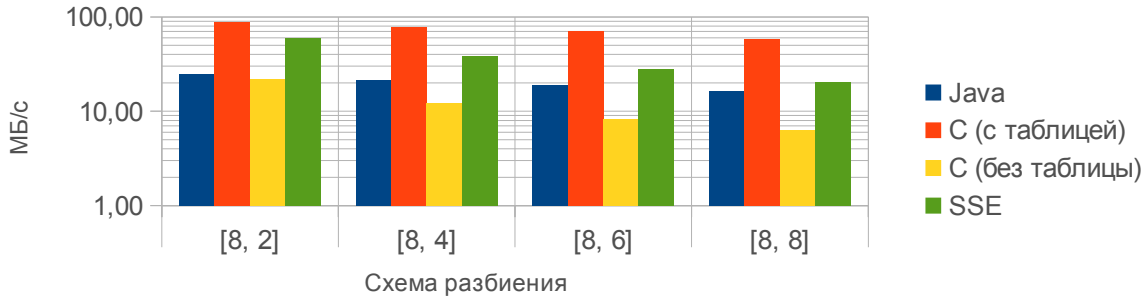

Рисунок 4: Худший результат работы векторизованной реализации (SSE)

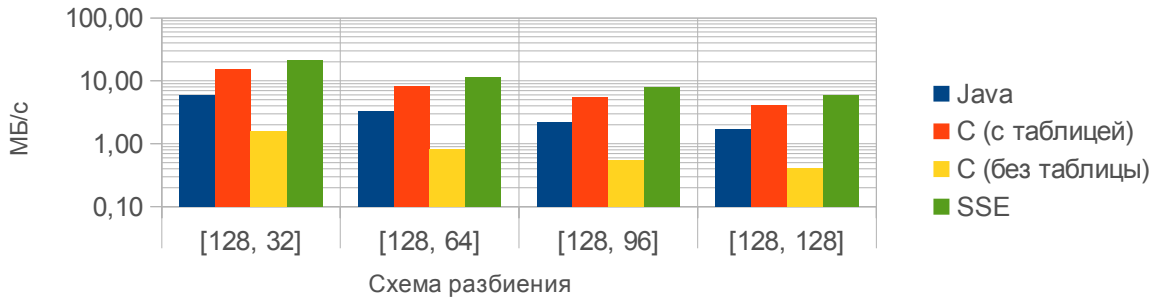

Рисунок 5: Лучший результат работы векторизованной реализации (SSE)

Общие результаты представлены в приложении B.

3.6.3 Реализация на C с использованием OpenCL

Использование графической карты для работы алгоритма можно обосновать

следующими соображениями:

- Вычисления на видеоадаптере позволяют снизить нагрузку на процессор, что в свою очередь позволит производить резервное копирование в любое время
- Вычисления на видеоадаптере могут дать значительный прирост производительности в силу своей архитектуры.

OpenCL – кроссплатформенная технология, реализацию которой поддерживают большинство вендоров.

В ходе оптимизации было создано множество реализаций, и лучшей из них стала реализация блочного умножения, о которой далее пойдет речь.

OpenCL позволяет интерпретировать множество вычислителей видеоадаптера как множество 1-, 2- или 3-мерных групп вычислителей. В данном случае удобно интерпретировать видеоадаптер как множество 2-мерных групп вычислителей, а все данные (как входные, так и выходные) как матрицы. Каждая группа вычислителей считает блок результирующей матрицы. Размеры матриц не всегда кратны размеру блока, поэтому матрицы необходимо расширять до размеров, кратных размеру блока, а ненужную продукцию отбрасывать при записи, как это делалось при векторизации алгоритма.

На небольших схемах разбиения размер продукции, которая будет отброшена, может значительно превосходить реальный размер данных, что делает данную реализацию недостаточно производительной. Тем не менее реализация показывает хорошие результаты на больших схемах.

В сравнении с реализацией на Java реализация на OpenCL производительнее в 3-10 раз на небольших схемах разбиения ($n = 8 \ldots 30$), на больших схемах OpenCL-реализация производительнее в 8-18 раз, в зависимости от схемы разбиения.

На двух представленных гистограммах можно видеть минимум и максимум увеличения производительности OpenCL-реализации относительно Java-реализации.

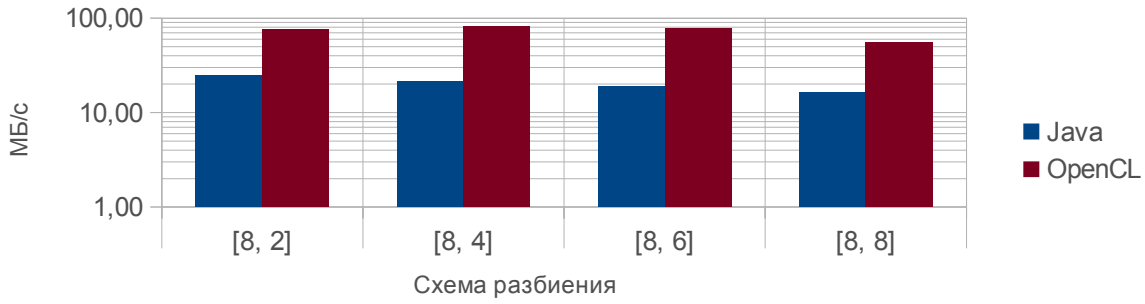

Рисунок 6: Худший результат работы OpenCL-реализации

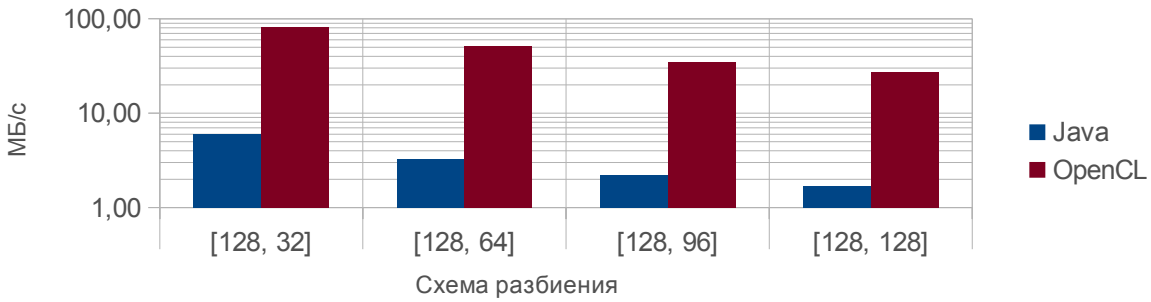

Рисунок 7: Лучший результат работы OpenCL-реализации

Общие результаты представлены в Приложении В.

ЗАКЛЮЧЕНИЕ

В ходе работы был исследован прототип системы распределенного резервного копирования, что повлекло за собой изучение сопутствующей теоретической базы. В ходе исследования были выявлены два недостатка: ограниченный размер данных, резервную копию которых можно сделать, и недостаточная производительность алгоритмов разбиения и сборки данных.

Первый недостаток был исправлен в коде прототипа, при этом необходимые модули были покрыты тестами, а так же был произведен рефакторинг модуля NK Algorithm.

Для исправления второго недостатка были написаны оптимизированные нативные реализации алгоритма избыточного кодирования данных. Все они были интегрированы в исходный прототип и протестированы.

Для измерения производительности была написана программа на Java для измерения времени работы реализаций алгоритма с учетом всех необходимых затрат на исполнение алгоритма, а так же с учетом оптимизаций Java JIT-компилятора.

В сравнении с исходной Java-реализацией нативные реализации алгоритма избыточного кодирования, исполняющиеся на процессоре, работают в 2.5-4 раза быстрее, а в случае с исполнением на видеоадаптере производительность была увеличена в 3-18 раз, в зависимости от схемы разбиения.

В качестве дальнейшего развития данной работы можно предложить несколько вариантов, реализации которых не исключают друг друга:

- Параллельные реализации на основе представленного нативного кода.
- Комбинированная реализация, которая будет использовать одновременно ресурсы процессора и видеокарты.
- Реализация, которая будет делить задачу разбиения между доверенными участниками сети, то есть между ними будет построена вычислительная сеть. Подобного эффекта можно добиться, например, средствами MPI.

Результаты работы были представлены и опубликованы на Международной научной студенческой конференции «Студент и научно-технический прогресс» в 2013 году [10].

Литература

1. Крис Касперский, Полиномиальная арифметика и поля Галуа или информация, воскресшая из пепла II // Системный администратор, - 2003 - №10 - С. 84-90.
2. Кнут, Дональд Э. Искусство программирования, том 1. Основные алгоритмы, 3-е изд. : Пер. с англ. : Уч. пос. – М : Издательский дом “Вильямс”, 2012 – 720 с.
3. Boost Basic Linear Algebra – 1.53.0 [Электронный ресурс], режим доступа: http://www.boost.org/doc/libs/1_53_0/libs/numeric/ublas/doc/index.htm, свободный.
4. OpenCL 1.1 Reference Pages. [Электронный ресурс]. - Режим доступа: http://www.khronos.org/registry/cl/sdk/1.1/docs/man/xhtml/, свободный.
5. OpenCL Programming Guide for the CUDA Architecture [Электронный ресурс], -2009, - 61 с. - Режим доступа: http://www.nvidia.com/content/cudazone/download/OpenCL/NVIDIA_OpenCL_ProgrammingGuide.pdf, свободный.
6. NVIDIA OpenCL Best Practices Guide. [Электронный ресурс], - 2009, - 49 с. - Режим доступа: http://www.nvidia.com/content/cudazone/CUDABrowser/downloads/papers/NVIDIA_OpenCL_BestPracticesGuide.pdf, свободный.
7. James S. Plank, Screaming Fast Galois Field Arithmetic Using Intel SIMD Instructions. Knoxville: University of Tennessee, [Электронный ресурс]. - 2013, - 8 с. - Режим доступа: https://www.usenix.org/system/files/conference/fast13/fast13-final76.pdf, свободный.
8. Чеботарев С. Е., Обеспечение защиты и целостности информации в пиринговых сетях: Выпускная квалификационная работа бакалавра / НГУ, - Новосибирск, 2006, - 23 с.
9. Барыкина И. В., Система резервного копирования на основе децентрализованного распределенного избыточного хранилища данных: Выпускная квалификационная работа бакалавра / НГУ, - Новосибирск, 2009, НГУ, - 26 с.
10. Сумбатянц И. И., Оптимизация алгоритмов разбиения и сборки системы распределенного резервного копирования на основе избыточного кодирования данных // Материалы 51-й международной научной студенческой конференции «Студент и научно-технический прогресс». Информационные технологии. Новосибирск: изд-во НГУ, - 2013. - Апрель. - с. 163.

Приложение А

Исходный код программы вычисления таблиц умножения и обратных элементов.

```
int* multTable = new int[SIZE * SIZE];
int* invTable = new int[SIZE];

memset(multTable, 0, SIZE * SIZE * sizeof(int));
memset(invTable, 0, SIZE * sizeof(int));
memset(sqrtTable, 0, SIZE * sizeof(int));

for (int i = 0; i < SIZE; i++) {
    for (int j = 0; j < SIZE; j++) {
        if (i % 2 == 0) {
            multTable[i * SIZE + j] = multTable[(i >> 1) * SIZE + j] << 1;
            if (multTable[i * SIZE + j] > 255)
                multTable[i * SIZE + j] ^= 0x11D;
        }
        else
            multTable[i * SIZE + j] = multTable[(i - 1) * SIZE + j] ^ j;

        if (multTable[i * SIZE + j] == 1)
            invTable[i] = j;
    }
}
```

Исходный код умножения двух элементов поля Галуа ($result = a \cdot b$).

```
uchar result = 0;
uchar carry;
for (int counter = 0; counter < 8; counter++) {
      if (b & 1)
            result ^= a;
      carry = (a & 0x80);
      a <<= 1;
      if (carry)
            a ^= 0x1D;
      b >>= 1;
}
```

Приложение Б

Контроль потребляемой памяти при исполнении нативного кода NK Algorithm.

В первую очередь определим некоторые переменные:

1. $partSizeLimit$ – ограничение размера данных, которые будут читаться за один раз.
2. $partsNumber = partSizeLimit / k$ – количество частей размера k, на которое можно разбить $partSizeLimit$
3. $invSize = k \cdot partsNumber$ – фактический размер данных, который будет прочитан за один раз.
4. $outvSize = n \cdot partsNumber$ – размер данных, который получится при разбиении $invSize$ входных данных
5. $allocSize = invSize + outvSize$ – общий размер выделяемой памяти под входные и выходные данные

Разбиение и сборка данных – это единственное место, в котором может происходить бесконтрольное выделение памяти. Для того, чтобы зафиксировать размер входных и выходных данных достаточно взять $partSizeLimit$ равным $k \cdot allocSize / (n + k)$, где $allocSize$ уже будет фиксированным.

Приложение В

Таблица 1 содержит результаты измерения производительности реализаций, представленных в данной работе. Все результаты – размер выходных данных (в мегабайтах) в секунду.

Схема разбиения	Java	C (с таблицей)	C (без таблицы)	SSE	OpenCL
[8, 2]	24.55	87.55	21.79	59.70	76.25
[8, 4]	21.58	77.47	12.11	38.20	80.78
[8, 6]	18.67	70.05	8.26	27.77	78.33
[8, 8]	16.25	57.69	6.28	20.29	55.80
[16, 4]	21.15	66.56	11.92	69.33	121.78
[16, 8]	16.12	55.22	6.23	39.96	136.48
[16, 12]	12.44	40.92	4.21	28.59	138.12
[16, 16]	10.37	31.18	3.19	21.11	77.90
[24, 6]	18.45	56.78	8.25	43.76	102.79
[24, 12]	12.45	38.58	4.20	24.67	115.58
[24, 18]	9.38	27.12	2.84	17.08	68.04
[24, 24]	7.46	20.61	2.15	12.64	49.86
[32, 8]	15.72	49.94	6.26	44.04	124.43
[32, 16]	10.16	29.18	3.19	24.50	145.07
[32, 24]	7.53	20.42	2.14	17.14	86.72
[32, 32]	5.94	15.57	1.61	10.17	49.53
[40, 10]	13.75	39.11	5.02	35.30	109.84
[40, 20]	8.67	23.37	2.57	19.68	77.23
[40, 30]	6.28	16.39	1.72	13.51	72.74
[40, 40]	4.98	12.55	1.29	5.12	37.69
[48, 12]	12.29	32.55	4.20	35.29	125.46
[48, 24]	7.54	19.99	2.14	19.29	90.30
[48, 36]	5.40	13.89	1.44	13.47	63.27
[48, 48]	4.25	10.64	1.08	7.90	34.65
[56, 14]	11.10	30.11	3.59	30.46	111.92
[56, 28]	6.69	17.19	1.84	17.06	80.71
[56, 42]	4.77	12.04	1.23	11.75	56.30
[56, 56]	3.72	9.22	0.93	7.17	32.34
[64, 16]	10.15	25.37	3.16	30.86	124.33
[64, 32]	6.00	15.28	1.61	16.96	90.37
[64, 48]	4.26	10.62	1.08	11.78	63.24
[64, 64]	3.30	8.11	0.81	6.61	46.87
[72, 18]	9.40	24.86	2.84	27.84	77.73
[72, 36]	5.46	13.69	1.43	14.96	59.98
[72, 54]	3.81	9.51	0.96	10.50	45.33
[72, 72]	2.83	7.21	0.73	5.56	24.53
[80, 20]	8.51	22.54	2.56	27.69	83.11
[80, 40]	4.95	12.54	1.29	15.27	65.75
[80, 60]	3.47	8.61	0.87	10.52	49.81
[80, 80]	2.58	6.58	0.65	3.73	39.82
[88, 22]	8.02	20.93	2.32	24.82	79.42

Схема разбиения	Java	C (с таблицей)	C (без таблицы)	SSE	OpenCL
[88, 44]	4.57	11.45	1.18	13.74	60.91
[88, 66]	2.98	7.87	0.79	9.44	37.91
[88, 88]	2.30	5.99	0.59	3.69	31.37
[96, 24]	7.22	19.42	2.14	25.27	88.13
[96, 48]	4.07	10.61	1.08	13.78	64.66
[96, 72]	2.71	7.12	0.73	9.47	41.02
[96, 96]	2.09	5.52	0.54	4.79	34.31
[104, 26]	6.79	18.29	1.97	22.94	79.24
[104, 52]	3.78	9.85	0.99	12.58	47.03
[104, 78]	2.52	6.73	0.66	8.64	38.37
[104, 104]	1.93	5.10	0.50	3.94	27.98
[112, 28]	6.70	17.09	1.83	22.80	85.28
[112, 56]	3.72	9.18	0.93	12.49	49.91
[112, 84]	2.48	6.28	0.61	8.63	34.94
[112, 112]	1.91	4.75	0.46	4.94	30.03
[120, 30]	6.33	15.70	1.74	21.39	79.14
[120, 60]	3.50	8.62	0.87	11.44	48.68
[120, 90]	2.31	5.87	0.57	7.88	32.90
[120, 120]	1.79	4.43	0.43	3.75	25.08
[128, 32]	5.98	15.25	1.60	21.11	81.40
[128, 64]	3.28	8.12	0.81	11.52	51.64
[128, 96]	2.18	5.50	0.54	7.85	34.94
[128, 128]	1.68	4.17	0.42	5.92	27.17

Таблица 1. Результаты измерения производительности реализаций

Результаты также были представлены в виде гистограмм. Для наглядности данные были разбиты по числам n в представленных (n, k)-схемах, а ось OY представлена в логарифмическом масштабе.

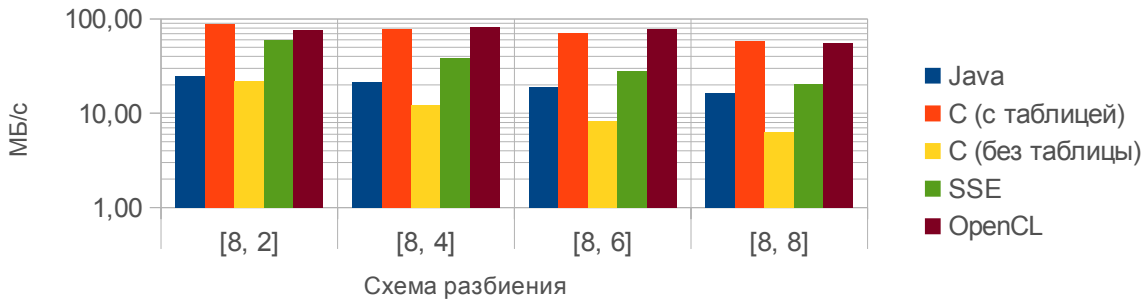

Рисунок 8. Гистограмма с результатами реализаций на (8, k)-схемах

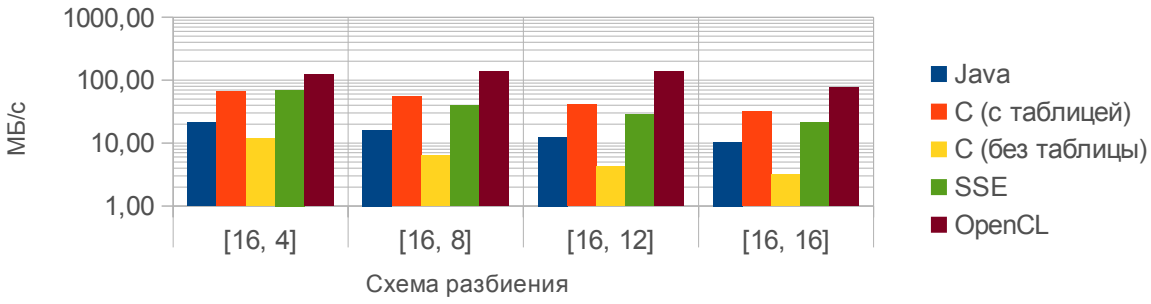

Рисунок 9. Гистограмма с результатами реализаций на (16, k)-схемах

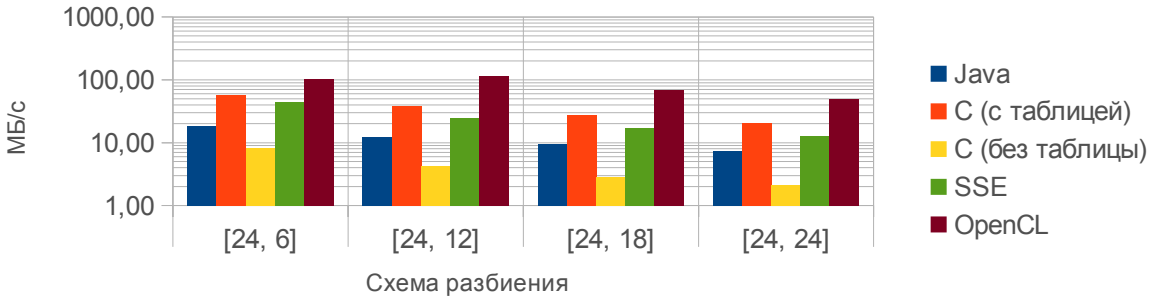

Рисунок 10. Гистограмма с результатами реализаций на (24, k)-схемах

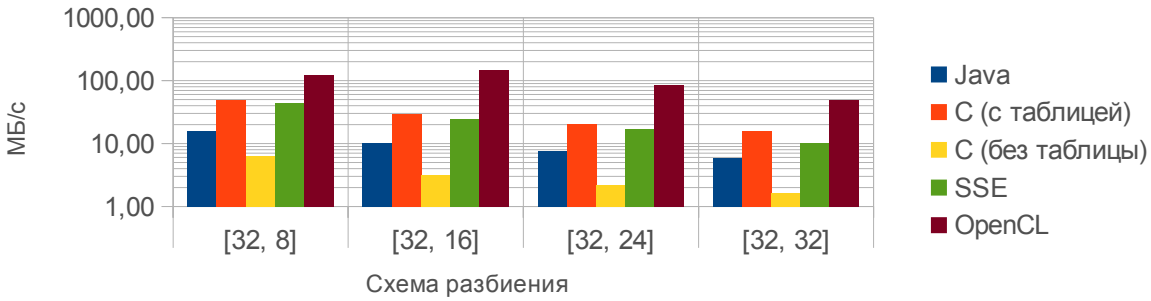

Рисунок 11. Гистограмма с результатами реализаций на (32, k)-схемах

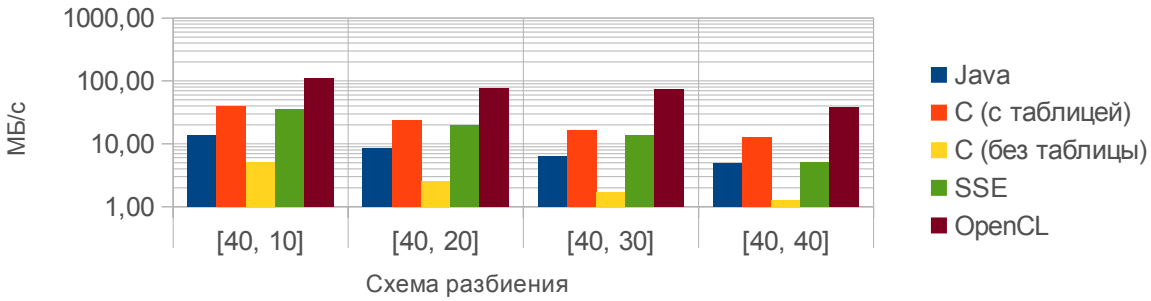

Рисунок 12. Гистограмма с результатами реализаций на (40, k)-схемах

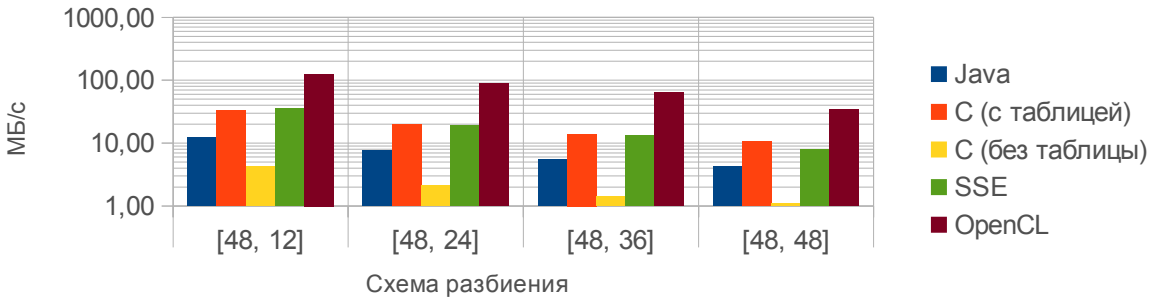

Рисунок 13. Гистограмма с результатами реализаций на (48, k)-схемах

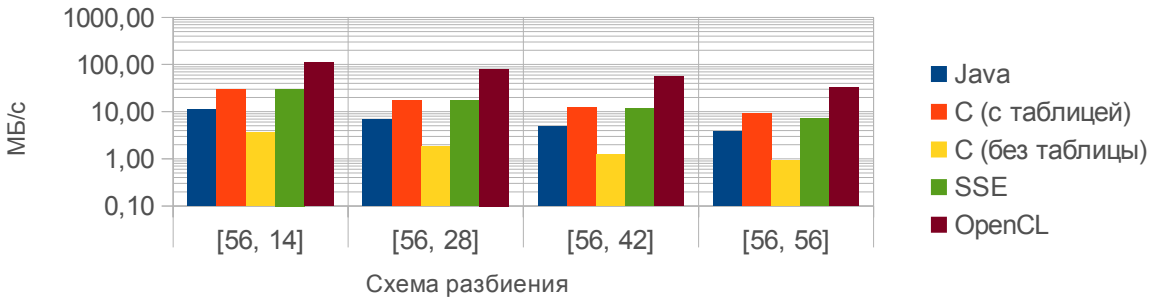

Рисунок 14. Гистограмма с результатами реализаций на (54, k)-схемах

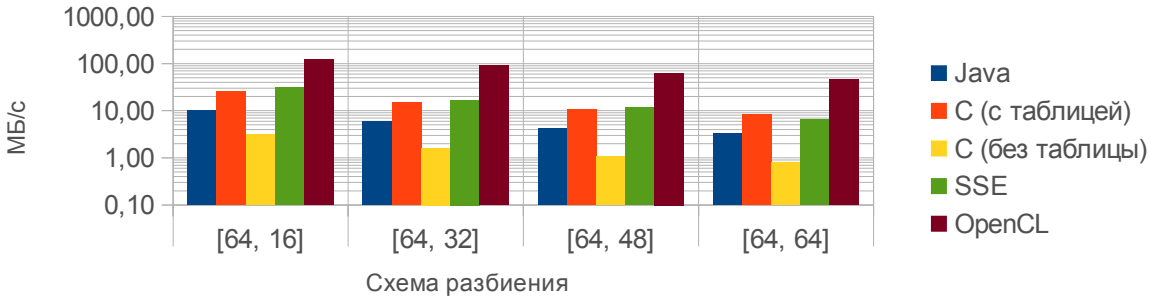

Рисунок 15. Гистограмма с результатами реализаций на (64, k)-схемах

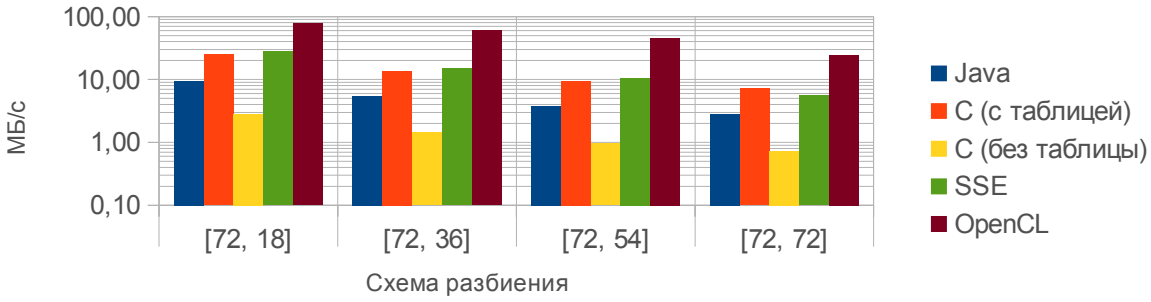

Рисунок 16. Гистограмма с результатами реализаций на (72, k)-схемах

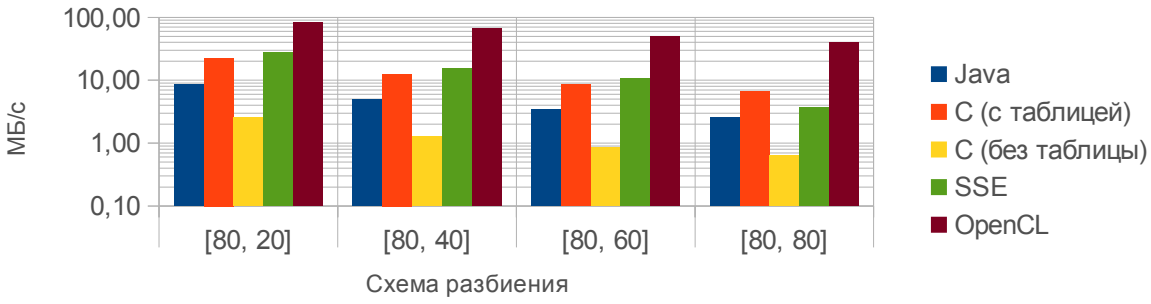

Рисунок 17. Гистограмма с результатами реализаций на (80, k)-схемах

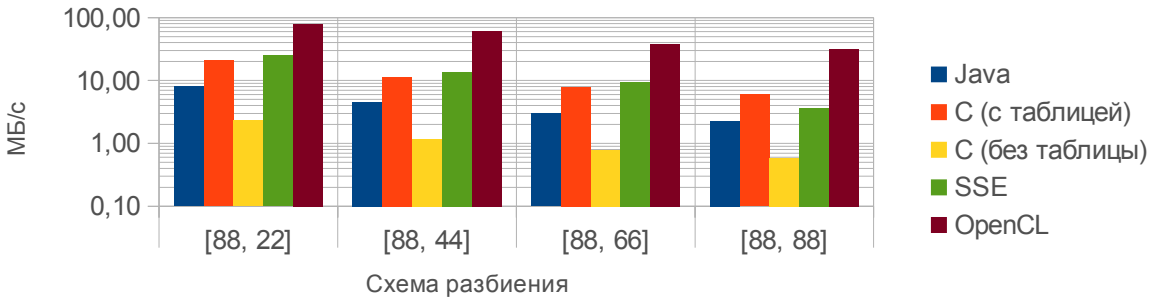

Рисунок 18. Гистограмма с результатами реализаций на (88, k)-схемах

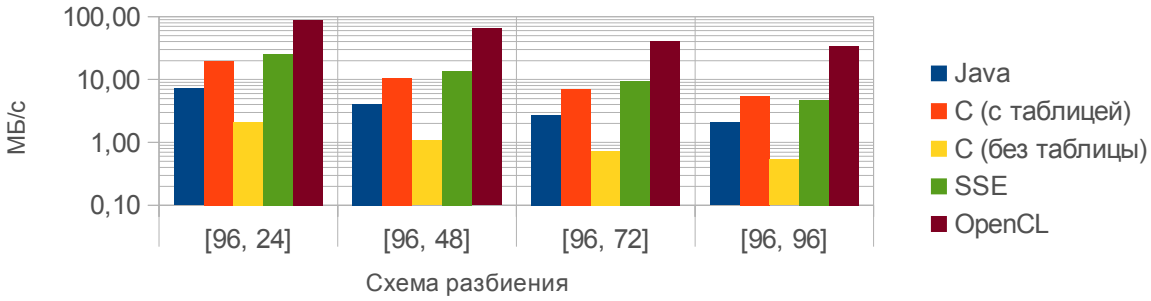

Рисунок 19. Гистограмма с результатами реализаций на (96, k)-схемах

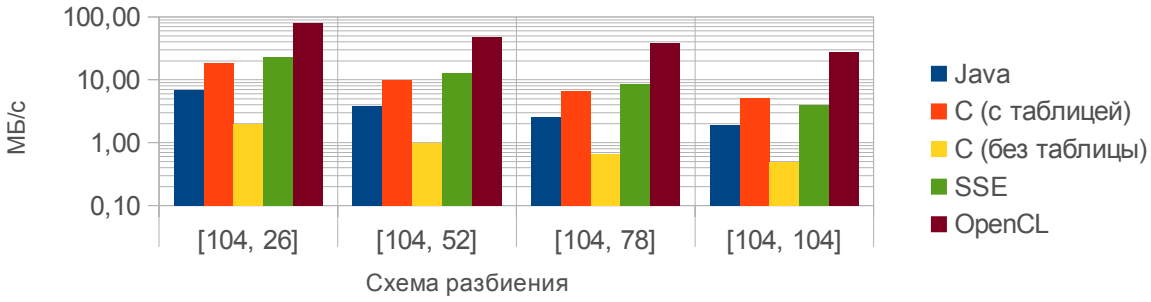

Рисунок 20. Гистограмма с результатами реализаций на (104, k)-схемах

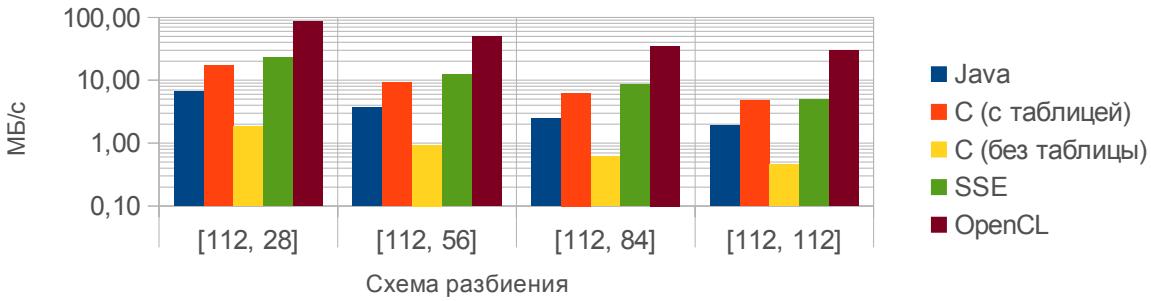

Рисунок 21. Гистограмма с результатами реализаций на (112, k)-схемах

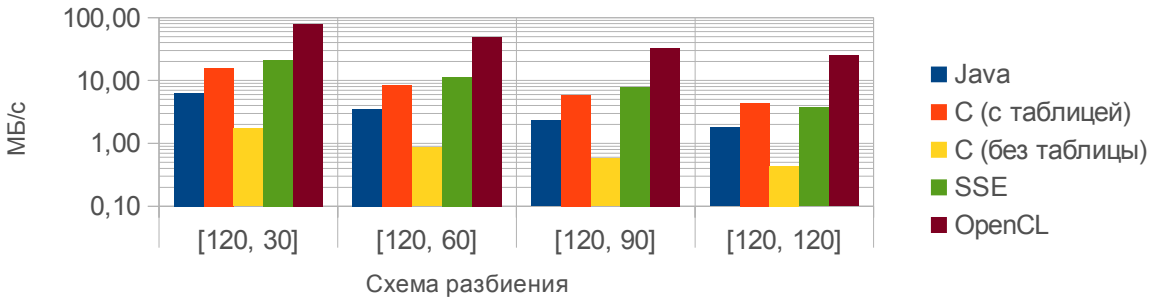

Рисунок 22. Гистограмма с результатами реализаций на (120, k)-схемах

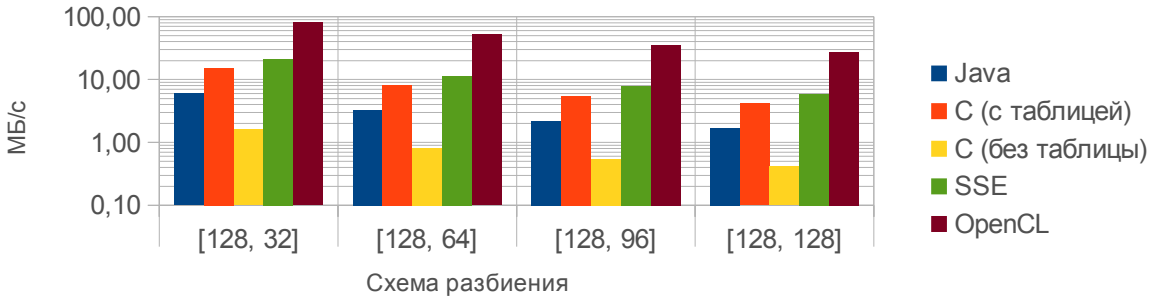

Рисунок 23. Гистограмма с результатами реализаций на (128, k)-схемах