Модифицированный фильтр Блума в гибридной CPU/GPU реализации для CUDA-ориентированных суперкомпьютерных систем хранения и обработки больших объемов данных

Modified Bloom Filter in Hybrid CPU/GPU Implementation for CUDA-oriented Supercomputer Large Volume Data Storage and Processing System

Васильев Николай Петрович
Ровнягин Михаил Михайлович

Аннотация. В статье рассматриваются проблемы реализации модифицированного фильтра Блума как дополнительного модуля для систем хранения данных больших объемов на суперкомпьютерах с гибридной архитектурой CPU/GPU. Предлагается использование модификации фильтра со счетчиками, что позволяет отслеживать не только добавление, но и удаление данных; применяемые хеш-функции основаны на делении полиномов. Проводится сравнительный анализ работы последовательной CPU-версии и гибридной версии реализации Блум-фильтра, основанной на использовании технологии NVIDIA CUDA.

Abstract: The paper examines problems of modified Bloom filter CPU/GPU implementation as an additional module for large scale data storage systems. It is proposed to use of modified filter with counters which can monitor both data addition and deletion. Hash functions used in this filter are based on polynom division. A comparative analysis of serial CPU-version and a hybrid version of the implementation of Bloom filter based on the use of technology NVIDIA CUDA.

Ключевые слова: фильтр Блума, гибридная архитектура CPU/GPU, NVIDIA CUDA, хеширование, основанное на делении полиномов.

Keywords: Bloom filter, hybrid CPU/GPU architecture, NVIDIA CUDA, polynom division based hashing.

Введение

В настоящее время во многих областях человеческой деятельности происходит бурный рост объема информации, что предъявляет весьма высокие требования к системам хранения, передачи и обработки данных. Наглядными примерами тому являются физические эксперименты мирового масштаба, в частности, LHC – Большой адронный коллайдер. По официальным данным, распределенная грид-система проекта LHC ориентирована на хранение, распределение и анализ данных в объеме около 25 петабайт (т.е. 25 млн. гигабайт) [1].

Для хранения больших объемов данных (от сотен терабайт до десятков петабайт) создаются специализированные хранилища различной архитектуры. Для ускорения доступа к данным, хранимым подобными системами, разработчики зачастую отходят от канонов построения классических SQL-систем [2]. В угоду производительности осуществляется децентрализация управления базой данных, исключаются некоторые проверки при добавлении новых данных. В результате подобных действий, среди прочего, возрастает вероятность ошибок поиска данных. Кроме того, значительное число поисковых запросов может быть адресовано к отсутствующим в хранилище данным, и время на обработку этих запросов будет затрачиваться. В силу большого объема информации, суммарное время обработки бесполезных запросов может быть значительным.

Таким образом, задача отсечения запросов на поиск данных, которых нет в хранилище, становится актуальной, и чем выше будет достигнута произво-

дительность фильтрации «лишних» запросов, тем больше будет производительность работы хранилища в целом.

Для частичного решения проблемы подавления заведомо лишних запросов поиска еще в 1970 году Бертоном Блумом была предложена вероятностная структура данных – фильтр Блума (ФБ) [3]. ФБ отражает сведения о добавленных элементах в произвольном хранилище данных, содержащем информацию в формате «ключ-значение». В классической реализации ФБ представляет собой надстройку над хранилищем в виде битового массива длиной M (вектор Блума, ВБ), который изначально заполнен нулями. При добавлении нового элемента данных в хранилище происходит расчет K хеш-функций (ХФ) от ключа, причем значения ХФ должны находиться в промежутке от нуля до M-1. В разряды ВБ, номера которых определяются результатами хеш-функций, записываются единицы.

Поиск данных в хранилище также происходит с применением фильтра Блума. Перед непосредственным обращением в хранилище по ключу происходит расчет K хеш-функций для этого ключа и проверка наличия единиц на вычисленных позициях. Наличие хотя бы одного нуля в любом из найденных разрядов ВБ однозначно указывает на то, что данные в хранилище отсутствуют. В том случае, когда на всех вычисленных позициях ВБ обнаруживаются единицы, возможны два варианта. Первый из них – данные действительно присутствуют, и операция поиска в хранилище будет успешной. Второй вариант – информация в хранилище отсутствует, а единицы были выставлены во время добавления других элементов, т.е. при вычислении хеш-функций для других ключей (ложноположительное срабатывание фильтра Блума). Причиной этого является свойство самих ХФ – проявление коллизий, когда два разных аргумента приводят к одинаковому результату хеширования [4].

Таким образом, фильтр Блума позволяет уменьшить общее количество запросов к хранилищу данных, и основным показателем эффективности работы ФБ является вероятность ложноположительного срабатывания (P). Чем меньше этот показатель, тем меньше будет совершено лишних запросов к физическим носителям.

Вероятность того, что после добавления N элементов в хранилище при изначально пустом векторе Блума, его i-й разряд останется равным нулю, выражается формулой:

$$P_i = (1 - \frac{1}{M})^{KN} = e^{-\frac{KN}{M}}, \qquad (1)$$

для достаточно большого M, ввиду второго замечательного предела.

Вероятность того, что при проверке наличия в хранилище элемента, не равного ни одному из реально добавленных, все K-бит, вычисленных хеш-функциями, окажутся равными единице, составляет:

$$\mathrm{P} = (1 - \mathrm{e}^{-\frac{\mathrm{KN}}{\mathrm{M}}})^{\mathrm{K}} \qquad (2)$$

Из формулы (2) можно сделать вывод, что с ростом N вероятность ложноположительного срабатывания возрастает, а при увеличении M - уменьшается.

При фиксированных N и M, минимизировать вероятность возникновения ложноположительного результата можно, выбрав K следующим образом:

$$\mathrm{K} = \frac{\mathrm{M}}{\mathrm{N}} \ln 2 \qquad (3)$$

При выборе размера вектора важно учесть предполагаемое число элементов в хранилище в будущем и изначально задать вероятность ложного срабатывания, тогда рассчитать длину вектора Блума можно по следующей формуле:

$$M = -\frac{N\ lnP}{(\ln 2)^2} \qquad (4)$$

Фильтр Блума несложно организовать, руководствуясь вышеописанными соотношениями. Он позволяет уменьшить нагрузку на подсистему доступа к данным, однако, обладает одним существенным недостатком. Из классического битового вектора Блума нельзя удалять элементы. А значит, он применим только в системах, где данные запрашиваются исключительно на чтение – иначе количество ложных срабатываний будет неминуемо расти.

Решить эту проблему можно, используя модификацию фильтра Блума со счетчиками. При таком подходе вектор Блума представляет собой массив L-разрядных счетчиков, хранящих значение от 0 до 2^L-1. При добавлении новых данных в хранилище хеш-функция вычисляет значение ключа и инкрементирует нужный счетчик. В случае, когда счетчик достиг максимального значения, его состояние не меняется. При удалении счетчик уменьшается на единицу или не изменяет своего состояния, если его значение равно нулю или максимуму. Благодаря подобным изменениям можно частично решить проблему удаления данных.

Таким образом, реализация модифицированного фильтра Блума для поддержки систем, хранящих большие объемы информации, является актуальной задачей.

Принципы работы модифицированного фильтра Блума

При проектировании модифицированного фильтра Блума важно заранее оценить требуемую разрядность его счетчиков (L) и размер вектора (M). При фиксированном M с ростом разрядности счетчиков увеличивается «порог» вместимости – максимальное значение счетчика, начиная с которого фильтр переходит в классический режим работы (только на добавление). Увеличение L на l приводит к увеличению размера вектора на M, что приводит к значительному росту затрат по памяти.

Одним из показателей правильности работы модифицированного фильтра Блума является то, что значения счетчиков должны различаться незначительно. Достичь этого можно, применяя качественную хеш-функцию, дающую равномерное распределение своих значений на всем интервале от 0 до M-1.

В работе [5] был предложен класс хеш-функций, выполняющих вычисление остатка от деления произвольного полинома на полином-делитель вида:

$$\Phi_{\deg}(x)=x^{\deg}+q_{\deg-1}x^{\deg-1}+\ldots+q_ix^i+\ldots+q_1x+1:\deg\in \mathrm{N},\ \forall i=\overline{1,\deg-1},\ q_i\in\{0,1\} \quad (5)$$

Обработка очередной компоненты ключа Key_i выполняется по формулам:

$$\begin{cases} y_1^i = y_{\deg}^{i-1} + Key_i \\ y_2^i = y_1^{i-1} + q_{\deg-1}y_{\deg}^{i-1} \\ \vdots \\ y_k^i = y_{k-1}^{i-1} + q_{\deg-k+1}y_{\deg}^{i-1} \\ \vdots \\ y_{\deg}^i = y_{\deg-1}^{i-1} + q_1 y_{\deg}^{i-1} \end{cases}, \quad k=\overline{2,\deg} \qquad (6)$$

Результат хеширования (остаток от деления) формируется во внутренних переменных $y_k, k=\overline{1,\deg}$ ХФ этого класса выполняются за время порядка O(n*deg), где n – степень полинома-делимого a deg - полинома-делителя и могут быть запрограммированы на языках высокого уровня в обобщенном виде, т.е. как одна функция. Полиномом-делимым является ключ; полином-делитель задается как массив коэффициентов $q_i\in\{0,1\}, \forall i=\overline{1,\deg-1}$, где deg – некоторая максимальная степень полинома. Отметим, что никаких реальных операций умножения не

требуется: в силу двоичности коэффициентов все сводится к сложению. В работе [4] указано, что ХФ, основанные на делении, обеспечивают хорошее распределение результатов хеширования, что и было экспериментально подтверждено для преобразований вида (6) в работе [5]. Функция, выполняющая хеширование в соответствии с (6) приведена в листинге 1.

```
void bhash(unsigned char* buf, unsigned char len, unsigned char* yold, unsigned char* y, unsigned char* q, unsigned char deg){
  for (unsigned char i=0; i<len; i++){
        y[1]=yold[deg]+buf[i];
        for (unsigned char j=2; j<=deg; j++){
              if (q[deg-j+1]==1)
                    y[j]=yold[j-1]+yold[deg];
              else
                    y[j]=yold[j-1];
        }
        for (unsigned char k=1; k<=deg; k++)
              yold[k]=y[k];
  }
}
```

Листинг 1. Реализация функции хеширования на языке Си

В функцию передаются: указатель на хешируемое слово (buf), длина хешируемой последовательности (len), указатели на массивы элементов хранения промежуточных результатов (y и yold), указатель на массив полиномиальных коэффициентов (q) и степень полинома (deg). Для инициализации множества хеш-функций использовался файл, содержащий полиномиальные коэффициенты примитивных полиномов.

При проведении эксперимента было решено абстрагироваться от типа хранилища. Предметом хранения являлись английские слова (ключи абстрактного ассоциативного хранилища), загружаемые из файла с текстом. В качестве текстов были выбраны официальные спецификации сетевых протоколов [6]. Для начальной инициализации вектора Блума сформирован массив alphabet (алфавит), который был считан из файла, после чего из него были удалены повторяющиеся элементы и слова, длина которых меньше степени полинома плюс два: для уменьшения числа коллизий при хешировании с разными полиномиальными коэффициентами. Из текстового файла input (половина слов которого встречается в alphabet) в массив входных данных были прочитаны слова для проверки их на членство в хранилище с помощью вектора Блума.

Далее по формулам (1), (2), (3) происходит расчет эффективных значений M и K для хранилища размером, равным длине массива уникальных слов из alphabet и предполагаемой. Все вышеперечисленные параметры могут быть заданы пользователем вручную из консоли.

Произвольным образом из файла полиноминальных коэффициентов выбирается K разных наборов. Для каждого из K наборов полиномиальных коэффициентов при добавлении очередного элемента функция bhash вычисляет хеш-функцию, по значению которой выбирается индекс элемента вектора Блума, по которому происходит увеличение счетчика.

Программа подсчитывает количество совпадений слов из алфавита и слов из файла input. После этого происходит проверка слов из input в предполагаемом хранилище с использованием фильтра Блума. Фиксируется количество

положительных (элемент есть в хранилище) ответов фильтра. Определив это значение и учитывая реальное количество совпадений, можно рассчитать экспериментальное значение ложноположительных результатов.

Далее происходит удаление половины элементов алфавита так, чтобы число совпадений со словами из файла input уменьшилось в два раза. Снова происходит проверка слов из файла input на членство с применением фильтра Блума и расчет экспериментального количества ложноположительных срабатываний.

Реализация модифицированного фильтра Блума с применением технологий NVIDIA CUDA

Испытания проводились на вычислительном кластере из 12 компьютеров следующей конфигурации: Intel Core i7 – 2600, 4Gb DDR3, NVIDIA GeForce GTX550Ti (1Gb GDDR5). Вышеуказанная видеокарта поддерживает CUDA Compute capability 2.1, что означает возможность использовать до 48K разделяемой памяти на каждый блок.

Возможны несколько подходов к реализации распараллеливания работы фильтра Блума. Например, можно хранить копию вектора Блума в разделяемой памяти и переложить большинство функций на GPU [7]. И, наоборот, можно все функции по управлению вектором Блума оставить за центральным процессором, а за GPU закрепить лишь обязанности по вычислению хеш-функций.

Первый способ должен принести потенциально больший эффект от распараллеливания, однако, содержит в себе потенциальные трудности в реализации. Предложенный выше вариант модифицированного фильтра Блума требует сравнительно больших затрат по памяти для хранения вектора. Так для предполагаемого хранилища на 10000 элементов (что весьма немного) и вероятности ложноположительного срабатывания 0.1, оптимальным будет Блум-вектор размером ~65K, что несколько больше максимального размера разделяемой памяти на блок. Хранение вектора в глобальной памяти устройства будет сопряжено с ростом числа конфликтов по доступу к памяти и ее большой латентностью.

Второй вариант не столь сильно связан с аппаратными ограничениями на размер предполагаемого хранилища и, соответственно, размер Блум-вектора. В разделяемой памяти выгодно хранить промежуточные значения (y,yold), полиноминальные коэффициенты и результаты хеширования в процессе преобразования их к значениям индексов вектора Блума.

В независимости от варианта реализации работы параллельного алгоритма на CUDA, важно обеспечить пакетную обработку поступающих запросов (добавление, удаление, проверка). В реальной системе запросы на проверку членства происходят гораздо чаще двух других видов запросов. Таким образом, процесс обработки входного потока заявок может выглядеть следующим образом: выбор из очереди подряд идущих запросов одного типа, вычисление хеш-функций, изменение вектора Блума.

В рамках данного исследования был выбран второй вариант реализации. На рисунке 1 показан пример расположения промежуточных значений хеширования (y,yold) в разделяемой памяти для параметров запуска ядра <<<2,4>>>.

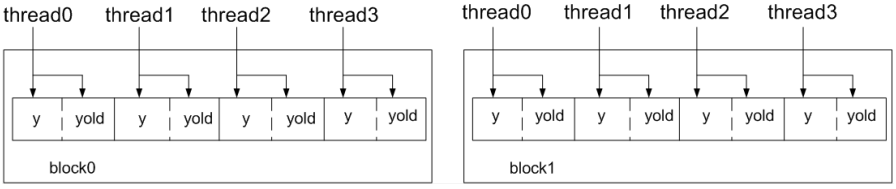

Рис. 1 Расположение промежуточных значений в разделяемой памяти GPU

Каждая нить вычисляет свою хеш-функцию. Количество блоков для запуска выбирается исходя из расчета 1 блок на один потоковый мультипроцессор, что позволяет сохранять все промежуточные результаты в разделяемой памяти, которая находится на кристалле мультипроцессора. Число нитей в CUDA Compute capability 2.1 ограничено 1024 единицами. Соответственно, одно устройство NVIDIA GeForce GTX550Ti (4 потоковых мультипроцессора) способно обрабатывать до 4096 запросов одновременно. При этом каждая нить вычисляет все K хеш-функций для каждого запроса.

Исследование эффективности функционирования модифицированного фильтра Блума

Измерение времени выполнения последовательной части программы на CPU осуществлялось при помощи функции clock(), которая возвращает текущее значение в тактах процессора [8]. Вычитая из значения, возвращенного этой функцией после вычислений, значение перед вычислениями, получаем количество тактов, затраченных на выполнение. Если разделить это число на количество тактов в секунду, получится время в секундах.

Определение времени выполнения тестовой программы с применение графического процессора проводилось при помощи API событий CUDA [9]. Событием в CUDA называется временная метка GPU, зафиксированная в определенный пользователем момент времени.

Параметры, переданные программе, и результаты ее выполнения (экспериментальные значения вероятности ложноположительного срабатывания и временные характеристики работы), записывались в лог-файлы. Процесс тестирования был автоматизирован при помощи BAT-файлов. В качестве файлов с ключами для проверки членства использовались текстовые файлы разного размера (input7 - 38Mb, input6 - 19Mb, input5 - 10Mb, input4 - 5Mb, input3 - 2400K, input2 - 1200K, input1 - 600K). Каждый файл наполовину состоял из alphabet-слов. Далее приведены диаграммы некоторых результатов, полученных во время испытаний.

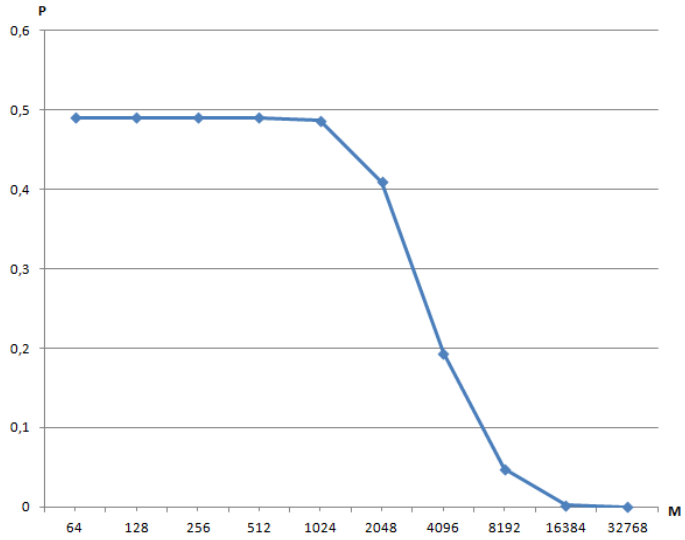

Рис. 2. График зависимости вероятности ложноположительного срабатывания от длины Блум-вектора

Зависимость экспериментального значения вероятности от длины вектора Блума представлена на рисунке 2. При уменьшении M график стремится к значению вероятности 0,5, так как половина из слов тестового файла (input) содержится в векторе.

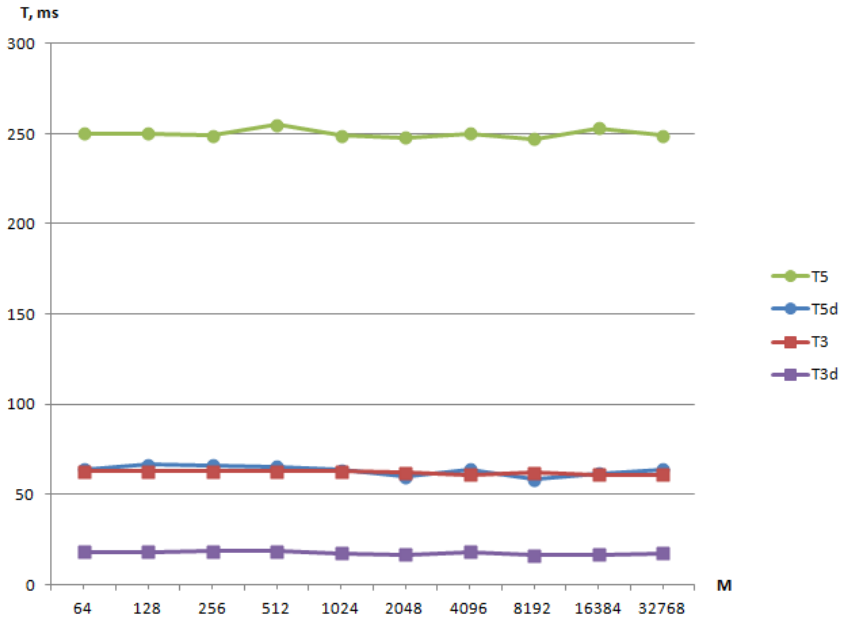

Рис. 3 График зависимости времени выполнения программы от длины Блум-вектора

На рисунке 3 представлены зависимости времени исполнения CPU-версии программы проверки членства (T5, T3) и CPU/GPU-версии (T5d, T3d) от M для файлов input5 и input3 соответственно. На этом и других рисунках обозначение Ti и Tid соответствует результату обработки тестового файла $input_i$ на CPU и CPU/GPU соответственно Как видно из графиков, с ростом M время выполнения не увеличивается, потому что основные вычислительные затраты связаны с вычислением хеш-функций, а не с размером вектора. Также можно заметить, что графики T3 и T5d почти совпадают. Если принять во внимание, что размер input5 в 4 раза больше input3 можно сделать вывод, что GPU-версия программы в данном случае работает в 4 раза быстрее CPU-версии.

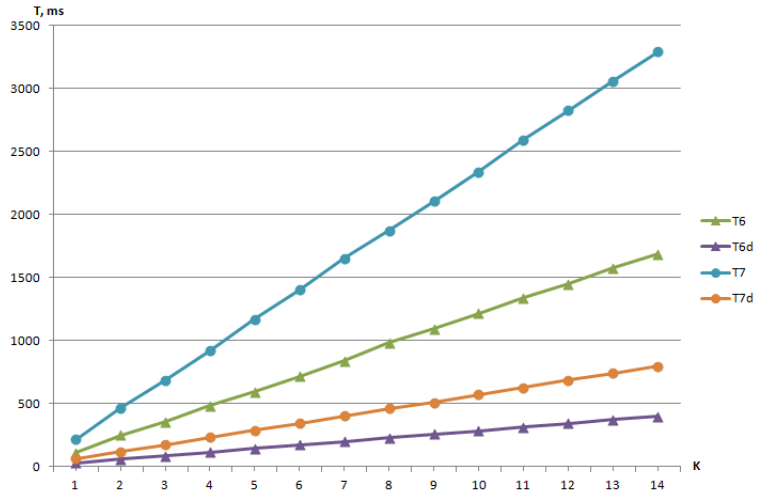

Рис. 4 График зависимости времени выполнения программы от количества хеш-функций

На рисунке 4 изображены зависимости времени выполнения последовательного варианта программы проверки членства (T7, T6) и параллельного (T7d, T6d) от K для файлов input6 и input7 соответственно. Можно заметить, что различие в скорости выполнения CPU и GPU версий программы с ростом K

увеличилось с 1,5 до 4,2 для K=14. Опираясь на формулы (3) и (4), можно утверждать, что K зависит от P следующим образом:

$$K = -\frac{lnP}{\ln 2} \qquad (7)$$

Для систем с желаемым значением вероятности ложноположительного срабатывания 0,01 количество функций хеширования составит K=7. Из графиков T7 и T7d видно, что при K=7 GPU-версия работает в 3,5 быстрее CPU-версии. Примечательным является и тот факт, что с ростом количества запросов на проверку членства наклон графиков увеличивается. Таким образом, гибридная версия программы тем эффективнее, чем больше нагрузка на систему.

На рисунке 5 представлены экспериментальные графики зависимости времени выполнения последовательной реализации программы проверки членства (T4, T3) и параллельной версии (T4d, T3d) от P для файлов input4 и input3 соответственно. С уменьшением значения желаемой вероятности возникновения ложноположительного срабатывания возрастает время обработки запросов (вследствие роста K, автоматически вычисляемого из P, N – фиксировано). Графики имеют вид гипербол. Можно заметить, что применение CUDA-вычислителя позволяет достичь меньшего количества ложноположительных срабатываний фильтра при одинаковых временных издержках.

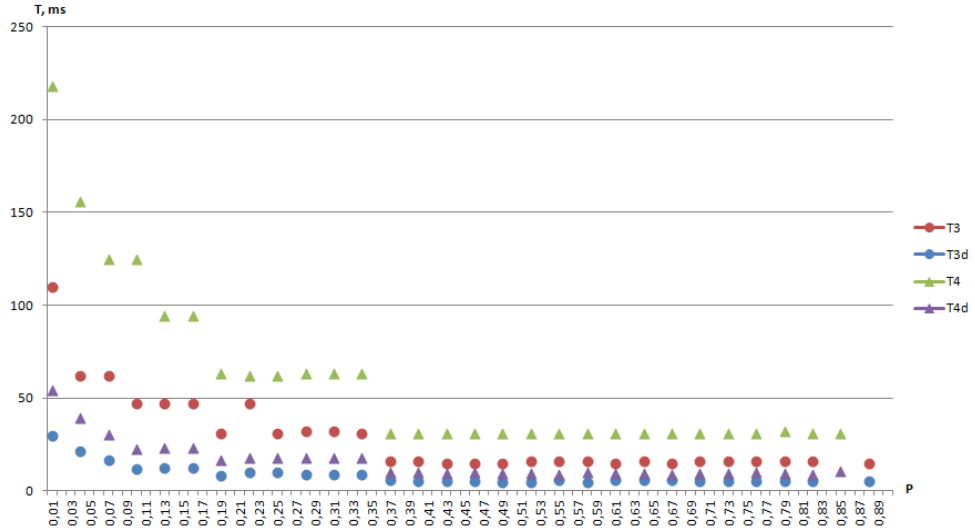

Рис. 5 График зависимости времени выполнения программы от вероятности ложноположительного срабатывания

Заключение

Были рассмотрены принципы работы фильтра Блума, предложена его модификация, позволяющая работать не только на добавление, но и на удаление элементов. Описаны основные характеристики фильтра Блума и их влияние на качество его работы. Рассмотрены основные аспекты реализации последовательной CPU и параллельной GPU-версии фильтра. Проанализированы экспериментальные графики и выявлен рост эффективности гибридного БФ с ростом числа запросов и размера хранилища по сравнению с CPU-версией.

Модифицированный фильтр Блума, ориентированный на гибридные вычислительные системы, по мнению авторов, может эффективно применяться в современных суперкомпьютерах как инструмент для отсечения большого числа бесполезных поисковых запросов при организации разнообразных систем хранения и обработки больших объемов данных.

Литература

1. Worldwide LHC Computing Grid (официальный веб-сайт) URL: http://wlcg.web.cern.ch/ (дата обращения: 1.11.2012).

2. Бартунов О., Велихов П., Симаков Р. Книжник К, Смирнов А. SciDB – новая СУБД для больших объемов научных данных, Суперкомпьютеры, 15(5), 2011.

3. B. Bloom. Space / time trade-offs in hash coding with allowable errors. Communications of ACM, 13(7): 422-426, 1970

4. Кормен Т. Х., Лейзерсон Ч. И., Ривест Р. Л., Штайн К. Алгоритмы: построение и анализ, 2-е издание. : Пер. с англ., М.: Издательский дом "Вильямс", 2005.

5. Васильев Н.П. Разработка и исследование алгоритмов хэширования и генерации псевдослучайных последовательностей: диссертация на соискание ученой степени кандидата технических наук (на правах рукописи), М.:МИФИ, 1998.

6. Internet Engineering Task Force Tools (официальный веб-сайт) URL: http://tools.ietf.org (дата обращения 1.11.2012).

7. Lin Ma, Roger D. Chamberlain, Jeremy D. Buhler, and Mark A. Franklin, “Bloom Filter Performance on Graphics Engines,” in Proc. of International Conference on Parallel Processing, September 2011, pp. 522-531.

8. Бьерн Страуструп, Язык программирования C++. Специальное издание.: Пер. с англ., М.: Бином, 2012.

9. NVIDIA CUDA C Programming Guide 5.0 (официальный веб-сайт) URL: https://developer.nvidia.com/cuda-downloads (дата обращения 1.11.2012).

Сведения об авторах

ФИО: Васильев Николай Петрович
Место работы и должность в настоящее время: НИЯУ МИФИ, доцент
Год окончания учебного заведения и его полное название: 1995 г. Московский инженерно-физический институт
Ученая степень и звание: к.т.н., доцент
Количество печатных работ и монографий: более 60
Область научных интересов: Облачные вычисления, гибридные суперкомпьютерные технологии, защита информации
Контакты: NPVasilyev@mephi.ru, +7(916)339-7937
Name: Vasilyev Nikolay Petrovich
Position: Associate Professor
Year of graduation: 1995
Academic degree: PhD
Number of publications and monographs: More than 60 publications
Research interests: Cloud computing, Hybrid supercomputer technologies, Information Security
Contact: NPVasilyev@mephi.ru, +7(916)339-7937

ФИО: Ровнягин Михаил Михайлович
Место работы и должность в настоящее время: аспирант кафедры компьютерные системы и технологии НИЯУ МИФИ
Год окончания учебного заведения и его полное название: 2012 Национальный исследовательский ядерный университет «МИФИ»
Ученая степень и звание: нет
Количество печатных работ и монографий: 3

Область научных интересов: Суперкомпьютерные технологии, организация поиска данных в высоконагруженных системах, разработка и верификация аппаратуры
Контакты: rovnyagin@gmail.com +7(915)2270086
Name: Michael M. Rovnyagin
Position: a postgraduate student in the Department of Computer Systems and Technologies NRNU MEPhI
Year of graduation: 2012 National Research Nuclear University «MEPhI»
Academic degree: no
Number of publications and monographs: 3
Research interests: Supercomputer technology, data search in high load systems, development and verification of digital hardware
Contact: rovnyagin@gmail.com +7(915)2270086