tangro вчера в 12:49 Разработка

«Магическая константа» 0x5f3759df

http://h14s.p5r.org/2012/09/0x5f3759df.html?mwh=1

Спортивное программирование, Программирование, Ненормальное программирование, Занимательные задачки, Блог компании Инфопульс Украина

Перевод

В этой статье мы поговорим о «магической» константе 0x5f3759df, лежащей в основе элегантного алгоритмического трюка для быстрого вычисления обратного квадратного корня.

Вот полная реализация этого алгоритма:

```
float FastInvSqrt(float x) {
  float xhalf = 0.5f * x;
  int i = *(int*)&x;  // представим биты float в виде целого числа
  i = 0x5f3759df - (i >> 1);  // какого черта здесь происходит ?
  x = *(float*)&i;
  x = x*(1.5f-(xhalf*x*x));
  return x;
}
```

Этот код вычисляет некоторое (достаточно неплохое) приближение для формулы

$$\frac{1}{\sqrt{x}}$$

Сегодня данная реализация уже хорошо известна, и стала она такой после появления в коде игры Quake III Arena в 2005 году. Её создание когда-то приписывали Джону Кармаку, но выяснилось, что корни уходят намного дальше – к Ardent Computer, где в сере 80-ых её написал Грег Уолш. Конкретно та версия кода, которая показана выше (с забавными комментариями), действительно из Quake.
В этой статье мы попробуем разобраться с данным хаком, математически вывести эту самую константу и попробовать обобщить данный метод для вычисления произвольных степеней от -1 до 1.

Да, понадобится немного математики, но школьного курса будет более, чем достаточно.

Зачем?

Зачем вообще нам может понадобиться считать обратный квадратный корень, да ещё и пытаться делать это настолько быстро, что нужно реализовывать для этого специальные хаки? Потому, что это одна из основных операций в 3D программировании. При рабо 3D графикой используются нормали поверхностей. Это вектор (с тремя координатами) длиной в единицу, который нужен для опис освещения, отражения и т.д. Таких векторов нужно много. Нет, даже не просто много, а МНОГО. Как мы нормализуем (приводим д к единице) вектор? Мы делим каждую координату на текущую длину вектора. Ну или, перефразируя, умножаем каждую координат вектора на величину:

$$\frac{1}{\sqrt{x^2+y^2+z^2}}$$

Расчёт $x^2 + y^2 + z^2$ относительно прост и работает быстро на всех типах процессоров. А вот расчёт квадратного корня и деление – дорогие операции. И вот поэтому – встречайте алгоритм быстрого извлечения обратного квадратного корня — FastInvSqrt.

Что он делает?

Что же делает вышеуказанная функция для вычисления результата? Она состоит из четырёх основных шагов. Первым делом она б входное число (которое пришло нам в формате float) и интерпретирует его биты как значение новой переменной i типа integer (це число).

```
int i = *(int*)&x;         // представим биты float в виде целого числа
```

Далее над полученным целым числом выполняется некоторая целочисленная арифметическая операция, которая работает достат быстро и даёт нам некоторую аппроксимацию требуемого результата

```
i = 0x5f3759df - (i >> 1);  // какого черта здесь происходит ?
```

То, что мы получили в результате данной операции, ещё не является, собственно, результатом. Это лишь целое число, биты котор представляют некоторое другое число с плавающей запятой, которое нам нужно. А значит, необходимо выполнить обратное преобразование int во float.

```
x = *(float*)&i;
```

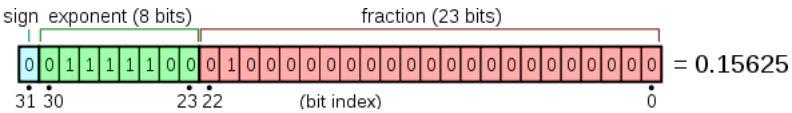
И, наконец, выполняется одна итерация метода Ньютона для улучшения аппроксимации.

```
x = x*(1.5f-(xhalf*x*x));
```

И вот теперь у нас имеется отличная аппроксимация операции извлечения обратного квадратного корня. Последняя часть алгорит (метод Ньютона) – достаточно тривиальная вещь, я не буду на этом останавливаться. Ключевой частью алгоритма является шаг № выполнение некоторой хитрой арифметической операции над целым числом, полученным от интерпретации битов числа с плаваю запятой в качестве содержимого типа int. На этом мы и сфокусируемся.

Какого черта здесь происходит?

Для понимания дальнейшего текста нам нужно вспомнить формат, в котором хранятся в памяти числа с плавающей запятой. Я опи здесь только то, что важно здесь для нас (остальное всегда можно подсмотреть в Википедии). Число с плавающей запятой хранит как комбинация трёх составляющих: знака, экспоненты и мантиссы. Вот биты 32-битного числа с плавающей запятой:



Первым идёт бит знака, дальше 8 битов экспоненты и 23 бита мантиссы. Поскольку мы здесь имеем дело с вычислением квадратн корня, то я предположу, что мы будем иметь дело лишь с положительными числами (первый бит всегда будет 0).

Рассматривая число с плавающей запятой как просто набор битов, экспонента и мантисса могут восприниматься как просто два положительных целых числа. Давайте обозначим их, соответственно, E и M (поскольку дальше мы будем часто на них ссылаться). другой стороны, интерпретируя биты числа с плавающей запятой, мы будем рассматривать мантиссу как число между 0 и 1, т.е. в нули в мантиссе будут означать 0, а все единицы – некоторое число очень близкое (но всё же не равное) 1. Ну и вместо того, чтоб интерпретировать экспоненту как беззнаковое 8-битное целое число, давайте вычтем смещение (обозначим его как B), чтобы получить знаковое целое число в диапазоне от -127 до 128. Давайте обозначим float-интерпретацию этих значений как e и m. Чтоб путаться, будем использовать прописные обозначения (E, M) для обозначения целочисленных значений и строчные (e, m) – для обозначения чисел с плавающей запятой (float).

Преобразование из одного в другое тривиально:

$$m = \frac{M}{L}$$

$$e = E - B$$

В этих формулах для 32-битных чисел L = 2^23, а B = 127. Имея некоторые значения e и m, можно получить само представляемое ч число:

$$(1+m)2^e$$

и значение соответствующей им целочисленной интерпретации числа:

$$M+LE$$

Теперь у нас есть почти все кусочки паззла, которые нужны для объяснения “хака” в коде выше. Итак, нам на вход приходит некот число x и требуется рассчитать его обратный квадратный корень:

$$y=\frac{1}{\sqrt{x}}=x^{-\frac{1}{2}}$$

По некоторым причинам, которые вскоре станут понятны, я начну с того, что возьму логарифм по основанию 2 от обеих частей да уравнения:

$$\log_2 y=-\frac{1}{2}\log_2 x$$

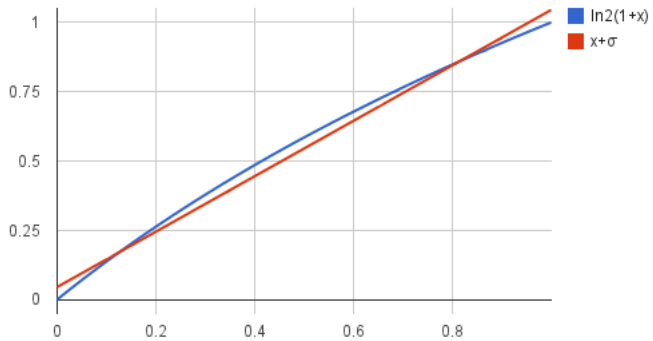
Поскольку числа, с которыми мы работаем, на самом деле являются числами с плавающей запятой, мы можем представить x и y согласно вышеуказанной формуле представления таких чисел:

$$\log_2(1+m_y)+e_y=-\frac{1}{2}(\log_2(1+m_x)+e_x)$$

Ох уж эти логарифмы. Возможно, вы не пользовались ими со школьных времён и немного подзабыли. Не волнуйтесь, немного дал мы избавимся от них, но пока что они нам ещё нужны, так что давайте посмотрим, как они здесь работают.
В обеих частях уравнения у нас находится выражение вида:

$$\log_2(1+v)$$

где v находится в диапазоне от 0 до 1. Можно заметить, что для v от 0 до 1 эта функция достаточно близка к прямой линии:



Ну или в виде выражения:

$$\log_2(1+v)\approx v+\sigma$$

Где σ – некоторая константа. Это не идеальное приближение, но мы можем попытаться подобрать σ таким образом, чтобы оно бы достаточно неплохим. Теперь, использовав его, мы можем преобразовать вышеуказанное равенство с логарифмами в другое, уже строго равное, но достаточно близкое и, самое главное, СТРОГО ЛИНЕЙНОЕ выражение:

$$m_y+\sigma+e_y\approx-\frac{1}{2}(m_x+\sigma+e_x)$$

Это уже что-то! Теперь самое время прекратить работать с представлениями в виде чисел с плавающей запятой и перейти к целочисленному представлению мантиссы и экспоненты:

$$\frac{M_y}{L}+\sigma+E_y-B\approx-\frac{1}{2}(\frac{M_x}{L}+\sigma+E_x-B)$$

Выполнив ещё несколько тривиальных преобразований (можно пропустить детали), мы получим нечто, выглядящее уже довольно знакомо:

$$\frac{M_y}{L} + E_y \approx -\frac{1}{2}(\frac{M_x}{L} + \sigma + E_x - B) - \sigma + B$$

$$\frac{M_y}{L} + E_y \approx -\frac{1}{2}(\frac{M_x}{L} + E_x) - \frac{3}{2}(\sigma - B)$$

$$M_y + LE_y \approx \frac{3}{2}L(B - \sigma) - \frac{1}{2}(M_x + LE_x)$$

Посмотрите внимательно на левую и правую части последнего уравнения. Как мы видим, у нас получилось выражение целочислен формы требуемого значения y, выраженное через линейного вида формулу, включающую целочисленное представление значения

$$\mathbf{I_y} \approx \frac{3}{2}L(B - \sigma) - \frac{1}{2}\mathbf{I_x}$$

Говоря простыми словами: "y (в целочисленной форме) – это некоторая константа минус половина от целочисленной формы x". В кода это:

```
i = K - (i >> 1);
```

Очень похоже на формулу в коде функции в начале статьи, правда?

Нам осталось найти константу K. Мы уже знаем значения B и L, но ещё не знаем чему равно σ.
Как вы помните, σ – это некоторое "поправочное значение", которое мы ввели для улучшения аппроксимации функции логарифма прямой линии на отрезке от 0 до 1. Т.е. мы можем подобрать это число сами. Я возьму число 0.0450465, как дающее неплохое приближение и использованное в оригинальной реализации. Используя его, мы получим:

$$\frac{3}{2}L(B - \sigma) = \frac{3}{2}2^{23}(127 - 0.0450465) = 1597463007$$

Угадаете, как число 1597463007 представляется в HEX? Ну, конечно, это 0x5f3759df. Ну, так и должно было получиться, поскольку выбрал σ таким образом, чтобы получить именно это число.

Таким образом, данное число не является битовой маской (как думают некоторые люди просто потому, что оно записано в hex-фо а результатом вычисления аппроксимации.

Но, как сказал бы Кнут: "Мы пока что лишь доказали, что это должно работать, но не проверили, что это действительно работает" Чтобы оценить качество нашей формулы, давайте нарисуем графики вычисленного таким образом значения обратного квадратно корня и настоящей, точной его реализации:

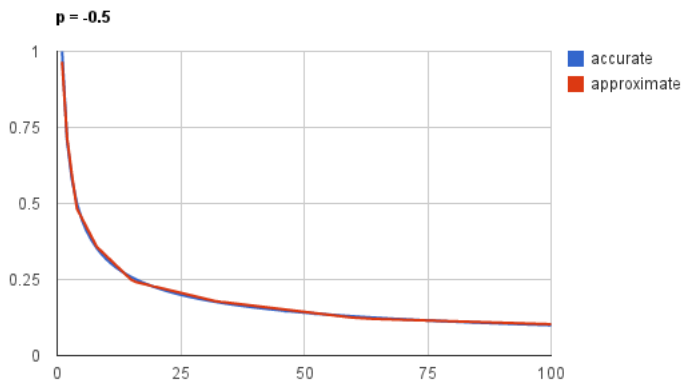


График построен для чисел от 1 до 100. Неплохо, правда? И это никакая не магия, не фокус, а просто правильное использование несколько, возможно, экзотичного трюка с представлением чисел с плавающей запятой в виде целочисленных и наоборот.

Но и это ещё не всё!

Все вышеуказанные преобразования и выражения дали нам не только объяснение константы 0x5f3759df, но и ещё несколько ценн выводов.

Во-первых, поговорим о значениях чисел L и B. Они определяются не нашей задачей по извлечению обратного квадратного корня форматом хранения числа с плавающей запятой. Это означает, что тот же трюк может быть проделан и для 64-битных и для 128-

битных чисел с плавающей запятой – нужно лишь повторить вычисления для рассчета других констант.

Во-вторых, нам не очень-то и важно выбранное значение σ. Оно может не давать (да и на самом деле – не даёт) лучшую аппроксимацию функции x + σ к логарифму. σ выбрано таким, поскольку оно даёт лучший результат совместно с последующим применением алгоритма Ньютона. Если бы мы его не применяли, то выбор σ являлся бы отдельной интересной задачей сам по себ эта тема раскрыта в других публикациях.

Ну и в конце-концов, давайте посмотрим на коэффициент “-1/2” в финальной формуле. Он получился таким из-за сути того, что мы хотели вычислить (“обратного квадратного корня”). Но, в общем, степень здесь может быть любой от -1 до 1. Если мы обозначим степень как p и обобщим все те же самые преобразования, то вместо “-1/2” мы получим:

$$\mathbf{I_y} \approx (1-p)L(B-\sigma) + p\mathbf{I_x}$$

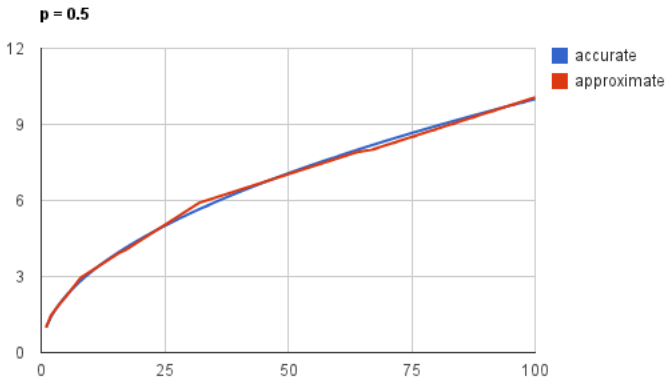
Давайте положим p=0.5 Это будет вычисление обычного (не обратного) квадратного корня числа:

$$\mathbf{I_y} \approx K_{\frac{1}{2}} + \frac{1}{2}\mathbf{I_x}$$
$$K_{\frac{1}{2}} = \frac{1}{2}L(B-\sigma) = \frac{1}{2}2^{23}(127-0.0450465) = \texttt{0x1fbd1df5}$$

В виде кода это будет:

```
i = 0x1fbd1df5 + (i >> 1);
```

И что, это работает? Конечно, работает:



Это, наверное, хорошо известный способ быстрой аппроксимации значения квадратного корня, но беглый гуглинг не дал мне его названия. Возможно, вы подскажете?
Данный способ будет работать и с более “странными” степенями, вроде кубического корня:

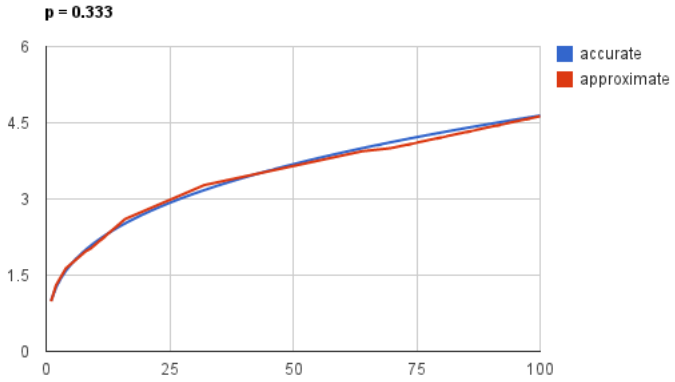
$$\mathbf{I_y} \approx K_{\frac{1}{3}} + \frac{1}{3}\mathbf{I_x}$$

$$K_{\frac{1}{3}} = \frac{2}{3}L(B-\sigma) = \frac{2}{3}2^{23}(127-0.0450465) = \texttt{0x2a517d3c}$$

Что в коде будет выражено, как:

```
i = (int) (0x2a517d3c + (0.333f * i));
```

К сожалению, из-за степени 1/3 мы не можем использовать битовые операции сдвига и вынуждены применить здесь умножение н 0.333f. Аппроксимация всё ещё достаточно хороша:



И даже более того!

К этому моменту вы уже могли заменить, что изменение степени вычисляемой функции достаточно тривиально меняет наши вычисления: мы просто рассчитываем новую константу. Это совершенно не затратная операция и мы вполне можем делать это да этапе выполнения кода, для разных требуемых степеней. Если мы перемножим две заранее известных константы:

$$L(B-\sigma) = 2^{23}(127 - 0.0450465) = \texttt{0x3f7a3bea}$$

То сможем вычислять требуемые значения на лету, для произвольной степени от -1 до 1:

```
i = (1 - p) * 0x3f7a3bea + (p * i);
```

Чуть упростив выражение, мы даже можем сэкономить на одном умножении:

```
i = 0x3f7a3bea + p * (i - 0x3f7a3bea);
```

Эта формула даёт нам “магическую” константу, с помощью которой можно рассчитывать на лету разные степени чисел (для степ от -1 до 1). Для полного счастья нам не хватает лишь уверенности в том, что вычисленное таким образом приближенное значение может быть столь же эффективно улучшено алгоритмом Ньютона, как это происходило в оригинальной реализации для обратного квадратного корня. Я не изучал эту тему более глубоко и это, вероятно, будет темой отдельной публикации (наверное, не моей).

Выражение выше содержит новую “магическую константу” 0x3f7a3bea. В некотором смысле (из-за своей универсальности) она д “более магическая”, чем константа в оригинальном коде. Давайте назовём её C и посмотрим на неё чуть внимательнее.

Давайте проверим работу нашей формулы для случая, когда p = 0. Как вы помните из курса математики, любое число в степени 0 равно единице. Что же будет с нашей формулой? Всё очень просто – умножение на 0 уничтожит второе слагаемое и у нас останет

```
i = 0x3f7a3bea;
```

Что, действительно является константой и, будучи переведённой во float-формат, даст нам 0.977477 – т.е. “почти 1”. Поскольку мы имеем дело с аппроксимациями – это неплохое приближение. Кроме того, это говорит нам кое-что ещё. Наша константа C имеет совершенно не случайное значение. Это единица в формате чисел с плавающей запятой (ну или “почти единица”)
Это интересно. Давайте взглянем поближе:

Целочисленное представление C это:

$$C_\sigma = L(B-\sigma) = LB - L\sigma$$

Это почти, но всё же не совсем, форма числа с плавающей запятой. Единственная проблема – мы вычитаем вторую часть выражен должны были бы её прибавлять. Но это можно исправить:

$$LB - L\sigma = LB - L + L - L\sigma = L(B-1) + L(1-\sigma)$$

Вот теперь это выглядит в точности как целочисленное представление числа с плавающей запятой. Чтобы определить, какого име числа, мы вычислим экспоненту и мантиссу, а потом уже и само число C. Вот экспонента:

$$e_{c_\sigma} = (E_{C_\sigma} - B) = (B - 1 - B) = -1$$

А вот мантисса:

$$m_{c_\sigma} = \frac{M_{C_\sigma}}{L} = \frac{L(1-\sigma)}{L} = 1 - \sigma$$

А, значит, значение самого числа будет равно:

$$c_\sigma = (1 + m_{c_\sigma})2^{e_{c_\sigma}} = \frac{1+1-\sigma}{2} = 1 - \frac{\sigma}{2}$$

И правда, если поделить наше σ (а оно было равно 0.0450465) на 2 и отнять результат от единицы, то мы получим уже известное н 0.97747675, то самое, которое “почти 1”. Это позволяет нам посмотреть на C с другой стороны и вычислять его на рантайме:

```
float sigma = 0.0450465;
float c_sigma = 1 - (0.5f * sigma);
int C_sigma = *(*int)&c_sigma;
```

Учтите, что для фиксированного σ все эти числа будут константами и компилятор сможет рассчитать их на этапе компиляции. Результатом будет 0x3f7a3beb, что не совсем 0x3f7a3bea из расчетов выше, но отличается от него всего на 1 бит (наименее значи Получить из этого числа оригинальную константу из кода (и заголовка данной статьи) можно, умножив результат вычислений ещё 1.5.

Со всеми этими выкладками мы приблизились к пониманию того, что в коде в начале статьи нет никакой “магии”, “фокусов”, “интуиции”, “подбора” и прочих грязных хаков, а есть лишь чистая математика, во всей своей первозданной красоте. Для меня гла выводом из этой истории стала новость о том, что преобразование float в int и наоборот путем переинтерпретации одного и того х набора бит – это не ошибка программиста и не “хак”, а вполне себе иногда разумная операция. Слегка, конечно, экзотическая, но очень быстрая и дающая практически полезные результаты. И, мне кажется, для этой операции могут быть найдены и другие применения – будем ждать.

Метки: корень квадратный

Комментарии 71

spmbt 22.08.17 в 13:20

> **0x5f3759df** *(это название статьи)*
Содержание — супер.
… А в плане SEO название статьи — предельно неудачное:) Когда клоны наедятся — стоит переименовать. Может, так и было задумано?

tangro 22.08.17 в 13:43

Это название оригинальной статьи. Ради какого-то там SEO портить авторскую задумку — несправедливо.

Ответить

AndreyNikolin 22.08.17 в 15:06

Должен отметить вы сломали TMFeed

ХАБРАХАБР сегодня в 12:49

1597463007

Блог компании Инфопульс Украина, Занимательные задачки, Ненормальное программирование, Программирование, Спортивное программирование

+26 1.30k 27 2

P.S. Привет из SEnC :)

Ответить

spmbt 22.08.17 в 16:30

Но SEO в данном случае я видел «в благородном смысле» — ради пользования. Читатель ни за что не запомнит название статьи и не введёт в поиск. Значит, возможность SEO для него пропала. Если бы статья называлась "**Быстрый алгоритм квадратного корня и 0x5f3759df**" — не пострадали бы ни поиск, ни задумка автора.

(Или, хуже вариант — «Зачем 0x5f3759df для квадратного корня»)

Ответить

Halt 22.08.17 в 13:50

> Со всеми этими выкладками мы приблизились к пониманию того, что в коде в начале статьи нет никакой “магии”, “фокусов”, “интуиции” “подбора” и прочих грязных хаков, а есть лишь чистая математика, во всей своей первозданной красоте. Для меня главным выводом из з истории стала новость о том, что преобразование float в int и наоборот путем переинтерпретации одного и того же набора бит – это не ошибка программиста и не “хак”, а вполне себе иногда разумная операция.

Ошибаетесь. В этом коде есть чудовищный и катастрофический хак, который просто обязательно надо было упомянуть. А именно, нарушен правила strict aliasing-а при разадресации указателя. Все беды в коде возникают оттого, что кто-то проникся «элегантным решением», не разобравшись во всех деталях, а потом начинает применять его к месту и не к месту.

Эту и другие темы я разбирал в своем докладе на конференции C++ Russia. Кому интересно, можете заглянуть.

Ответить

Halt 22.08.17 в 13:58

Подробный разбор по strict aliasing-у можно найти в тексте другой моей статьи.

Ответить

erlyvideo 23.08.17 в 06:53

а что поделать, тут пользуются аппаратным ускорением логарифмирования

Ответить

Halt 23.08.17 в 07:14

Вопрос не в том, пользуются или не пользуются. Любое трюкачество должно быть уместно. А если его таки применяют, должен быть большой жирный комментарий, почему оно здесь и какие условия нужны для того чтобы все работало, включая фазы луны и прочее.

Код в статье будет работать, если компилятору передали ключ -fno-strict-aliasing, либо если эта опция умолчательная для компилятор

Если через n лет проект начнут портировать на другую архитектуру (или переедут на другой компилятор) и забудут про этот маленьк трюк, последствия могут оказаться самыми печальными.

Ответить

tangro 23.08.17 в 10:28

Чтобы печальных последствий не было, достаточно 1-2 юнит-тестов. Если компилятор вдруг решит поменять размер int или стрюка с алиасингом — это произойдёт на этапе компиляции и легко отловится тестом.

Ответить

Halt 23.08.17 в 11:05

К сожалению, этого *не достаточно*. Программа, содержащая неопределенное поведение — некорректна. *Не существует* никаких критериев, позволяющих доказать корректность программы, кроме формального анализа и ручного разбора ассемблера в *данн конкретном случае*. Даже компилятор сам не знает, есть ли там UB или нет.

Поменяв одно выражение в любой части кода вы можете повлиять на инлайнинг и как следствие на порядок выполнения проход компилятора. Вам не доводилось видеть ситуацию, когда в программе выполняются *обе* ветви условия *одновременно*? Это один возможных вариантов реакции компилятора на UB.

Выполняя оптимизации, компилятор исходит из предположения, что неопределенного поведения в программе нет. Соответстве абсолютно все действия, выполняемые компилятором, заботятся только об обеспечении наблюдаемого поведения. Код из стать таковым не является.

Компилятор может изменять структуру программы так, чтобы сохранить *наблюдаемое* поведение и упростить себе жизнь. Под н могут попадать выражения, условия, и даже целые функции. Представьте, что одним из таких условий был assert в вашем юнит т

Резюмируя, юнит тест в таком случае даст не больше, чем комментарий «мамой клянусь, оно работает!». Даже хуже, проход те может вселить только ложную уверенность в том, что все хорошо.

Ответить

mayorovp 23.08.17 в 11:36

assert в тесте как раз защитить можно — надо только модуль с тестами компилировать без оптимизации, и линковать с остал программой динамически (или статически, но в два этапа — второй этап линковки уже без оптимизации).

Хуже другое — UB может никак не проявляться в тестах, но проявиться уже в реальной программе. Вот тут и правда никак не закрыться.

Ответить

Halt 23.08.17 в 11:46

Идя таким путем, вы вступаете на очень зыбкую почву. Неизвестно еще, что будет хуже.

Если возникает объективная потребность в небезопасных манипуляциях, их нужно изолировать, вплоть до ручного написан функции на ассемблере.

Другое дело, что вся эта машинерия может негативно сказаться на производительности и потерять в итоге смысл.

Современные компиляторы достаточно умные, чтобы использовать весь набор доступных инструкций, включая уже назван rsqrtps. Главное просто не мешать и не считать себя умнее компилятора.

Ответить

mayorovp 23.08.17 в 12:38

Так я потому и пишу: *хуже другое — UB может никак не проявляться в тестах, но проявиться уже в реальной программе. тут и правда никак не закрыться.*

Ответить

Halt 23.08.17 в 13:46

Я видел. Это были мысли вслух, а не наезд :)

Ответить

F0iL 23.08.17 в 12:18

Материалы по ссылкам прочёл, и даже понимание пришло в целом, но не могу понять, что конкретно не так в коде функции в статье? Ти примитивные, массивов нет, компилятор поместит i и x в регистры или на стеке… и что не так? Ради интереса скомпилировал тест с -O3 O0 с включенным и выключенным алиасингом на g++ и clang++ (std=c++14) — результат выполнения одинаков. В чем должен быть подво

Ответить

Halt 23.08.17 в 12:33

> что конкретно не так в коде функции в статье?

Код в статье нарушает стандарт C++ и провоцирует UB. Дальше можно уже не разбираться, ибо на этом месте заканчиваются все гарантии.

> Типы примитивные, массивов нет, компилятор поместит i и x в регистры или на стеке… и что не так?

Что именно компилятор будет делать со значениями зависит от сотни-другой параметров, включая оценку весов при инлайнинге и re pressure в данной точке. Сама попытка объяснить, что сделает компилятор из общих соображений и «здравого смысла» уже обречен провал.

> Ради интереса скомпилировал тест с -O3 и -O0 с включенным и выключенным алиасингом на g++ и clang++ (std=c++14) — результа выполнения одинаков.

То что он одинаков здесь и сейчас не значит, что он будет одинаков всегда. В этом и засада. Пример того, что может случиться, озвучивается в докладе в районе 28 и 29 слайдов.

К сожалению, самое страшное, что может сделать программист в непонятной ситуации, — «это проверить на практике и убедиться, ч все хорошо».

Ответить

F0iL 23.08.17 в 13:13

То что это нарушение стандарта и UB — это с самого начала было понятно, интересно было как раз увидеть реальный пример «что конкретно может пойти не так», для этого, собственно, и пробовал скомпилировать с разными ключами, чтобы получить разные результаты и пройтись потом по отладочному выводу.

Ответить

mayorovp 23.08.17 в 13:16

Так кидали же уже ссылку ниже...

Ответить

Halt 23.08.17 в 13:45

Могу посоветовать две статьи: раз и два.

Ответить

mayorovp 23.08.17 в 12:35

Код основан на трюке с обращением к float через указатель на int: `int i = *(int*)&x;, x = *(float*)&i;`. Вот это и есть "не так".

Подвох в том, что эта операция — по стандарту UB (нарушение strict aliasing rule). *Некоторые* компиляторы имеют ключи для отключен этого правила или вообще не учитывают его при оптимизациях, потому код и работает. Но такой код является непереносимым не тол на другие платформы, но даже на новую версию компилятора.

Ответить

lieff 23.08.17 в 12:36

Подвох в том, что по стандарту это UB. Несмотря на то что указатель скастили вот только что — на указателях этих есть пометка, что гарантированно не пересекаются (когда они гарантированно пересекаются), нарушается правило strict aliasing стандарта. Кстати в ст не указан еще один метод: пометить указатели как могущие пересекаться __attribute__((__may_alias__)), но это уже расширение.

Ответить

Xandrmoro 22.08.17 в 13:51

> Со всеми этими выкладками мы приблизились к пониманию того, что в коде в начале статьи нет никакой “магии”

Скорее, наоборот, я ещё больше укрепился в том, что математика — это магия.

Ответить

windgrace 22.08.17 в 17:20

Третий закон Кларка, однако.

> Any sufficiently advanced technology is indistinguishable from magic

Ответить

Arastas 22.08.17 в 13:52

> нам не хватает лишь уверенности в том, что вычисленное таким образом приближенное значение может быть столь же эффективно улучшено алгоритмом Ньютона

Я тут на прикинул на бумажке. Получается, что для возведения x в степень p, где p не ноль, шаг по Ньютону будет $x = x\left(1 - p + pzx^{-\frac{1}{p}}\right)$

z это аргумент функции (который используется в оригинальном коде для вычисления xhalf). Похоже, что удобно будет только когда -1/p — положительное целое.

Ответить

Browning 22.08.17 в 14:27

В блоке "интересные публикации" название статьи отображается *переведённым в десятичную систему счисления*. Exploitable?

Ответить

tangro 22.08.17 в 16:09

Вот это да. Ну, заодно и потестили Хабр.

Ответить

AngReload 22.08.17 в 16:34

В TMFeed аналогично — «1597463007».

Ответить

michael_vostrikov 22.08.17 в 17:12

Возможно там при рендеринге есть что-то типа:

```
if (is_numeric($value)) echo (0 + $value);
```

Работает на PHP 5.6

Ответить

tangro 22.08.17 в 18:13

Администрация Хабра переименовала статью :)
Ну ок.

Ответить

sebres 22.08.17 в 19:18

Интересно какие ксплойты повылазят, если опубликовать перевод к "0x5f3759df(appendix)"...

```
ReferenceError: appendix is not defined?
```

> Ради какого-то там SEO портить авторскую задумку — несправедливо.

+1
Про заворот ~~слепых~~ кишок у SEO-шников опосля второй статьи я помолчу лучше...

Ответить

andy_p 22.08.17 в 14:28

Хорошо, что на данной платформе sizeof(int) == sizeof(float).

Ответить

x893 22.08.17 в 14:41

Это надо в начале алгоритма поставить как assert, иначе долго будут искать причину гуру программирования когда неправильно считат будет.

Ответить

khim 22.08.17 в 19:55

С современными компиляторами этот код вообще использовать нельзя, нарушения стандарта — они такие, да. Компилятор вполне мо «соптимизировать» весь этот код в «return 0».

Ответить

x893 22.08.17 в 21:04

Что то меня терзают сомнения на тему таких умных компиляторов. Может конечно алгоритмы которые в шахматы выигрывают и скомпилируют в return 0, но те что в шашки — точно так не смогут. Да и отладчик с #pragma останется всегда. На крайняк и на ассемблере можно наколбасить. На нём уж точно замучаются оптимизировать.

Ответить

dkozh 22.08.17 в 22:08

Good news, everyone.

И хотя более новые версии GCC так (пока) не делают, такое поведение формально не противоречит стандарту, потому что в результате undefined behaviour может получиться все, что угодно (по факту оптимизатор решил, что `(unsigned short *)&a` и `&a` алиасятся, и решил переставить инструкции местами, потому что почему бы и нет).

Ответить

vlanko 22.08.17 в 22:46

такое чудо вылазит, начиная с O2
A Clang сразу пишет 4, забив на 5.

Ответить

symbix 23.08.17 в 00:08

Я ненастоящий сварщик, но — volatile проставить и все, не?

Ответить

agmt 23.08.17 в 11:37

Не уверен, станет ли валиднее код, но медленнее — точно.

Ответить

ShinRa 22.08.17 в 16:13

У меня дежавю, кажется что-то похожее я видел в хабе «Разработка игр».

Ответить

gresolio 22.08.17 в 18:47

Упоминалось в переводе «Повышаем производительность кода: сначала думаем о данных».
Также замечено в комментах: раз и два. Но перевода именно этой статьи раньше не было.

Ответить

alsii 22.08.17 в 16:55

О! Еще один элегантный способ выяснить в матрице мы или нет? :-)

Ответить

Iqorek 22.08.17 в 17:22

Есть какие то замеры производительности по сравнению с традиционным способом? С 2005 года и уж с тем более с 80х прошло много лет может быть сегодня овчинка уже не стоит выделки?

Ответить

tangro 22.08.17 в 18:13

Что-то с 2005 года поменялось в теории сложности или архитектуре процессоров, что операции корня и деления стали работать быстре суммирования и умножения? Вряд ли.

Ответить

khim 22.08.17 в 20:01

> Что-то с 2005 года поменялось в теории сложности или архитектуре процессоров

Таки да

Ответить

tangro 22.08.17 в 23:20

Вот это да. И всё же вопрос был о сравнении с «традиционными способами», а вшитая в проц аппроксимация — это не оно.

Ответить

ProLimit 22.08.17 в 23:00

Есть же еще математические сопроцессоры (FPU). Например, в STM32 вычисление квадратного корня занимает 14 тактов, что не так много. Деление — еще 14. Так что если нужен обратный квадратный корень — лучше взять этот алгоритм, если прямой — лучше вычислить на FPU.

Ответить

rPman 23.08.17 в 01:29

а теперь используйте сопроцессор в коде шейдера, например.

Ответить

chersanya 23.08.17 в 01:33

Вот как раз в шейдерах-то таких хаков не надо, они только вредят. GPU обычно существенно быстрее работают с плавающей то они на это заточены.

Ответить

LynXzp 22.08.17 в 17:41

Вау! Прямо новая глава для Hacker's delight.

Ответить

Scf 22.08.17 в 20:06

Шикарно!

Поделюсь трюком, который я вычитал в исходниках Elite для zx spectrum: быстрое умножение двух однобайтных чисел.
`ab = (a^2 + b^2 – (a–b)^2)/2`
Эта формула элементарно выводится из школьного разложения квадрата разности. Т.е. можно быстро и точно умножать однобайтные чис
используя таблицу квадратов (512 байт)

Ответить

khim 22.08.17 в 20:31

Это было реально круто на процессорах Z80 или 6502, на которых отсутствовала операция умножения. Современные же процессоры перемножают числа за 2-3 такта, так что необходимость в подобных трюках, в общем и целом, отпала…

Ответить

mayorovp 23.08.17 в 08:36

Современные процессоры реализуют эту оптимизацию в железе, потому и так мало тактов требуется.

Ответить

netch80 23.08.17 в 15:25

С современными возможностями по количеству вентилей вообще умножение строят на сетках и затем параллельных сумматорах.. что обычно не «эта» оптимизация. Хотя результат ещё лучше :)

Ответить

meta4 22.08.17 в 20:42

Три дереференса + две суммы + одно вычитание + 1 побитовый сдвиг вместо одного умножения? oO

Ответить

ainoneko 23.08.17 в 10:38

Там всегда a не меньше b?
Иначе ещё нужна проверка. (Или я чего-то не понимаю?)

Ответить

Iqorek 22.08.17 в 21:59

> я вычитал в исходниках Elite для zx spectrum

Где где? Можно ссылку на исходники?

Ответить

Scf 22.08.17 в 22:13

Ну это просто. Загружаешь MONS на 58500, запускаешь его. вбиваешь загрузчик кода Elite, он с адреса 5B00, длину блока указываеш так, чтобы MONS не перетереть. Потом переходишь на 7CB7, это точка входа в игру. И жмешь комбинацию клавиш для показа листин Вуаля — на экране исходник.

20 лет прошло, до сих пор помню :-)

Ответить

netch80 23.08.17 в 15:20

Есть более известный вариант:

```
ab = (a+b)^2/4 – (a–b)^2/4
```

только два лукапа по таблице, для a+b и a-b; деление на 4 выполняется нацело, дробные взаимно компенсируются.

Ответить

Scf 23.08.17 в 15:39

Тогда a+b может быть больше 255, т.е. нужно оперировать словами и таблицу квадратов удваивать по размеру.

Ответить

shukan 22.08.17 в 21:46

Кто бы еще смог понять, что здесь происходит —
▸ Эвклид без Эвклида ?

Ответить

chersanya 23.08.17 в 01:07

Это не алгоритм Евклида, а существенно более оптимальный, применимый для такого частного случая — получается сложность O(log k) вместо O(k) для Евклида. Переход можно обосновать так: если `y = x^-1 mod 2^k`, то `xy = a 2^k + 1`; домножим, как в вашей функции, н xy: получим `xy * (2 - xy) = (a 2^k + 1) * (2 - a 2^k - 1) = -a^2 2^(2k) + a 2^k - a 2^k + 1 = 1 mod 2^(2k)`, то есть из обратн x значения по модулю 2^k получили обратное по модулю 2^(2k). Делая такие переходы как раз из обратного по модулю 2^3 получим по 2 т.д. Также отсюда ясно, почему для перехода к 64 битам достаточно одного шага.

Ответить

chersanya 23.08.17 в 09:22

А, про первые 3 бита забыл написать. Тут можно очень просто убедиться, перебрав все нечётные числа от 1 до 7 (т.к. рассматриваем обратное по модулю `2^3 = 8`) — как видно, каждое число является обратным самому себе. Ну и ещё — к чётным числам обратного ясно дело не существует, поэтому их вообще не рассматриваем.

Ответить

vlanko 22.08.17 в 22:37

А в C++ в те времена int уже был 32-битным? Или это не имеет значения для указателей?

Ответить

tangro 22.08.17 в 23:28

В какие «те»? Quake III Arena, о котором идёт речь в статье, вышел в 1999 году, а int стал 32-битным (сначала это было даже опциональн зависимости от модели памяти) с распространением 32-битных операционок, т.е. ещё на пару лет раньше.

Ответить

janatem 22.08.17 в 22:58

Только для денормализованных значений аргумента (близких к нулю) в результате, кажется, будет мусор, а не ожидаемое +inf.

Ответить

vlanko 22.08.17 в 23:32

да, формула только для x>1

Ответить

askv 22.08.17 в 23:31

А побитовое извлечение корня разве не быстрее? Там только целочисленные операции, без операций с плавающей точкой. Кто-нибудь сравнивал?

Ответить

mayorovp 23.08.17 в 08:39

Как вы будете извлекать квадратный корень из числа с плавающей точкой без операций с плавающей точкой?

Ответить

Keroro 23.08.17 в 06:26

Для вычисления целочисленных корней на 8 бит PIC микроконтроллере давно использую Hardware algorithm [GLS] из hackersdelight. Очень компактный и шустрый.

Ответить

askv 23.08.17 в 06:37

Есть ещё целочисленный алгоритм, там только побитовые сдвиги и сравнения, нет умножений. Когда-то писал на ассемблере, но листо затерялся, не могу найти. Но можно попробовать воспроизвести.

Ответить

Aivendil 23.08.17 в 10:28

Огромное спасибо! Ради таких статей я и пришёл в свое время на этот ресурс.

Ответить

Написать комментарий

Предпросмотр Отправить Markdo

САМОЕ ЧИТАЕМОЕ

Сутки | Неделя | Месяц

- «Магическая константа» 0x5f3759df
  +188 23k 320 71
- Не виноватая я. Он сам пришел
  +271 51,5k 76 383
- Как Яндекс научил искусственный интеллект понимать смысл документов
  +110 18,1k 108 133
- Шесть мифов о блокчейне и Биткойне, или Почему это не такая уж эффективная технология
  +154 43,6k 228 223
- Путешествие за бугор и обратно: как не надо устраиваться работать за рубежом
  +392 98,5k 207 971

ИНТЕРЕСНЫЕ ПУБЛИКАЦИИ

- Погружение в разработку на Ethereum. Часть 1
  +8 825 14 0
- Заполняем «Соглашения, налоги и банковскую информацию» в iTunes connect для русского ООО
  +13 525 11 0
- Защита сайта от атак с использованием WAF: от сигнатур до искусственного интеллекта
  +19 683 4 0
- Академия Veeam — практические классы для начинающих C# разработчиков
  +9 773 8 2
- С ветерком! Как мы внедряли бесконтактную оплату поездок в метро
  +15 1,9k 11 26