L1VM Primenum Benchmark 02

Stefan Pietzonke 2020/05/07 – spietzonke@gmail.com

Hier ist ein Benchmark für C und die L1VM.
Es werden die Primzahlen bis 99999989 gesucht und in der Shell ausgegeben.
Ich habe für C den Sourcecode von „Rosettacode Primes C“ [1] genommen.

Für L1VM Brackets schrieb ich das Programm selbst.
Die Benchmarks compiliert man mit dem "make.sh" Script.
Das C Programm ist wie zu erwarten um den Faktor 7 schneller als die L1VM.
Wenn man bei Linux die Ausgabe nach ">/dev/null" leitet.
Die L1VM liegt IMHO damit immer noch im effizienten Bereich!

Ich nahm noch Python mit zu dem Benchmark dazu.

Hier ist das Ergebnis:

C: (clang -O2)

$ time ./primes-rosetta >/dev/null

real 0m3,787s
user 0m3,601s
sys 0m0,184s

L1VM:

$ time vm/l1vm-cli prog/primes-4-long >/dev/null

real 0m27,663s
user 0m27,538s
sys 0m0,112s

Python 3.8:

$ time python3.8 primes.py >/dev/null

real 0m31,937s
user 0m31,597s
sys 0m0,328s

Die Faktoren zu C:
C primes-rosetta: 1,0
L1VM: 7,3
Python 3.8: 8,4

Die Programme, die verwendet wurden sind hier:

C:

```
// taken from https://rosettacode.org/wiki/Sieve_of_Eratosthenes#C

#include <stdio.h>
#include <malloc.h>
void sieve(int *, int);

int main(int argc, char **argv)
{
     int *array, n = 100000000;
     array =(int *)malloc((n + 1) * sizeof(int));
     sieve(array,n);
     free (array);
     return 0;
}

void sieve(int *a, int n)
{
     int i=0, j=0;

     for(i=2; i<=n; i++) {
          a[i] = 1;
     }

     for(i=2; i<=n; i++) {
          // printf("\ni:%d", i);
          if(a[i] == 1) {
               for(j=i; (i*j)<=n; j++) {
                    // printf ("\nj:%d", j);
                    // printf("\nBefore a[%d*%d]: %d", i, j, a[i*j]);
                    a[(i*j)] = 0;
                    // printf("\nAfter a[%d*%d]: %d", i, j, a[i*j]);
               }
          }
     }

     printf("\nPrimes numbers from 1 to %d are : ", n);
     for(i=2; i<=n; i++) {
          if(a[i] == 1)
               printf("%d\n", i);
     }
     printf("\n\n");
}
```

L1VM Brackets:

```
// primes-4.l1com - Sieve of Eratosthenes
// primenum search
(main func)
     (set int64 1 zero 0)
     (set byte 100000000 primes)
     (set int64 1 limit 100000000)
     //(set int64 1 limit 100)
     (set int64 1 one 1)
     (set int64 1 two 2)
     (set int64 1 i 2)
     (set int64 1 j 0)
     (set int64 1 z 1)
     (set int64 1 f 0)
     (set int64 1 a 0)
     (set int64 1 k 0)
     (set int64 1 n 0)
     // array primes is filled by zeroes
     // search primes
     (optimize-if)
     (two i =)
     (:search_primes)
     ((i i *) n =)
     (((n limit <) f =) f if)
          (primes [ i ] a =)
          (((a zero ==) f =) f if)
               (i j =)
               (:search)
               ((i j *) k =)
               (((k limit <) f =) f if)
                    (one primes [ k ] =)
                    ((j one +) j =)
                    (:search jmp)
               (endif)
          (endif)
          ((i one +) i =)
          (:search_primes jmp)
     (endif)
     // print primes
     //
     (two i =)
     (:print_primes)
     (((i limit <) f =) f if)
          (primes [ i ] a =)
          (((a zero ==) f =) f if)
               (4 i 0 0 intr0)
               (7 0 0 0 intr0)
          (endif)
          ((i one +) i =)
          (:print_primes jmp)
     (endif)
     (255 0 0 0 intr0)
(funcend)
```

Python 3.8:

```
# Python program to print all primes smaller than or equal to
# n using Sieve of Eratosthenes

def SieveOfEratosthenes(n):

    # Create a boolean array "prime[0..n]" and initialize
    #  all entries it as true. A value in prime[i] will
    # finally be false if i is Not a prime, else true.
    prime = [True for i in range(n+1)]
    p = 2
    while (p * p <= n):

        # If prime[p] is not changed, then it is a prime
        if (prime[p] == True):

            # Update all multiples of p
            for i in range(p * 2, n+1, p):
                prime[i] = False
        p += 1

    # Print all prime numbers
    for p in range(2, n):
        if prime[p]:
           print (p)

# driver program
SieveOfEratosthenes(100000000)
```

Das C Programm wurde mit Clang mit „-O2“ Optimierung compiliert.
Die L1VM ohne SDL Bibliotheken und auch mit Clang „-O2“ Optimierung erstellt.

Für Python wurde die 3.8 Version zum Lauf verwendet.
Die L1VM liegt fast mit Python gleich. Wobei der Python Code vielleicht noch optimiert werden könnte.

Hier ist der zweite Teil über meinen L1VM Primzahlen Benchmark.
Die Webseite ist bei [2]. Dort sind auch die Programme in einem Archiv enthalten. [3]

Hier gehe ich genauer auf den L1VM Benchmark ein.
Wenn in der L1VM in „include/global.h“ die Überprüfung von Arraygrenzen ausgeschaltet ist, dann ist der Faktor zu C nicht 7 sondern nur noch etwa 5!

„Einige Leute argumentieren, dass Effizienz bei Interpretern keine Rolle spielt, da sie sowieso langsam sind. Diese Haltung führt dazu einen Interpreter zu haben der mehr als um den Faktor 1000 bei vielen Programmen langsamer ist als ein nativ compiliertes Programm das mit einem optimierenden Compiler erstellt wurde.
Während die Verlangsamung bei einem effizientem Interpreter nur bei etwa Faktor 10 liegt.“ [4]

Optimierung am Assembly Code

Ich wollte wissen wie sich das L1VM Primzahlenprogramm noch weiter beschleunigen lässt. Um es kurz zu machen: ja das geht, wenn man den Assembly Code von Hand optimiert.

Die innere Schleife in der die Primzahlen in das Array gespeichert werden ist der ideale Platz dafür:

```
30|(:search)
31|  ((i j *) k =)
32|  (((k limit <) f =) f if)
33|  (one primes [ k ] =)
34|  ((j one +) j =)
35|  (:search jmp)
36|  (endif)
```

Diese Schleife wird mehrere Tausend mal durchlaufen.
Hier ist der Assembly Code:

```
:search
loada j, 0, 11
muli 1, 11, 12
load k, 0, 13           //
pullqw 12, 13, 0        // fällt weg
loada k, 0, 14          //
lsi 14, 5, 15
jmpi 15, :if_2
jmp :endif_2
```

Wie man sieht, kann man die Variable „k“ reduzieren und das Ergebnis der Multiplikation direkt bei „lsi“ einfügen:

```
lsi, 12, 5, 15
```

Hier ist der komplette optimierte Assembly Code:

```
:search
loada j, 0, 11
muli 1, 11, 12
lsi 12, 5, 15
jmpi 15, :if_2
jmp :endif_2
:if_2
```

```
Hier sind die Laufzeiten des optimierten Programms:

$ time vm/l1vm-nojit prog/primes-4-long-opt >/dev/null

real 0m19,682s
user 0m19,619s
sys  0m0,056s
```

Faktor zu C: 5,20

Ohne Arrayindex Überprüfung:

```
$ time vm/l1vm-cli prog/primes-4-long-opt >/dev/null

real 0m16,623s
user 0m16,592s
sys  0m0,028s
```

Faktor zu C: 4,39

Wie man sehen kann ist der Unterschied zu nicht optimiertem Code groß.
Es kann sich also lohnen sich den Assembly Code anzusehen.
Der Code der vom Compiler erstellt wurde ist gut, aber noch nicht so hoch optimiert.

[1] https://rosettacode.org/wiki/Sieve_of_Eratosthenes#C
[2] http://www.midnight-koder.net/blog/software/l1vm/2020/05/03/L1VM-primenum.html
[3] http://midnight-koder.net/blog/assets/l1vm/primes-benchm-02.zip
[4] https://www.jilp.orhg/vol5/v5paper12.pdf M. Anton Ertl & David Gregg: „The Structure and Performance of Efficient Interpreters“