

CACHE OPTIMIZED SIEVE

ANTAL JÁRAI AND EMIL VATAI

ABSTRACT. We present a new version of the sieve of Eratosthenes, which we expect to be faster by a constant factor than the method of Tomás Oliveira e Silva [1]. This is achieved via three methods of sieving with small, medium and large primes. The greatest boost comes from the use of cyclic data structures represented as linear arrays, allowing the efficient sieving with large primes, by sorting these large primes using an algorithm similar to bucket sort and keeping them sorted by strictly linear accesses. Technical details about the implementation of the sieve with small and medium primes are also presented, and so are some ideas about how to split the work among more processors.

The method described is somewhat similar to the one used by Tomás Oliveira e Silva and in some implementations of the GNFS factoring algorithm [2] combining sieving with bucket sort. Our algorithm was developed independently and is specific in using linear access for rather large chunks. It will be used for verifying the Golbach conjecture for even numbers in the vicinity of 10^{18} .

1. INTRODUCTION

In this paper we present a new version of the sieve of Eratosthenes, which should improve performance over currently available implementations. The idea behind it is to use cyclic data structures represented as linear arrays, allowing the efficient sieving with large primes, by semi-sorting these large primes using an algorithm similar to bucket sort and keeping them semi-sorted by strictly linear accesses. The idea to combine sieving with bucket sort is known, although we found it independently: similar attempts were

Received by the editors: September 30, 2010.

2010 *Mathematics Subject Classification.* 11-04, 11Y11, 68W99.

1998 *CR Categories and Descriptors.* F.2.1 [**ANALYSIS OF ALGORITHMS AND PROBLEM COMPLEXITY**]: Numerical Algorithms and Problems – *Number-theoretic computations*; E.1 [**DATA STRUCTURES**]: Lists, stacks, and queues.

Key words and phrases. sieve, cache memory, number theory, primality.

made by Tomás Oliveira e Silva [1] and Jens Franke and Thorsten Kleinjung [2]. Remaining parts seem to be new.

We implemented the sieve algorithm for the *AMD64 architecture*¹, using 64 bit general purpose registers and 128 bit *SSE* registers and illustrate the ideas with this implementation. To understand the advances of our method we suggest the reader to think about how smoothly our method can be implemented on *Cell Broadband Engine (CBE)*² processor in a way that all 8 SPU's work on the same sieving; on this processor SPU's have 256 kB local store and they can access the main store only by DMA.

Suppose the cache size is 2^s bits (we do not specify the actual cache size, for s is a parameter of the program), then we can divide all primes into:

Small primes: : $p < 64$,

Medium primes: : $64 < p < 2^s$,

Large primes: : $p > 2^s$.

If memory access wouldn't matter, i. e. if sieving one cell (bit) of the sieve table with a prime would take a constant amount of time, then sieving with small primes would take up roughly half of the run-time, sieving with medium primes 1/4 and with large primes the remaining 1/4. But things aren't that simple: memory access can tremendously slow down the sieving process, because the (naive) algorithm doesn't access memory in a sequential way, and that why the caching mechanism of the CPU is overloaded, and accessing memory not in cache can take up to hundreds of clock cycles [3]. This is the reason why in real life, the slow part of a sieving algorithm is the sieving with the large primes because they have to "skip" large parts in memory, larger then the cache size. To optimize performance, we use specialized strategies for small, medium and large primes, which will be described in the following section.

2. SMALL PRIMES

For small primes we decided to use 64 bit and 128 bit long bit masks stored in registers and the CPU's **shift**, **and** and **or** operations.

¹AMDTM is a trademark of Advanced Micro Devices, Inc.

²Cell Broadband EngineTM is a trademark of Sony Computer Entertainment Incorporated.

The small primes are:

$$P_s = \{33, 35, 13, 17, 19, 23, 29, 31, 37, 41, 43, 47, 53, 59, 61\}.$$

Notice that 2, 3, 5, 7, 11 are not in P_s and 33 and 35 are composite numbers.

2.1. Representation of the sieve table. The number 2 is excluded, because we sieve only odd numbers, that is if the i -th bit of the sieve table, denoted by $S[i]$, is set to 0, it means that $2i + 1$ is prime and if it is set to 1, then $2i + 1$ is composite.

2.1.1. Combined small primes. 5 and 7 are multiplied, so we have $35 = 5 \times 7$ (and similarly 33 is the product of 3 and 11). Now we can store the bit mask for 5 and 7 in a single register and shift it as if sieving with the “prime” 35.

2.2. Sieving with small primes. The bit masks of the small primes are stored in a C array as 64 bit unsigned integers. When sieving with these masks, they are loaded in 9 *XMM* registers and 6 general purpose registers. 4 *XMM* and 4 general purpose registers are used to combine the masks with the `orpd` instruction (`orpd` for *XMM* registers). These combined masks are then copied to the main memory: first an *XMM* register is moved (`movupd`), then two general purpose registers are `or`-ed. Finally the new masks are created: the mask for the prime p is copied to an auxiliary register; shifted right $64 \bmod p$ bits ($128 \bmod p$ for *XMM* registers) in the original register; shifted right $p - (64 \bmod p)$ bits ($p - (128 \bmod p)$ for *XMM*) in the auxiliary register; and then recombined with `or`.

2.3. Timing. In one iteration $512 = 2^9 = 4 \times 128$ bits are sieved. For each *XMM* register, we sieve 2 general purpose registers: one for the first half of the *XMM* register and another for the second half.

We iterate until a segment of 2^s bits is not processed. Then old masks in the memory are replaced with the new, shifted masks.

3. MEDIUM PRIMES

$$P_m = \{p \in \mathbb{Z} : p \text{ prime and } 64 < p < 2^s\}$$

The sieving of medium primes is not that difficult, although some ideas explained later can be applied to speed up this part of the algorithm.

Medium (as well as large) primes are stored in an array of prime-offset pairs (denoted by (p, q)). For medium primes the offset stores the index of the first bit to be marked as composite, counting from the beginning of the current segment.

Some simple improvements to implementation: because the bit (re)setting operations take a bit base, and an index as operands, we start by subtracting 2^s from q and setting the bit base to be the end of the segment, thus eliminating the need the index (i. e. iterator) to be compared with 2^s , because the iteration ends when the iterator becomes non-negative.

4. LARGE PRIMES

$$P_l = \{p \in \mathbb{Z} : p \text{ prime and } 2^s < p\}$$

Large primes are problematic, because they “skip” large parts of memory, and not in a trivially predictable manner. The improvement in our implementation is based on the following simple observation:

Lemma: If the prime p , which satisfies the condition $k2^s \leq p < (k+1)2^s$, sets a bit in the i -th segment, denoted by s_i , then the next segment the algorithm will access when sieving with p will be segment s_{i+k} or s_{i+k+1} .

The idea is, for each k , to store all the primes that satisfy the condition $k2^s \leq p < (k+1)2^s$, in a circle denoted by c_k . The circle c_k is divided into $k+1$ number of “buckets” (b_l^k for $l = 0, \dots, k$) and every circle has a “current bucket” (we will denote its index with c , that is b_c^k is the current bucket of c_k). The *current bucket* contains (p, q) pairs, such that q is the offset for p in the *current segment*. For $j = 1, \dots, k$ buckets $b_{(c+j) \bmod (k+1)}^k$ contain (p, q) pairs, such that p will sieve j segments after the current segment and q is $j2^s$ less than the offset for p in the current segment, thus after j segments, when $b_{(c+j) \bmod (k+1)}^k$ will be the current bucket, each q in it will have right offset.

After sieving, the new offset q is calculated $(\bmod 2^s)$ and put back in the current bucket or in the previous one, depending on whether it will skip $k+1$ or k segments. The next bucket $\bmod k+1$ becomes the new *current bucket* ($c \leftarrow [(c+1) \bmod (k+1)]$) and the process can be repeated.

Summary: to process all the large primes, the algorithm needs to process the current buckets of all the circles and set the current bucket to be the next one $(\bmod k+1)$ for the circle c_k .

4.1. Representation of circles and buckets. Large primes are also stored as prime-offset pairs in the same array where medium primes are stored. Circles and buckets are also stored as C arrays. Buckets are simply pointers to prime-offset pairs defining the beginning of the bucket, circles contain pointers to prime-offset pairs defining the end (in our implementation) of each circle, the index of the current bucket and the index of the *broken bucket*.

4.1.1. Broken buckets. A bucket is broken bucket if the pointer to the beginning of the bucket is above or equal to the pointer to the ending of the bucket. In the case of sieving with the broken bucket, a different procedure is needed, because we have to process the primes in two separated intervals: from the beginning of the bucket to the end of the circle and from the beginning of the circle to the end of the bucket. The index of the broken bucket is vital, and cannot be calculated: for example, it might happen that all the primes in a circle end up in the same bucket. In this situation all the pointers representing the beginning of the buckets would point to the same address and each bucket could be the non-empty, i. e. the broken one, while the others are empty.

4.2. Implementation. The core of the loop of the algorithm for processing large primes in one bucket, described above, is implemented in 15 assembly instructions, executed in 5 clock cycles on today's architectures: We keep records of the addresses of the first and the last unprocessed prime-offset pair in the registers denoted by *beginning* and *end*, and the pairs at these addresses are also stored in registers. One pair is used for sieving, then its new offset is calculated and

- if k segments will be skipped: $k2^s$ is subtracted from the new offset, the pair is saved at *beginning* and *beginning* is incremented.
- if $k + 1$ segments will be skipped: $(k + 1)2^s$ is subtracted from the new offset, the pair is saved at *end* and *end* is decremented.

We repeat the sieving until *end* points to a location above *beginning*. This is implemented without branching, using only conditional move (`cmovXY`) instructions, so its execution time is constant, depending only on the speed of memory access, which can be eliminated with interleaved processing of the primes. This is necessary anyway, because places at *beginning* and *end* both have to be empty during the processing of a prime.

4.2.1. *Interleaved processing.* Because accessing a prime-offset pair from the secondary cache take at 5-6 clock cycles, the sieving is interleaved: while sieving is done with a pair, the next pair is loaded from the *beginning* or *end*, depending where the previous pair have been written.

4.2.2. *Loop unrolling.* Further speed can be gained by “unrolling” the loop processing the primes: if the difference between *beginning* and *end* is $d = q \cdot 2^t + r$ (for some fixed integer t), both q and r can be calculated efficiently with bit twiddling, and the loop can be split up into q batches of executing the core 2^t times without any branching, and then executing it r times one by one.

4.2.3. *Broken buckets improvement.* The loop unrolling described above cannot be applied directly to the broken buckets, it is hard to predict exactly when and which of the two pointers is going to “skip” from the beginning to the end or the other way around. We can only tell that it won’t skip more than the minimum of differences between the beginning of the bucket and the end of the circle, or the beginning of the circle and the end of the bucket. This can be repeated, because only about half of the primes will be moved to the other end of the bucket the other side will stay in place, so neither can the differences, mentioned above, be predicted, thus the unrolling should be repeated as long as it is possible.

5. MEDIUM PRIMES REVISITED

The medium primes behave well until they reach the vicinity of $\approx 2^{s-1}$. If a medium prime p satisfies $\frac{2^s}{t+1} < p < \frac{2^s}{t}$, it will be iterated t or $t+1$ times for one segment. If t is small (e.g. $t < 16$), the branch miss prediction penalty can cost more than the actual work done by the sieve.

Similarly, as we have organized the large primes in buckets, we can group the “larger” medium primes in groups for the first few values of t , and have an entirely “unrolled” loop for each of them, without branching, just conditional move operations, eliminating the confusion of the branch prediction mechanism.

For each t , the larger medium primes are grouped, so that $\frac{2^s}{t+1} < p < \frac{2^s}{t}$, and every group has a pointer dividing it into the “lower” and “upper” primes. Lower primes need to be iterated t times and the upper primes $t+1$ times.

Because there aren't too many of these medium primes, they do not have to be sorted in-place: each lower prime from is processed, and copied to the beginning or the end of a different array, and after that the upper primes are processed in a similar manner.

6. CONCLUSION AND FUTURE WORK

We implemented the sieve of Eratosthenes, but our approach can be applied to other, similar algorithms e.g. to the Quadratic Sieve [4] factoring algorithm and probably in some form to the General Number Field Sieve. The ideas behind our algorithm make it also possible to implement sieving algorithms on architectures like *CBE* which seemingly are not suited for sieving at all.

Our further work will include the fine-tuning of the algorithm and its parameters, based on measurements yet to be taken; implementing these ideas in some of the sieving algorithms used for factoring; possibly an implementation on the *CBE* architecture.

6.1. Parallel execution. Another important aspect is enabling the algorithm to run on multiple cores all working on the same sieving. Sieves of different primes are independent from each other, e.g. sieving with small primes can be done on one core, with medium primes on the other and sieving with different circles can be done on different cores. But there is a dependence within one prime (or one group of primes, for example a circle): the offset to be correct for a segment, the sieving for that particular prime (or group of primes) has to be done for all the segments before it, hence we need scheduling. Therefore, the procedure of sieving with circles can be executed separately for each circle and sieving with medium primes can be executed partially, by defining the interval of primes we want to sieve with.

7. ACKNOWLEDGMENT

The Project is supported by the European Union and co-financed by the European Social Fund (grant agreement no. TAMOP 4.2.1/B-09/1/KMR-2010-0003).

REFERENCES

- [1] T. Oliveira e Silva, *Goldbach conjecture verification*
<http://www.ieeta.pt/~tos/goldbach.html>
- [2] J. Franke, T. Kleinjung, *Continued fractions and Lattice sieving. Proceedings SHARCS 2005*,
<http://www.ruhr-uni-bochum.de/itsc/tanja/SHARCS/talks/FrankeKleinjung.pdf>
- [3] J.L. Hennessy, D.A. Patterson, *Computer architecture: a quantitative approach*, Morgan Kaufmann, 3rd edition, 2002.
- [4] D.M. Bressoud, *Factorization and Primality Testing*, Springer-Verlag, 1989.

EÖTVÖS LORÁND UNIVERSITY, FACULTY OF INFORMATICS, DEPARTMENT OF COMPUTER ALGEBRA, PÁZMÁNY PÉTER SÉTÁNY 1/C, 1117 BUDAPEST, HUNGARY
E-mail address: ajarai@moon.inf.elte.hu, emil.vatai@gmail.com