SIEVE OF ATKIN REVISITED

Mircea GHIDARCEA[1] and Decebal POPESCU[3]

The ***Sieve of Atkin (SoA)*** *represented a significant advancement in the domain of prime number identification, promising substantial efficiency gains over its predecessors – yet, despite its theoretical advantages, practical enhancements to SoA have seen limited exploration. This paper presents a set of comprehensive enhancements to SoA, achieving an order-of-magnitude improvement in its performance. Our approach introduces a set of algorithmic refinements aimed at optimizing its core computational processes, carefully designed to leverage the algorithm's inherent strengths while addressing known limitations. Subsequently, we implement a parallel, incremental and cache-intensive sieve as proof-of-concept for this new SoA algorithm.*

Our empirical analysis demonstrates through benchmarking that these enhancements not only can revitalize SoA but also position it within the same performance echelon as the most efficient sieves currently available. These findings underscore the untapped potential of SoA in prime number sieving, opening new avenues for research and application in mathematical computing and beyond.

Keywords: Prime numbers sieving, Sieve of Atkin, Algorithms, Algorithm optimization, Parallel algorithms

1. **Introduction**

Although is the last important prime sieving algorithm, **Sieve of Atkin** is quite old: it was officially published in 2003 [1], but the preprint for the article dates from 1999, as seen in References section of [2]. Nevertheless, extremely few serious attempts were made to improve upon this promising algorithm, as **systematic generation of prime numbers in sequence** (aka *prime sieving*) was practically ignored lately. No real progress was made to advance prime numbers sieving algorithms in the last decades [3] – nowadays, the only sieving algorithm that receives any attention beyond didactic landscape is the **Sieve of Eratosthenes (SoE)**.

[1]Doctoral student, University POLITEHNICA of Bucharest – Computer Science, Romania; e-mail: `mircea.ghidarcea@stud.acs.upb.ro`

[2]Professor, University POLITEHNICA of Bucharest – Computer Science, Romania; e-mail: `decebal.popescu@upb.ro`

In our systematic study of the prime sieving domain [3] we could identify only two papers trying to improve upon SoA: Galway in 2000 [2] and Farach/Tsai in 2015 [4]. Galway's contribution is interesting, but it does not address the binary operational complexity of the algorithm; Farach/Tsai's work is even more abstract – it introduces an improved theoretical algorithm that may lower the complexity of SoA, but with no practical discussions and no follow-up.

Given that SoA stands alongside SoE as the only algorithms of practical significance in prime number sieving, as shown in [3], we think there is much potential to unearth in SoA based algorithms. To prove this, our article takes over from Atkin/Bernstein and carry their work several steps further, in two stages:

1) We apply some refinements to SoA and show that impressive speed-up is still possible even at the level of the single threaded reference implementation.

2) We implement a parallel, incremental, cache-intensive sieve as a proof-of-concept and show that its performance really is of the same order of magnitude with as best modern prime sieves.

This study is primarily addressed to researchers and practitioners in the fields of number theory, cryptography, computational mathematics, parallel computing and software optimization. Our improvements on SoA show that significant advancements in prime number sieving efficiency is still possible, potentially impacting various applications from cryptography to computational research methodologies. By targeting these key areas, we aim to contribute to the ongoing development and optimization of mathematical computational techniques.

NOTE 1*: Our experiments used C/C++ to create succinct, self-contained and high-performance code that can be readily compiled across several different platforms. Accompanying code used in this article can be found on* **GitHub** *at* `https://github.com/mirceag70/SoA_Revisited`.

NOTE 2*: All the timings for this paper were measured on a workstation with AMD Ryzen 9 7900X. For a better standardisation of results, all the components of the generation process must be included in the timings to be compared, including any data preparation that occurs before sieving (like root primes generation) or after sieving (like really, effectively obtaining the value of all prime numbers and getting those values in order).* [1]

[1]An optimizing compiler will discard futile code, so very often it is not sufficient to simply compute the number in code without using it – one must assure that the number is really computed in the benchmarking process.

2. **Original SoA**

Sieve of Atkin [1] is a quadratic sieve, based on the mathematical properties of some quadratic forms. This sieve has the lowest known theoretical complexity (similar to Sieve of Pritchard (SoP) [5], but with significantly lower overhead) and thus it is quite promising.

The algorithm uses the fact that all the numbers $p = 12k + r$ that:

a) have an odd number of positive solutions at these equations:

- $p = 4x^2 + y^2$, where r=1 or 5, x and $y > 0$;
- $p = 3x^2 + y^2$, where r=7, x and $y > 0$;
- $p = 3x^2 - y^2$, where r=11, $x > y > 0$;

and

b) are squarefree (not a multiple of the square of a prime);

are primes (in cited paper [1] one can find more about the mathematical reasoning behind SoA). The structure of SoA is more related to Sieve of Sundaram [3] than SoE and has great potential for parallelization (the sieve was published from the start in the segmented form). In the accompanying code you can find three basic implementation for this sieve: a very simple, naive version, a straightforward, standard implementation and a lightly optimized one, implementing the $6k \pm 1$ pattern and bit compression. The multiple *modulo* operations impair the performance, as you can see in table 1 where we added also results for Singleton's **357** [6, 3] as a baseline. We must note that we tried several tricks in the code to speed up the computations, for example by substituting multiplications with additions etc. – still, the additive versions does not work faster than the normal one because the optimizing compiler is better at optimizing straightforward code (the optimizations will generally be executed better by the compiler if the code is as clear as possible), thus we kept using the straightforward code for clarity.

Tbl 1. SoA timings (ms) for generating primes up to 10^n – basic versions

n	$\pi(10^n)$	SoA			357
		Naive	Standard	1bit $6k \pm 1$	
5	9'592	1	1	1	-
6	78'498	2	1	1	1
7	664'579	22	16	14	9
8	5'761'455	642	456	148	86
9	50'847'534	-	6'043	2'415	849
10	455'052'511	-	-	53'842	7'648

The space complexity of the basic sieve is $O(N)$ – in fact the memory required is exactly N for the uncompressed version or $N/24$ in the one implementing the compressed pattern.

As mentioned, the original technique is somewhat improved for very large limits by Galway in [2], or Farach/Tsai in [4], but still there remain a huge number of complex arithmetic operations that have to be executed. It becomes clear that we need to eliminate these complex operations in order to get a practical sieve.

3. Optimized SoA

3.1. Avoiding futile work

Here is the code for one of the sieving loops (as said before, the optimizing compiler will deal with all those multiplications, this is not the issue):

```
//straighforward code
for (x = 1; 4 * x * x < limit; x++)
  for (y = 1; ; y++)
  {
      n = (4 * x * x) + (y * y);
      if (n > limit) break;
      if (n % 12 == 1 || n % 12 == 5)
          FlipBit(n);
  }

//additive version
for (x = 1; x * x < (limit / 4); x++)
  for (y = 1, n = 4*x*x + 1; n <= limit; n += 2*y + 1, y++)
      if (n % 12 == 1 || n % 12 == 5)
          FlipBit(n);
```

We can see two main problems here:

- The loop generates far more values than those that are really useful;
- The validation of a value implies *modulo* operations, which is one of the most taxing operations in terms of CPU resources.

We should try to generate from the start only the values that are already valid, thus avoiding both problems. For this we use the observation that any quadratic form Q has the following properties:

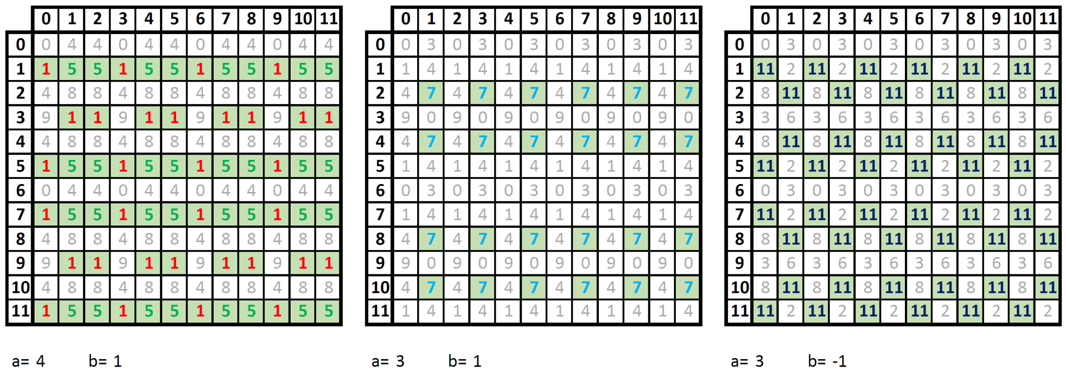
$$Q(a,b) = a \cdot x^2 + b \cdot y^2 \ \mid x = 12k_x + dx;\ y = 12k_y + dy$$
$$Q(a,b) \equiv (a \cdot dx^2 + b \cdot dy^2)(mod\, 12) \tag{1}$$

At this moment a possible solution becomes clear: for each desired remainder identify and use all the eligible tuples

$$(dx, dy) \mid dx, dy < 12 \cap Q(a,b) \equiv r\,(mod\ 12) \mid r \in \{1,5,7,11\}$$

and use only those in the sieve. A simple program will enumerate these values – see figure 1, where we can spot some beautiful symmetries.

From all 144 possible values for each r we are left now with only 32 tuples for $r = 1$, respectively $r = 5$. Similar, only 24 for $r = 7$ and 48 for $r = 11$ –



	0	1	2	3	4	5	6	7	8	9	10	11
0	0	4	4	0	4	4	0	4	4	0	4	4
1	1	5	5	1	5	5	1	5	5	1	5	5
2	4	8	8	4	8	8	4	8	8	4	8	8
3	9	1	1	9	1	1	9	1	1	9	1	1
4	4	8	8	4	8	8	4	8	8	4	8	8
5	1	5	5	1	5	5	1	5	5	1	5	5
6	0	4	4	0	4	4	0	4	4	0	4	4
7	1	5	5	1	5	5	1	5	5	1	5	5
8	4	8	8	4	8	8	4	8	8	4	8	8
9	9	1	1	9	1	1	9	1	1	9	1	1
10	4	8	8	4	8	8	4	8	8	4	8	8
11	1	5	5	1	5	5	1	5	5	1	5	5

a= 4 b= 1

	0	1	2	3	4	5	6	7	8	9	10	11
0	0	3	0	3	0	3	0	3	0	3	0	3
1	1	4	1	4	1	4	1	4	1	4	1	4
2	4	7	4	7	4	7	4	7	4	7	4	7
3	9	0	9	0	9	0	9	0	9	0	9	0
4	4	7	4	7	4	7	4	7	4	7	4	7
5	1	4	1	4	1	4	1	4	1	4	1	4
6	0	3	0	3	0	3	0	3	0	3	0	3
7	1	4	1	4	1	4	1	4	1	4	1	4
8	4	7	4	7	4	7	4	7	4	7	4	7
9	9	0	9	0	9	0	9	0	9	0	9	0
10	4	7	4	7	4	7	4	7	4	7	4	7
11	1	4	1	4	1	4	1	4	1	4	1	4

a= 3 b= 1

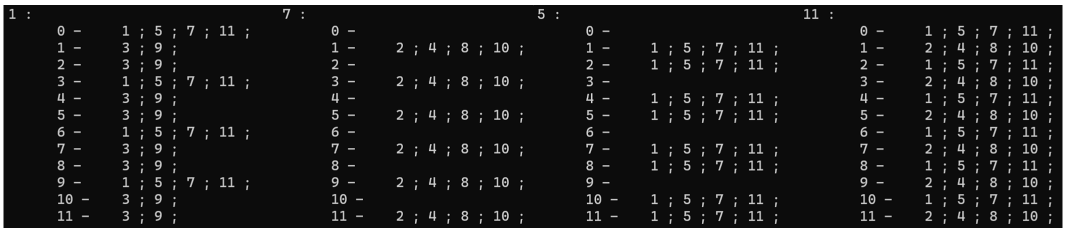
	0	1	2	3	4	5	6	7	8	9	10	11
0	0	3	0	3	0	3	0	3	0	3	0	3
1	11	2	11	2	11	2	11	2	11	2	11	2
2	8	11	8	11	8	11	8	11	8	11	8	11
3	3	6	3	6	3	6	3	6	3	6	3	6
4	8	11	8	11	8	11	8	11	8	11	8	11
5	11	2	11	2	11	2	11	2	11	2	11	2
6	0	3	0	3	0	3	0	3	0	3	0	3
7	11	2	11	2	11	2	11	2	11	2	11	2
8	8	11	8	11	8	11	8	11	8	11	8	11
9	3	6	3	6	3	6	3	6	3	6	3	6
10	8	11	8	11	8	11	8	11	8	11	8	11
11	11	2	11	2	11	2	11	2	11	2	11	2

a= 3 b= -1

Fig 1. All valid tuples

in total, we have only 136 valid tuples to verify for primality out of each 576, more than four time reduction. Moreover, as r is predetermined for each tuple, modulo operations are eliminated completely, at the cost of a small overhead induced by the tuples management. In the accompanying code you can find a version that capitalizes on this observation, with timings reflected in table 2, column *Tuples*.

Next step is suggested by another representation of those values, arranged as in figure 2 – it becomes clear that generally dy follows a $6k \pm i$ pattern, with dx either on a $2k$ pattern for $r = 7$ and 11 or a “100” pattern for $r = 1$, respectively a “011” pattern for $r = 5$ – in total there are six pattern: two for $r = 1$ and 11, one for $r = 5$ and 7.



```
1 :                                     7 :                                     5 :                                     11 :
       0 -      1 ; 5 ; 7 ; 11 ;               0 -                                     0 -                                     0 -      1 ; 5 ; 7 ; 11 ;
       1 -      3 ; 9 ;                        1 -      2 ; 4 ; 8 ; 10 ;               1 -      1 ; 5 ; 7 ; 11 ;               1 -      2 ; 4 ; 8 ; 10 ;
       2 -      3 ; 9 ;                        2 -                                     2 -      1 ; 5 ; 7 ; 11 ;               2 -      1 ; 5 ; 7 ; 11 ;
       3 -      1 ; 5 ; 7 ; 11 ;               3 -      2 ; 4 ; 8 ; 10 ;               3 -                                     3 -      2 ; 4 ; 8 ; 10 ;
       4 -      3 ; 9 ;                        4 -                                     4 -      1 ; 5 ; 7 ; 11 ;               4 -      1 ; 5 ; 7 ; 11 ;
       5 -      3 ; 9 ;                        5 -      2 ; 4 ; 8 ; 10 ;               5 -      1 ; 5 ; 7 ; 11 ;               5 -      2 ; 4 ; 8 ; 10 ;
       6 -      1 ; 5 ; 7 ; 11 ;               6 -                                     6 -                                     6 -      1 ; 5 ; 7 ; 11 ;
       7 -      3 ; 9 ;                        7 -      2 ; 4 ; 8 ; 10 ;               7 -      1 ; 5 ; 7 ; 11 ;               7 -      2 ; 4 ; 8 ; 10 ;
       8 -      3 ; 9 ;                        8 -                                     8 -      1 ; 5 ; 7 ; 11 ;               8 -      1 ; 5 ; 7 ; 11 ;
       9 -      1 ; 5 ; 7 ; 11 ;               9 -      2 ; 4 ; 8 ; 10 ;               9 -                                     9 -      2 ; 4 ; 8 ; 10 ;
       10 -     3 ; 9 ;                        10 -                                    10 -     1 ; 5 ; 7 ; 11 ;               10 -     1 ; 5 ; 7 ; 11 ;
       11 -     3 ; 9 ;                        11 -     2 ; 4 ; 8 ; 10 ;               11 -     1 ; 5 ; 7 ; 11 ;               11 -     2 ; 4 ; 8 ; 10 ;
```

Fig 2. Symmetries in tuples

With a little effort, these patterns can be implemented directly in code, avoiding also the overhead induced by the tuples management – here is an example for one of the six patterns:

```
const unsigned xmax = (unsigned)floor(sqrt(limit / 4));
for(unsigned x=1, jmp=0; x<=xmax; x += 1 + jmp, jmp = 1 - jmp)
    for (unsigned y = 3; ; y += 6)
    {
        unsigned n = (4 * x * x) + (y * y);
        if (n > limit) break;
        FlipBit(n, sieve1);
    }
```

Again, in the accompanying code you can find an implementation of the pattern based algorithm, with timings in the same table 2, column *Patterns*.

Tbl 2. SoA timings (ms) for generating primes up to 10^n – optimized versions

n	$\pi(10^n)$	SoA					357
		1bit $6k \pm 1$	Tuples	Patterns	4 sieves	Incremental	
5	9'592	1	1	1	1	1	-
6	78'498	1	1	1	1	1	1
7	664'579	14	10	6	6	6	9
8	5'761'455	148	97	67	63	56	86
9	50'847'534	2'415	1'590	1'241	643	533	849
10	455'052'511	53'842	-	29'807	17'969	5'420	7'648

And a final touch is to split the sieve buffer in four, based on the fact that now each of the 4 rs (1, 5, 7, 11) is basically treated individually, so we can do all the associated computations decoupled, each r with its own sieve: this will further reduce the arithmetic complexity of the code in sieving and squares detection, at the cost of some added complexity in the final phases. Regarding phase two – eliminating squares – another observation about the square of any prime p is useful here:

$$p^2 = (12k_1 + i)^2 = 12k_2 + i^2 \equiv i^2\,(mod\ 12) \tag{2}$$

For $i \in \{1, 5, 7, 11\}$ we have:

$$i^2 \in \{1, 25, 49, 121\} = \{1, 24+1, 48+1, 120+1\} \equiv 1\,(mod\ 12)$$

thus all $p^2 \equiv 1\,(mod\ 12)$. In conclusion, $(12k + r)p^2 \equiv r\,(mod\ 12)$, which make it very simple to efficiently parse only those multiples of squares that are appropriate for a certain r. In the accompanying code you can find an implementation of the 4 sieves with pattern algorithm – see timings in the same table 2, column *4 Sieves*: more than three times the performance of the basic optimized SoA. It is visible that our SoA variant is now significantly more efficient than basic SoE and, as long as it stays within L2/3 cache, it even beats **357** – only when L3 becomes to small for the sieve size, **357** regains the advantage cause it's incremental nature keeps its footprint under those limits. Nevertheless, it is clear that the pattern doubles the speed, and splitting the sieve in four gives us 3x improvement.

3.2. From incremental to parallel

Given the previous conclusion it becomes evident that we must process SoA incrementally to have the same advantage as **357** when when it comes to cache intensity. Here we must concentrate on the first phase of the sieving (quadratics), because the last part is already uniform and the middle one (squares) is already very cheap in our variant.

There are two approaches to this problem: a pure segmented one (in which each segment is treated individually and computes its own boundaries) and a really incremental one (where each segment takes over from the previous – as in **357**). The segmented approach is a little heavier, but is simpler to manage and can be used also in parallelization; the pure incremental one can be faster, but it requires additional space ($O(\sqrt{N})$) and for a relatively small number of iterations the performance impact should be insignificant. Thus, we implemented the incremental sieving in phase one (quadratics) with the timings reflected in table 2, column *Incremental* – the better performance and especially better linearity of the algorithm, compared with **357**'s SoE is pretty visible now. The time spent in the different phases of the algorithm is also interesting: from the 5.4 seconds spent for 10 billion, the sieving part takes only 3.3 seconds (of which 0.3 in phase 2 – squares); 2.1 seconds are spent in the last phase (the actual generation and counting of the primes, after sieving is completed), which is relatively huge – that's why is very important to consider all phases in benchmarking. Anyway: the code used here is pretty optimized, but is not production quality code, so we assume there is space for another 20-40% performance improvements. Nevertheless, we must note the 10x improvement over the original optimized implementation for large limits.

Of course, the next step is to employ some light parallelization. Hefty parallelization of SoA inside a segment is somewhat difficult because, where SoE simply takes a sieve value one way from 1 to 0 or vice-versa (operation which does not need much synchronization), SoA requires in the first phase (the heaviest one) repeated updates of the same value, where synchronization can not be avoided. Until now the simple segmentation was the only technique to avoid the problem – the 4 sieves technique gives more liberty here, because each sieve is continuous and can be efficiently processed individually without any impact on the other sieves. With only 4 threads we can dramatically reduce the duration, as seen in table 3, column *Light parallel* – in the accompanying code you can find an implementation of this light parallel technique. We must observe that now, from the 2 seconds consumed up to 10 billion, only half is spent sieving: 1 second is wasted computing primes in the correct order.

The light parallel version above does not exploit optimally the resources of the CPU – at least because the sieving for patterns 1 and especially 5 is twice as lengthy as those for pattern 7 and 11: we should allocate twice as many threads for 5 as per 11. Similar for squares sieving, although with much lower impact – for $r = 1$ we have almost 11 times more values to strike out than for $r = 11$: there are unused cores that could be involved in sieving. Of course, one can not have many more threads than physical cores here, because otherwise cache will not be exploited optimally and, above a certain point, thread management / context switching overhead will kill any gain from parallelization. On the other hand, in counting phase the cache usage is uniform, so there we can increase drastically the number of threads to optimally exploit CPUs internal pipelines.

Tbl 3. SoA timings (ms) for generating primes up to 10^n – advanced versions

n	SoA					
	Incremental	Light parallel	Parallel	Full	Segmented	Optim
6	1	1	-	-	-	-
7	6	3	17	19	-	-
8	56	18	21	24	135	-
9	533	210	76	70	200	166
10	5'420	2'048	670	518	583	530
11	55'784	20'350	6'712	5'155	5'866	5'414
12	665'816	208'716	75'786	59'241	59'583	55'700
13	-	-	-	-	750'850	591'257
14	-	-	-	-	11'732'744	6'711'497

We implemented the extended parallel version of the algorithm on our AMD 7900X workstation and experimentally determined an optimum number of 40 threads (10 per sieve) in sieving and 400 threads in counting. This approach dramatically reduced the duration down to 75 seconds up to 1 trillion, as seen in table 3, column *Parallel* – in the accompanying code you can find an implementation of this localized parallel technique.

Still, the middle step (striking out multiples of squares) is not yet segmented, as it needs to access the first prime candidates to compute squares and multiples – it is the least consumer of time, but now that the outer phases were segmented, we must deal with this one, too. Like in standard SoE, we can pre-compute the root primes and use them here. Anyway, up to 10^7 the light parallel version is significantly faster and it allows us to compute primes up to 10^{14}. Even at 10^8 the light version is still faster and that suffice for 10^{16}. One can compute the root primes initially, or in parallel with sieving – for really big numbers the 100 milliseconds saved doing it in parallel are not significant and mess up our conclusions about sieving itself, so the accompanying code implements the sequential version.

The list of root primes is stored as a vector of gaps between primes for better cache locality – the first gap value greater than 256 appears at $p = 436,273,009$ [2], so we are safe to use an 8bit vector up to 4.3×10^8. The time for 1 trillion is now under 1 minute, as seen in table 3, column *Full* – the technique to mark squares based on root primes is exemplified in the accompanying code.

All the phases of the algorithm are now segmented, but they work on the full memory buffer, so the algorithm itself is not segmented as a whole and it

[2] `https://en.wikipedia.org/wiki/Prime_gap`, still valid novemeber 2023

consumes $O(N)$ space – actually $N/24$ bytes, but still this amounts to 42GB for 1 trillion. To get over 1 trillion on a 64GB machine we need to implement the segmentation at the level of the full algorithm, not only for individual phases, and use a small static buffer for all processing – of course, we have to keep the root primes vector, so we can't go lower than $O(\pi(\sqrt{N})) = O(\frac{\sqrt{N}}{\ln\sqrt{N}})$ for space complexity (this can be achieved with a segmented computation for root primes). The overhead imposed by segment and chunk management will hurt the performance, especially for lower ranges, as visible in table 3, column *Segmented*. Still, the performance for higher values is closing in with the *Full* version because of better L2/3 cache intensity.

Nevertheless, there is one little problem: although the number of markings per segment is indeed linear, for really huge values the number of y positionings (we need to compute the start y for each x, as in the code bellow – see accompanying code for all the details) becomes greater and greater – around $n = 10^{14}$, for a segment size 10^{10}, we have approximately 110 million markings but we already have more than 120 million positionings, each one with an *sqrt* operation, hence the grater and greater duration inflicted for huge values.

```
void Pattern11S(uint64 start, uint64 stop, uint8 sieve[])
{
const uint64 nmax = stop - segment_start;
const uint64 xmax = (tpPrime)(sqrt(stop / 4));
for (uint64 x=1, jmp=0; x <= xmax; x += 1+jmp, jmp = 1-jmp)
{
    //get in position
    uint64 y, n0 = 4 * x * x;
    if (n0 < start)
    {
        const uint64 yy = (tpPrime)ceil(sqrt(start - n0));
        y = 6 * (yy / 6) + 3;
        if (y < yy) y += 6;
    }
    else y = 3;

    //sieve
    for (uint64 n = n0 + y*y - segment_start; n <= nmax;
                                    n += 12*y + 36, y += 6)
    {
        assert(n >= start - segment_start);
        FlipBit(n, sieve);
    }
}};
```

Increasing the step size will reduce the number of positionings, but it will worsen the L1/L2 cache locality, so the best compromise for the CPU and desired range is to be found experimenting with several values. The optimized implementation in the accompanying code is interpolating a different cache buffer value for each segment to mitigate this aspect. Also, that version is not computing the start y for each x independently, but it derives it from the previous y value, thus reducing significantly the effort in positioning – this new optimized version is able to compute up to 10^{13} in under 10 minutes, as seen in table 3, column *Optim.* The theoretical complexity of SoA may be sublinear, but in practice, with all the overhead, its arithmetic complexity is slightly over-linear – nevertheless, the original linear character is much more visible now in the optimized version.

Tbl 4. SoA timings (ms) for generating primes in intervals of $length = 5 \cdot 10^{10}$

Interval at	**SoA**	***primesieve***
10^{11}	3'051	402
10^{12}	3'231	514
10^{13}	3'720	626
10^{14}	4'622	767

The segmented algorithm has very low memory footprint and is able to generate the primes using only 0.5GB RAM or less (depending on the segment size). The algorithm manages now to compute all 3.2 trillion primes up to 10^{14} in less than 2 hours, using only 12 CPU cores (on our Ryzen 9 7900X workstation). It is also capable to compute all primes in any interval up to 2^{64} – a small sample of such timings is depicted in table 4.

Regarding the practical significance of SoA, table 4 contains, in the last column, the corresponding timings for Kim Walisch's ***primesieve***[3], the most advanced sieve we know of, computed on the same machine. We see a six times apparent performance gap, but, considering that our proof-of-concept implementation really computes all the primes, where *primesieve* only counts them with extremely advanced, very fast techniques, that gap is really around four times, which is the same order of magnitude and within practical domain. Moreover, our proof-of-concept doesn't even come close to *primesieve* when code optimization is considered: it is quite probable that the same level of optimization can reduce the gap to 2x or less (*primesieve* was at version v11.1 at the moment of writing this article (august 2023) and benefits from many years of intensive code optimization and refinement). This is important because, although primesieve may remain faster for numbers up to 2^{64}, beyond the word size of x64 CPUs the sub-linear SoA, with much better theoretical complexity, may win over SoE due to lower number of operations overall.

[3]`https://github.com/kimwalisch/primesieve`

3.3. Future work perspectives

There are several directions to develop SoA from here:

- Improve the pattern algorithm, for example using separate loops for each eligible value i in $12k + i$;
- Implement the algorithm on GPU devices to exploit their massive parallel hardware advantages;
- Approach Galway [2] and Farach/Tsai [4] work using the same or a similar pattern implementation;
- Achieve a similar level of optimization as *primesieve*; there is ample space for improvement especially regarding the problem of y positioning at the start of each interval, as explained earlier;
- Implement the algorithm beyond 2^{64} using specific techniques; for example, one can still work with values greater than 2^{64} as long as it stays within intervals lower than 2^{64} (which is only natural for any incremental / segmented approach) – any value here can be represented using a base (interval start) plus a value Δ that is lower than 2^{64}: as long as we are very careful to keep track of interval limits we can perform all arithmetic here only with variables Δ lower than 2^{64}.

We think that especially the last point in the enumeration above can be very significant when we try to asses the practical significance of prime sieving techniques. Almost every possible theory regarding primes was tested up to 2^{64} (or 4 x 10^{18}) using SoE based fast sieves, but beyond this value not much was done. We strongly believe that an implementation of an optimized SoA like ours (or a better one hopefully provoked by this modest optimization tentative) will outperform any fast SoE based sieve for such very large values due to sheer complexity advantage. This could advance the limits for testing conjectures like Goldbach and others, improve our knowledge concerning the mysterious distribution of primes and twin prime numbers or exploit lists of very big contiguous primes which are useful resources for pseudo-random sequences (noise simulation, generator seeds, statistical randomness based on prime distribution etc.)

4. Conclusions

For a very long time it was considered that SoE is by far the only approach to practical sieving, although other algorithms were known to have better complexity.

In this article we presented a variant of SoA which allows for significant performance gains over an optimized version of standard SoA in lower ranges, while the gain is even higher for bigger ranges – up to ten times single-threaded performance over an optimized implementation of the original algorithm. Moreover, our method retains better linearity over a much larger set of values, allowing to compute up to one trillion (10^{12}) in about 10 minutes single threaded and less than 4 minutes lightly threaded. A fully parallelized version of the algorithm computes all primes up to one trillion in just 1 minute using only 12 cores of a Ryzen 9 7900X CPU. A segmented version is able to compute up to 10^{14} in less than two hours. This version will compute all the primes in the last interval of length $5 \cdot 10^{10}$ before 10^{14} in just under 5 seconds and proves to have performance in the same order of magnitude with ***primesieve***.

The techniques mentioned here are completely illustrated in the accompanying code – although the code is meant for academic purposes and is not production quality, it does includes a lot of optimizations necessary to get the most out of the algorithm. Nevertheless, we are confident that there is much space for improvement, as we propose in section 3.**3** *Future work*.

Acknowledgement: We express our gratitude to professors Nirvana POPESCU, Emil SLUSANSCHI and Vlad CIOBANU from Computer Science Department in University POLITEHNICA of Bucharest, for their invaluable guidance and advice – their input was decisive for the quality of this paper.

REFERENCES

[1] *A. O. L. Atkin, D. J. Bernstein*, Prime sieves using binary quadratic forms, Mathematics of Computation **73** (2003) 1023–1030. doi:10.1090/S0025-5718-03-01501-1.

[2] *W. F. Galway*, Dissecting a Sieve to Cut Its Need for Space, Technical Report, Springer Berlin Heidelberg, Berlin, Heidelberg, 2000.

[3] *M. Ghidarcea, D. Popescu*, Prime Numbers Sieving - A Systematic Review with Performance Analysis, Algorithms 2024, 17, 157. doi:10.3390/a17040157.

[4] *M. Farach-Colton, M.-T. Tsai*, On the complexity of computing prime tables, 2015. arXiv:1504.05240.

[5] *P. Pritchard*, A sublinear additive sieve for finding prime number, Communications of the ACM **24** (1981) 18–23. doi:10.1145/358527.358540.

[6] *R. C. Singleton*, Algorithm 357: An efficient prime number generator [A1], Communications of the ACM **12** (1969) 563–564. doi:10.1145/363235.363247.