GPGPU AND SIEVE OF SUNDARAM

Mircea GHIDARCEA[1] and Decebal POPESCU[3]

In the quasi-forgotten universe of prime sieving, Sieve of Sundaram is seen as an elegant approach to prime generation, but its performance is significantly worse than basically every other algorithm but brute-force trial-division. Yet, exactly this less than stellar arithmetic complexity makes it an ideal candidate to demonstrate the power of GPU based computing in tackling this kind of problems that evade the usual matrix oriented, symmetrical jobs for which the GPGPU paradigm was natively devised. This paper intends to demonstrate how a well thought approach can benefit from the immense throughput of modern GPUs to solve problems like sieving, which are not perfectly symmetric.

Keywords: GPGPU; prime numbers sieving; OpenCL; algorithms; algorithm optimization; parallel algorithm

1. Introduction

Along with the Sieve of Eratosthenes (SoE) and Euler's Sieve (SoE+), the Sieve of Sundaram (SoS) is one of the three early prime sieves devised by mathematicians [1]. The algorithm was initially thought to be more efficient than the Sieve of Eratosthenes because it eliminates half of the numbers at the start and only needs to iterate through the remaining numbers for marking.

However, after being experimented with and its complexity being studied more closely, it was proven that SoS is significantly less efficient than the Sieve of Eratosthenes in practically every scenario.

In recent years, the use of specialized hardware devices such as GPUs has become increasingly prevalent in performing mathematical calculations, facilitating sometimes dramatic performance improvements compared to traditional calculation methods. Yet, GPUs are generally not optimal for problems that are not symmetrical, as GPUs are designed for highly parallel computations, which are prevalent to symmetrical problems. Non-symmetrical problems can be less suited to the parallel architecture of GPUs, potentially leading to inefficiencies.

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

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

In particular, GPUs are not very well suited for sieving due to the inherently irregular and data-dependent nature of the sieving process. Sieving algorithms often involve a series of conditional checks and memory accesses that are difficult to parallelize effectively. This irregularity leads to poor utilization of the GPU's parallel processing capabilities, as threads can become idle waiting for data or conditional branches to resolve. Additionally, the memory access patterns in sieving are typically random and sparse, which can result in inefficient use of the GPU's high-bandwidth memory architecture. Consequently, the advantages of GPUs in performing highly parallel computations are not fully leveraged in sieving tasks, making them less optimal for this type of problem.

In this paper, we will attempt to demonstrate that GPUs can nonetheless be used to achieve significant speed improvements in sieving. To this end, we will use SoS, a sieve known to have high complexity and, therefore, very low practical efficiency, precisely to illustrate the performance gains that can be obtained from using a GPU.

***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/SoS`*

***NOTE 2**: All the timings for this paper were measured on the same desktop computer with AMD Ryzen 9 7900X CPU (Auto OC on) and AMD 7900XTX GPU. 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]

2. SoS Fundamentals

Some 90 years ago, S.P.Sundaram [2], departing from the observation that any odd composite number N can be expressed as the product of two smaller odd numbers

$$N = (2i + 1)(2j + 1) = 4ij + 2i + 2j + 1 = 2(2ij + i + j) + 1 = 2k + 1$$

concluded that for all integers k for which exists i and j greater than 0 so that $k = 2ij + i + j$, the number

$$N = 2k + 1$$

will not be prime. Vice versa, the number $N = 2k + 1$ will be prime if k cannot be expressed as $2ij + i + j$. So, if we sieve out all numbers in the form of $k = 2ij+i+j$ from an interval $[0 \,..\, N]$, the remaining numbers will generate all prime numbers in interval $[0 \,..\, 2 \cdot N]$ as $2k + 1$.

[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.

Algorithm 2.1 presents a basic version of the sieve, in both normal (straightforward) and additive (avoiding as possible multiplication and other complex operation) versions.

Algorithm 2.1 Sieve of Sundaram

- 1bit implementation
- the additive version is less readable and not really that much faster when using a modern optimizing compiler.

```
uint64_t Sundaram(const uint64_t Nmax, uint8_t vPrimes[]) {
    const uint64_t max_k = Nmax / 2;
    memset(vPrimes, 0, max_k / 8 + 1);
    //sieve
#ifdef ADDITIVE
    for (uint64_t k = 4, delta = 3;
          k < max_k; delta += 2, k += delta + delta - 2)
        for(auto k1 = k; k1 < max_k; k1 += delta)
            vPrimes[k1 / 8] |= BIT_MASK[k1 % 8];
#else
    const uint64_t max_k_sqrt = sqrt(max_k);
    for (uint64_t i = 1; i <= max_k_sqrt; i++)
        for(uint64_t j = i;;j++) {
            uint64_t k = 2 * i * j + i + j;
            if (k > max_k) break;
            vPrimes[k / 8] |= BIT_MASK[k % 8];
        }
#endif // ADDITIVE
    //count
    uint64_t numPrimes = 1; AddPrime(2); // to account for 2
    for (uint64_t k = 1; k < max_k; k++)
        if (not(vPrimes[k / 8] & BIT_MASK[k % 8])) {
            uint64_t n = 2 * k + 1;
            AddPrime(n); numPrimes++;
        }
    return numPrimes;
}
```

As explained in [3], the complexity of Sieve of Eratosthenes (SoE) is:

$$C(SoE) = O(N \ln(\ln(N)))$$

while **SoS** is quite worse with:

$$C(SoS) = O(N \ln(N))$$

The timings indicated in Table 1, comparing similar basic, single-threaded, bit-oriented implementations of the two sieves, corroborate this significant contrast: the disparity is more pronounced than it initially seems. Due to this less than stellar complexity, SoS is not really a contender when it comes to prime sieves with practical value, as shown also in article [1].

Table 1. SoS timings [ms] and cycles [].

Limit(10^N)	**SoE 1bit**	**SoS 1bit**	
		Additive	***Explicit***
7	8	10	10
8	80	124	120
9	1'520	3'757	4'119
10	21'063	68'078	71'963

Placing counters inside the inner loop of the sieve (lines 15–17 in Algorithm 2.1) we can better understand the performance difference between SoE and SoS — see Table 2.

Table 2. SoS vs. SoE counters

Limit (10^N)		7		8		9		10	
SoE	cycles	9.246e6		99.152e6		1.05e9		11.026e9	
	overwrites (%)	4.910e6	(53)	54.913e6	(55)	0.60e9	(57)	6.481e9	(59)
SoS	cycles	17.074e6		199.520e6		2.283e9		25.708e9	
	overwrites (%)	12.738e6	(75)	155.281e6	(78)	1.833e9	(80)	21.163e9	(82)
SoS / SoE		185%		201%		217%		233%	
Net strikes		4.335e6		44.238e6		0.450e9		4.545e9	

For $N = 9$, where SoE executes 1 billion inner cycles with 60% repeat rate, SoS does almost 2.3 billion, with a huge 80% repeat rate — not only more work, but also significantly more duplicated work, and things get worse and worse with larger limits, as seen in Table 2.

2.1. Fast SoS

Although not necessarily impossible, there are slim chances to optimize the algorithm itself, as it does not display any obvious patterns in data processing and the operations are already very simple — the problem lays in the sheer quantity of computations that must be done and which increases exponentially at larger limits.

The other strategy to enhance the algorithm's efficiency would be to optimize its implementation. Looking at the very large value for `Store Latency` in Figure 1, the first line of attack here is to employ contemporary cache-intensive methods, aiming to fully leverage the CPU's capabilities against the huge amount of otherwise very simple instructions: a more refined, incremental code should manage to get decent values even from Sundaram. The sieving part of such a segmented algorithm is presented in Algorithm 2.2.

Indeed, as we can verify in Table 3, the CPI goes dramatically down from 0.88 to 0.37 and the other parameters are overall improved. Table 4 shows the

new, much better timings of the improved version, compared with the basic 1bit additive version. For reference, the table contains also timings for the established **357** sieve [4] as a baseline (as shown in [1], 357 is the most rapid classic sieve algorithm). The complete implementation details can be checked out in the accompanying code.

Algorithm 2.2 Sieve of Sundaram — iterative sieving

- Additive version

```
void Generate1Chunk0(void) {
    const uint64_t val_min = offset;
    const uint64_t val_max = offset + BUFF_SPAN;
    uint64_t i, delta;
    //flag non-primes
    for (i = (unsigned)sqrt(val_max/2), delta = 2 * i + 1;
                                i > val_min; i--, delta -= 2) {
        uint64_t n = 2 * (i + i * i);
        for (unsigned nn = unsigned(n - val_min);
                                nn < BUFF_SPAN; nn += delta)
            mark_val(nn);
    }
    for (; i > 0; i--, delta -= 2) {
        uint64_t j = (uint64_t)((val_min - i) / (1.0 + 2 * i));
        uint64_t n = i + j + 2 * (uint64_t)i * j;
        while (n < val_min)
            n += delta;
        for (unsigned nn = unsigned(n - val_min);
                                nn < BUFF_SPAN; nn += delta)
            mark_val(nn);
    }
}
```

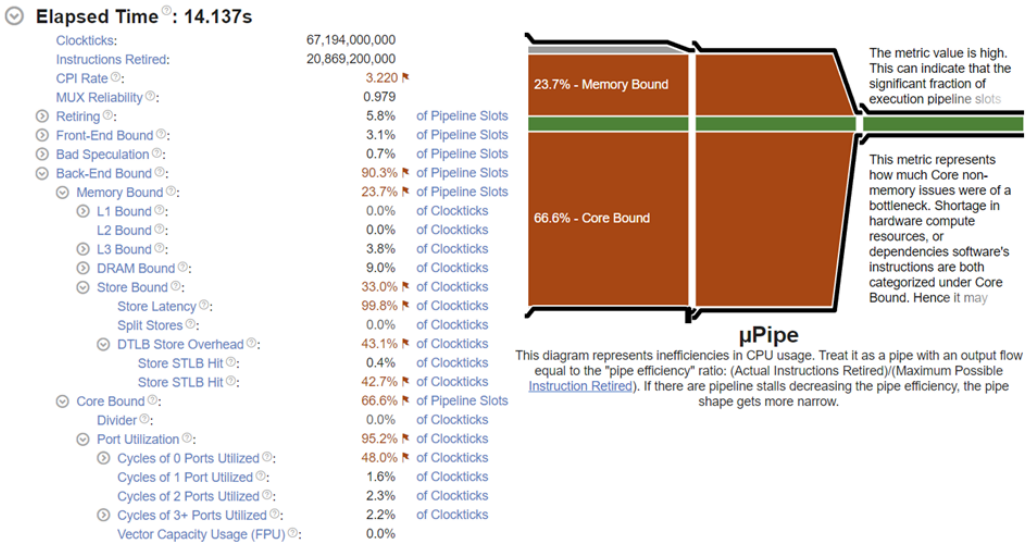

Figure 1. Basic SoS CPU profile (Intel).

Based on this incremental version one could implement a full parallel version of the algorithm — nevertheless, it is clear from these values that SoS is not only a worse performer, but is also by far less linear than SoE/357, and for large ranges it is not at all a viable choice.

Table 3. CPU statistics for SoS variants (AMD uProf) at $Limit = 10^9$.

Process	Basic	Incremental
CYCLES NOT IN HALT	10'476'250'000	4'000'000'000
RETIRED_INST	11'887'750'000	10'953'000'000
CPI	**0.88**	**0.37**
IC MISSES (PTI)	4.70	4.24
IC MISS RATIO	0.35	0.34
L1 ITLB MISSES (PTI)	0.32	0.35
L1 ITLB MISS RATE	0.00	0.00
L1 ITLB REQUESTS (PTI)	160.03	182.44
L2 ACCESSES FROM IC MISSES	3.78	4.14
L2 MISSES FROM IC MISSES	0.79	0.77
OP CACHE FETCH MISS RATIO	0.07	0.06

Table 4. Incremental SoS vs. **357** [ms].

Limit(10^n)	**SoS 1bit**	**SoS inc.**	**357**
7	10	10	9
8	124	104	83
9	3'757	1'120	817
10	68'078	12'120	8'304
11	-	145'180	87'160

3. GPGPU Implementation

Besides being quite a simple algorithm, so it is quite suited for experimentation, SoS has another special characteristic that makes it a very good candidate for a proof-of-concept massively parallel sieve implementation: it does not require the pre-generation and storage of a massive list of root primes (the set of primes up to N that are used to sieve for larger primes up to N^2) like any variant of Eratosthenes and, compared with pattern based Atkin [5] or vertical sieve [7], each chunk has quite a simple and independent positioning procedure, as seen in Algorithm 2.2.

It is exceedingly complex to work directly with a GPU, necessitating the use of a framework or SDK that abstracts this complexity and provides a user-friendly, generic API. Among the available options, **CUDA** stands out as perhaps the most widely used framework today, though it is restricted to NVIDIA hardware. **Metal**, on the other hand, is tailored specifically for Apple's ecosystem. **Vulkan** is a relatively new entrant in the GPGPU domain, though it has not yet gained widespread adoption.

OpenCL is a mature and versatile platform, currently at version 3, and enjoys support from all the major vendors, including NVIDIA, AMD/Xilinx, Intel, and ARM, as well as numerous smaller players. OpenCL also extends its reach to devices like FPGAs and a variety of niche hardware from companies such as Qualcomm, Samsung, and Texas Instruments. This broad compatibility makes OpenCL a highly practical choice, particularly because an OpenCL program can run on any OpenCL-compliant platform, encompassing most CPUs as well. Consequently, we have chosen to implement our GPGPU algorithms using the C++ wrapper for OpenCL (`https://www.khronos.org/opencl/assets/CXX_for_OpenCL.html`).

For those already acquainted with CUDA, it is important to highlight that OpenCL and CUDA share many fundamental similarities. Both are parallel computing platforms specifically designed to leverage the computational prowess of GPUs for general-purpose computing tasks. These frameworks facilitate the development of programs that execute in parallel on the GPU, capitalizing on the extensive parallelism characteristic of GPU architectures. Both OpenCL and CUDA utilize a comparable programming model centered around **kernels**, which are small functions executed in parallel across numerous threads on the GPU. They also feature memory models that are specifically optimized for GPU architectures, with distinct types of memory (such as global, local, and constant memory) that have particular access patterns optimized for parallel execution.

Algorithm 3.1 Sieve of Sundaram — basic kernel

- Basic additive version

```
kernel void SoS_simple(global uint8_t* outBuff,
                       uint32_t outBuffSize, uint64_t offset) {
    uint64_t i = get_global_id(0) + 1;
    uint32_t gsize = get_global_size(0);
    uint32_t gsize2 = gsize + gsize;
    uint64_t val_min = offset;
    uint64_t val_max = offset + outBuffSize;
    uint32_t delta = (2 * i + 1);
    //flag non-primes
    for (; 2 * (i + i * i) < val_max; i += gsize, delta += gsize2) {
        //get to the current interval
        uint64_t j = (val_min > i) ?
                (uint64_t)((val_min - i) / (1.0 + 2.0 * i)) : i;
        uint64_t nstart = i + j + 2 * i * j;
        for (; nstart < offset; nstart += delta);
        nstart -= offset;
        // mark interval
        if (nstart < outBuffSize)
        for (uint32_t n = (uint32_t)nstart; n < outBuffSize; n += delta)
            outBuff[n] = false;
    }
}
```

Furthermore, both platforms organize computations into threads and thread blocks (in the case of CUDA) or work-items and work-groups (in the case of OpenCL). These units of parallelism are then scheduled and executed on the GPU hardware, allowing developers to harness the full potential of GPU computing power.

In Algorithm 3.1 we have a straightforward kernel for a basic implementation of SoS on a GPU.

We have in Table 5 the timings for such a basic implementation of SoS running on an AMD 7900XTX GPU — check the accompanying code for details. Although the kernel code tries to obey the basic rules of GPU programming — avoid complex operations and 64bit data whenever possible, and don't expect too much from the optimizer — we can see that the results are horrendous: although we exploit 96 CU (compute units) each having 32 "threads", resulting in more than 3000 native threads, the performance is worse than the basic one-threaded incremental implementation.

Table 5. SoS GPGPU Timings [ms].

Limit(10^n)	**357**	**Incremental**	**GPGPU SoS Variants**			
			Basic	**Cutoff**	**LDS**	**LDS Prll.**
7	9	10	1'147	298	337	505
8	83	104	1'196	345	381	518
9	817	1'120	5'173	848	769	580
10	8'304	12'120	31'988	5'678	5'388	1'980
11	87'160	145'180	-	72'380	48'877	18'790
12	-	-	-	-	505'372	419'319

There are some generic explanations involving the lower frequency of the GPU cores, the overall cache efficiency of a CPU and generally speaking the huge differences in performance when comparing one-to-one a CPU thread to a GPU one, but the fundamental explanation comes from the fact that those 3072 GPU "threads" are not really independent threads — they correspond with the work-items in OpenCL and are grouped in wavefronts (or warps for CUDA), each consisting of (usually) 32 threads that run in sync — that means that each thread in the workgroup will not end until the last thread in the workgroup has finished all the work, thus keeping the whole CU occupied. Because we are using a stripping approach for domain segmentation, the thread lengths vary significantly within a work-group, the result being that, although the vast majority of threads have finished, a very small number of threads keep everything stalled, and those are the work-items that includes is with very small values. While for i above 100 there are only several hundred iterations in the inner loop, for values bellow 10 there are many thousands, something qualitatively similar to $f(x) = x^{-1}$ function as in Figure 2.

Although most of the work is done in the first hundred milliseconds, a small number of threads will keep working to process all the js for those small is — thus, because the GPU thread is significantly weaker than a CPU thread, we get worse performance.

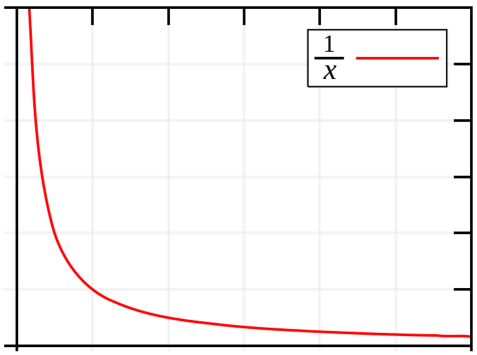

Figure 2. Normal j distribution.

The real art in devising a parallel algorithm is to find a segmentation method for the problem domain that is able to evenly distribute the computation effort between segments — basically the goal is to flatten the curve, in order to achieve the best possible occupancy of the GPU. One possible solution here is to create two different kernels: one very similar with the simple one used above, that works quite well for big values of i, and another one for very small is. The second kernel will transpose the problem, iterating on j in the outer loop, thus surfacing the depth of the inner loop and flattening the curve like in Figure 3. The new results for the cutoff-transpose algorithm are represented also in Table 5, where we can see the dramatic improvement — check the accompanying code for details.

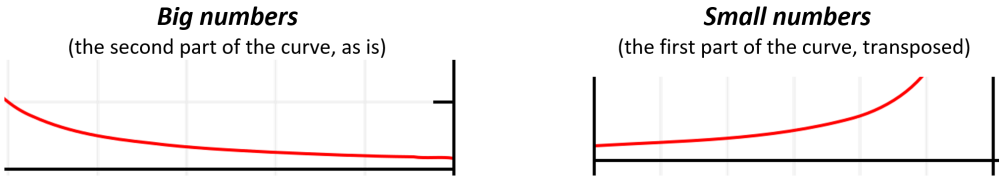

Figure 3. Transposed approach.

Another immediate solution is to use an approach similar with the one for the incremental SoS and exploit the L0 cache (the local data of the GPU CU) — the curve is relatively leveled naturally, and the overhead for positioning in each chunk is mitigated by the fast access of L0 cache, resulting in better timings as seen in Table 5: the improvement is higher as the limit grows. The gist of the segmented kernel is presented in Algorithm 3.2 — further details can be checked out in the accompanying code.

Nevertheless, because in our proof-of-concept the prime counting is executed single-threaded on the host, the overall performance remains not at all impressive. To get a better idea we need to look closer at the sieving part of the algorithm, which is the one performed on the GPU, with timings in Table 6.

Algorithm 3.2 Sieve of Sundaram — segmented kernel

- Barriers are used to guard local buffer initialization and upload
- The outer loop is descending for simplified break logic

```
kernel void SoS_simple_local(global uint8_t* outBuff,
    uint32_t outBuffSize, local uint8_t* workBuff, uint32_t workBuffSize) {
    const uint32_t localID = get_local_id(0), localDim = get_local_size(0);
    const uint32_t thread_range = workBuffSize / localDim;
    const uint32_t thread_offset = localID * thread_range;
    for (uint32_t i = 0; i < thread_range; i++)
        workBuff[thread_offset + i] = 0;
    barrier(CLK_LOCAL_MEM_FENCE);
    const uint32_t groupID = get_group_id(0), groupsNo = get_num_groups(0);
    const uint64_t global_offset = get_global_offset(0);
    const uint32_t localDim2 = localDim + localDim;
    const uint32_t group_offset = groupID * workBuffSize;
    const uint32_t group_span = workBuffSize * 1;
    const uint64_t val_min = global_offset + groupID * group_span;
    const uint64_t val_max = val_min + group_span;
    const uint32_t istart = (unsigned)sqrt(val_max / 2.0f);
    uint32_t i = istart - localID, delta = 2 * i + 1;
    for (;; delta -= localDim2) {
        uint64_t j = (val_min > i) ?
                        (uint64_t)((val_min - i) / (1.0 + 2 * i)) : i;
        uint64_t n = i + j + 2 * (uint64_t)i * j;
        while (n < val_min)
            n += delta;
        for (unsigned nn = (unsigned)(n - val_min);
                      nn < group_span; nn += delta)
            workBuff[nn] = 1;
        if (i > localDim)
            i -= localDim;
        else
            break;
    }
    barrier(CLK_LOCAL_MEM_FENCE);
    for (uint32_t i = 0; i < thread_range; i++)
        outBuff[group_offset + thread_offset + i] =
                                workBuff[thread_offset + i];
}
```

Although the results are embarrassing compared to primesieve (the golden standard in sieving, see [6, 7]), the comparative performance analysis above shows that, with a little effort, a GPU can improve dramatically the sieving performance. Our last step in optimizing this implementation was to parallelize the final step of the sieving pprocess, the actual counting/generation of

primes, so that the final timings would reflect exclusively the duration of the sieving executed on the GPU, plus the minimal unavoidable overhead — the timings can be checked in Table 5; see also the accompanying code for all the implementation details.

Table 6. GPU Timings *(sieving on GPU + data download from device to host)*[ms].

Limit(10^n)	**Primesieve**	**GPGPU SoS**	
		Transposed	**LDS**
7	1	29 + 12	55 + 21
8	2	29 + 12	56 + 22
9	10	126 + 60	104 + 48
10	72	1'421 + 547	962 + 376
11	742	56'305 + 4'675	16'639 + 3'274
12	9'177	-	398'876 + 37'196

Of course, the results here are only at the level of proof-of-concept — there is still a lot of space for optimization, for example:

- flattening the curve inside each chunk using the cutoff-transpose technique, as described above;
- avoiding unnecessary loops because, for larger is, only a small number of js will actually impact a certain chunk and a bucket-like algorithm [8] may benefit the implementation;
- using on the GPU buffer a 1-bit approach for the buffer, as we used on the normal CPU — on the GPU this is particularly difficult as one has to be sure that in the same workgroup cycle there aren't two threads that will try to update the same byte, as this will result in keeping only the last data value and loosign the others.

3.1. Future work

There are two directions that can be pursued regarding Sundaram, based on these findings: a) the implementation of a fully parallel version that optimally utilizes CPU resources; b) the development of a technique that enables 1-bit compression of the sieving buffer for much more efficient exploitation of GPU resources.

In general, the techniques described here for GPU-based massively parallel sieving can also be applied to other variants that allow segmentation and incremental processing, such as Atkin or fast sieve variants of the Sieve of Eratosthenes (SoE).

We are not aware of any GPGPU variant for the Sieve of Atkin, and for the Sieve of Eratosthenes, we have found only one notable implementation:

CUDASieve (`https://github.com/curtisseizert/CUDASieve`) from Curtis Seizert, which is currently quite old and limited to CUDA/Nvidia. It is time to address the sieving problem based on GPGPU using the latest techniques and finding innovative mechanisms to efficiently solve the 1-bit GPU buffer compression problem.

4. Conclusions

In this paper we analyzed some aspects regarding the implementation of SoS.

In the first part we analyzed the pitfalls of Sundaram's algorithm, second only to brute force, and we have implemented an incremental algorithm that could be easily generalized to a full parallel version.

In the second part we demonstrated the utility of GPGPU to increase performance of such algorithms, showcasing two important aspects that have to be considered when tackling a sieving algorithm on a GPU: flattening the curve to maximize GPU utilization and using L0 (Local Data) incrementally to minimize memory latency.

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

REFERENCES

[1] *Ghidarcea, M.; Popescu, D.* Prime Number Sieving—A Systematic Review with Performance Analysis. Algorithms 2024, **17**, 157. `https://doi.org/10.3390/a17040157`.

[2] *Aiyar, V.R.* Sundaram's Sieve for Prime Numbers. Math. Stud. 1934, **2**, 73.

[3] *Crandall, R.; Pomerance, C.* Prime Numbers: A Computational Perspective; Springer-Verlag: New York, NY, USA, 2005. `https://doi.org/10.1007/0-387-28979-8`.

[4] *Singleton, R.C.* Algorithm 357: An Efficient Prime Number Generator [A1]. Commun. ACM 1969, **12**, 563–564. `https://doi.org/10.1145/363235.363247`.

[5] *Ghidarcea, M.; Popescu, D.* Sieve of Atkin Revisited. Sci. Bull. Univ. Politeh. Buchar. 2024, **86**, 15–26.

[6] *Walisch, K.* Primesieve. 2023. Available online: `https://github.com/kimwalisch/primesieve`

[7] *Ghidarcea, M.; Popescu, D.* Static Wheels in Fast Sieves. J. Control. Eng. Appl. Inform. 2024, **26**, 36–43. `https://doi.org/10.61416/ceai.v26i1.8860`.

[8] *Oliveira e Silva, T.* Fast Implementation of the Segmented Sieve of Eratosthenes. 2015. Available online: `https://sweet.ua.pt/tos/software/prime\_sieve.html`