



NILS-CHRISTIAN KEMPKE<sup>1</sup>, THORSTEN KOCH<sup>2</sup>

**A GPU accelerated variant of  
Schroepel-Shamir's algorithm for  
solving the market split problem**

---

<sup>1</sup>  0000-0003-4492-9818  
<sup>2</sup>  0000-0002-1967-0077

Zuse Institute Berlin  
Takustr. 7  
14195 Berlin  
Germany

Telephone: +49 30 84185-0  
Telefax: +49 30 84185-125

E-mail: [bibliothek@zib.de](mailto:bibliothek@zib.de)  
URL: <http://www.zib.de>

ZIB-Report (Print) ISSN 1438-0064  
ZIB-Report (Internet) ISSN 2192-7782

# A GPU accelerated variant of Schroeppel-Shamir’s algorithm for solving the market split problem

Nils–Christian Kempke\*, Thorsten Koch†

February 20, 2026

## Abstract

The market split problem (MSP), introduced by Cornuéjols and Dawande (1998), is a challenging binary optimization problem on which state-of-the-art linear programming-based branch-and-cut solvers perform poorly. We present a novel algorithm for solving the feasibility version of this problem, derived from Schroeppel–Shamir’s algorithm for the one-dimensional subset sum problem. Our approach is based on exhaustively enumerating one-dimensional solutions of MSP and utilizing GPUs to evaluate candidate solutions across the entire problem. The resulting hybrid CPU-GPU implementation significantly outperforms a parallel CPU-only variant, efficiently solving instances with up to 10 constraints and 90 variables. We demonstrate the algorithm’s performance on benchmark problems, solving instances of size (9, 80) in less than fifteen minutes and (10, 90) in up to one day. Given our results, sorting based algorithms can be considered competitive for solving the MSP on modern hardware.

## 1 Introduction

The *market split problem* (MSP) as in [2], is given as the optimization problem


$$\begin{aligned} \min \quad & \sum_{i=1}^m |s_i| \\ \text{s.t.} \quad & \sum_{j=1}^n a_{ij}x_j + s_i = d_i, \quad i = 1, \dots, m \\ & x_j \in \{0, 1\}, \quad j = 1, \dots, n \\ & s_i \in \mathbb{Z}, \quad i = 1, \dots, m. \end{aligned}$$


Here,  $x_j$  are binary decision variables,  $m, n \in \mathbb{N}$ , and we assume  $a_{ij}, d_i \in \mathbb{N}_0$ . The feasibility version of MSP (fMSP) is equivalent to the  $n$ -dimensional subset sum problem ( $n$ -SSP): Find a vector  $x \in \{0, 1\}^n$  such that

$$\sum_{j=1}^n a_{ij}x_j = d_i \quad i = 1, \dots, m. \quad (1)$$

The  $n$ -SSP is NP-complete [5]. As in [2],  $n$ -SSP can be reduced to 1-SSP by introducing a *surrogate constraint*: Given  $D \in \mathbb{N}, D > a_{ij}$ , we replace the set of

---

\*  0000-0003-4492-9818

†  0000-0002-1967-0077

equations in eq. (1) with

$$\sum_{i=1}^m (nD)^{i-1} \sum_{j=1}^n a_{ij} x_j = \sum_{i=1}^m (nD)^{i-1} d_i \quad (2)$$

and obtain an equivalent 1–SSP. Following [2], this article examines  $n$ –SSPs where, given  $m \in \mathbb{N}$  we set  $n = 10(m - 1)$ ,  $a_{ij}$  is chosen uniformly in the range  $[0, 99]$  and  $b_i = \lfloor \frac{1}{2} \sum_{j=1}^n a_{ij} \rfloor$ . These problems are often referred to as  $(m, n)$  for a given  $m$ . In [1] the authors show that this choice of  $m$  and  $n$  leads to a set of hard  $n$ –SSPs exhibiting very few expected solutions ranging from 0.19 expected solutions for  $m = 4$  to 3.32 for  $m = 8$ , monotonically growing with increasing  $m$ .

Several techniques for solving fMSP have been proposed. Branch and cut and branch and bound performs particularly poorly on these instances [1, 2, 9, 12, 8]. The main reason is the vast number of linear basic solutions with value 0 and little pruning during tree exploration. Dynamic programming and sorting based methods suggested in [2] rely on using a surrogate constraint eq. (2) to reduce the problem to 1–SSP. Lattice-based reduction techniques have proven most successful for solving  $n$ –SSP. These approaches include basis reduction with polytope shrinkage [2], basis reduction with linear programming [1], and basis reduction with lattice enumeration [10].

The core contribution of this work is a novel hybrid CPU-GPU co-design that efficiently decomposes and parallelizes the search space, going beyond a simple GPU port to provide a scalable architecture for exact combinatorial search. In this paper, we present a novel GPU-accelerated sorting-based method for solving  $n$ –SSP not relying on a surrogate constraint. Our approach solves instances up to  $(9, 80)$  and  $(10, 90)$  in often less than fifteen minutes or one day, respectively. We report the to-date fastest times for these instances as published in the literature and show that sorting-based methods are competitive for solving  $n$ –SSP. We compare our implementation to a multi-threaded CPU only variant and demonstrate significant speed-ups of our hybrid algorithm.

*Notation* For a given set  $S$ , we use  $2^S := \{s \mid s \subset A\}$  to denote its power set. The cardinality of  $2^S$  is given by  $2^{|S|}$ . We use  $\mathbb{N}$  to denote the natural numbers and define  $\mathbb{N}_0 := \mathbb{N} \cup \{0\}$ .

## 2 Enumerating solutions of 1-SSP

Our algorithm is based on the exhaustive enumeration of all solutions of 1–SSP. Let  $S := \{1, \dots, n\}$  be an index set for  $n \in \mathbb{N}$  and for each  $i \in S$  let  $a_i \in \mathbb{N}_0$  be an integer weight. For any subset  $s \subset S$  we define its subset sum as  $a(s) := \sum_{i \in s} a_i$ . A classic way to find a solution to 1–SSP is Horowitz-Sahni’s two-list algorithm [4], also used in [2] in combination with a surrogate constraint. The two-list algorithm splits the coefficients of 1–SSP into two subsets, generates all subset sums of each subset sorted by value, and then traverses the two sorted lists in ascending and descending order until a pair is found whose combined value satisfies the 1–SSP. For large 1–SSP instances, the space complexity  $\mathcal{O}(2^{\frac{n}{2}})$  of Horowitz-Sahni’s algorithm quickly becomes prohibitive.

An improvement in space complexity is provided by the algorithm of Schroepel-Shamir [7] as shown in Algorithm 1. Instead of generating the two power sets used by Horowitz-Sahni, it uses heaps to dynamically generate each power set, reducing the space complexity to  $\mathcal{O}(2^{\frac{n}{4}})$ . In Algorithm 1, we modified the original algorithm to collect all solutions of 1–SSP. The subsets  $E_A$  and  $E_C$  in line 15 can be determined quickly by linearly iterating the sorted  $2^A$  and  $2^C$  staring at the  $i$ -th and  $j$ -th elements, respectively.

---

**Algorithm 1** Schroepel-Shamir's algorithm for all solutions of 1-SSP

---

**Input:**  $S = \{1, \dots, n\}$ , weights  $(a_1, \dots, a_n)$ ,  $a_i \in \mathbb{N}_0$ ,  $d \in \mathbb{N}_0$ , weight function  $a : 2^S \rightarrow \mathbb{N}_0$

- 1: Initialize set of solutions  $\mathcal{R} \leftarrow \emptyset$
  - 2: Partition  $S$  into  $A, B, C, D$  of size  $\approx \frac{n}{4}$
  - 3: Let  $(s_1^A, \dots, s_{2^{|A|}}^A)$  be an ordering of  $2^A$  with  $a(s_1^A) \leq \dots \leq a(s_{2^{|A|}}^A)$
  - 4: Let  $(s_1^C, \dots, s_{2^{|C|}}^C)$  be an ordering of  $2^C$  with  $a(s_1^C) \geq \dots \geq a(s_{2^{|C|}}^C)$
  - 5: Initialize min-heap  $H_1 \leftarrow \{(s_1^A, s^B) \mid s^B \in 2^B\}$  with key  $a(s_1^A) + a(s^B)$
  - 6: Initialize max-heap  $H_2 \leftarrow \{(s_1^C, s^D) \mid s^D \in 2^D\}$  with key  $a(s_1^C) + a(s^D)$
  - 7: **while**  $H_1$  and  $H_2$  not empty **do**
  - 8:    $(s_i^A, s^B) \leftarrow \text{top}(H_1)$ ,  $(s_j^C, s^D) \leftarrow \text{top}(H_2)$
  - 9:    $\alpha := a(s_i^A) + a(s^B)$ ,  $\beta := a(s_j^C) + a(s^D)$
  - 10:   **if**  $\alpha + \beta < d$  **and**  $i + 1 \leq 2^{|A|}$  **then**
  - 11:      $\text{pop}(H_1)$  and insert  $(s_{i+1}^A, s^B)$  into  $H_1$
  - 12:   **else if**  $\alpha + \beta > d$  **and**  $j + 1 \leq 2^{|C|}$  **then**
  - 13:      $\text{pop}(H_2)$  and insert  $(s_{j+1}^C, s^D)$  into  $H_2$
  - 14:   **else**
  - 15:      $E_A := \{s \in 2^A \mid a(s) = a(s_i^A)\}$ ,  $E_C := \{s \in 2^C \mid a(s) = a(s_j^C)\}$
  - 16:      $\mathcal{R} \leftarrow \mathcal{R} \cup \{s^A \cup s^B \cup s^C \cup s^D \mid s^A \in E_A, s^C \in E_C\}$
  - 17:      $\text{pop}(H_1)$
  - 18:     **if**  $i + |E_A| \leq 2^{|A|}$  **then** insert  $(s_{i+|E_A|}^A, s^B)$  into  $H_1$  **end if**
  - 19:      $\text{pop}(H_2)$
  - 20:     **if**  $j + |E_C| \leq 2^{|C|}$  **then** insert  $(s_{j+|E_C|}^C, s^D)$  into  $H_2$  **end if**
  - 21:   **end if**
  - 22: **end while**
  - 23: **return**  $\mathcal{R}$
-

### 3 GPU accelerated Schroepel-Shamir for the $n$ -SSP

Given Algorithm 1 for retrieving all solutions of the 1-SSP, we solve  $n$ -SSP in the following way: For a given  $n$ -SSP, we use Schroepel-Shamir’s algorithm to find all solutions of the 1-SSP obtained by only considering the first row of eq. (1). We note that, in order to be correct, the 1-SSP algorithm used for  $n$ -SSP needs to take generate all solutions to 1-SSP explicitly including zero coefficients as well (as done in Algorithm 1). Whenever a set of solutions to 1-SSP is found, we verify it against the rest of the problem. The procedure is shown in Algorithm 2. To

---

#### Algorithm 2 Schroepel-Shamir for $n$ -SSP

---

**Input:**  $S := \{1, \dots, n\}$ ,  $A \in \mathbb{N}_0^{m \times n} = (a_{ij})$ ,  $d \in \mathbb{N}_0^m$

- 1: Set up heaps  $H_1, H_2$  as in Algorithm 1 w.r.t. the weights  $(a_{1i}, \dots, a_{1n})$  and  $d_1$ .
  - 2: **while**  $H_1, H_2$  not empty **do**
  - 3:   Iterate  $H_1, H_2$  as before, using  $\alpha, \beta, s^B$ , and  $s^D$
  - 4:   **if**  $\alpha + \beta = d_1$  **then**
  - 5:     Extract the sets of equal weight  $E_A$  and  $E_C$  as before
  - 6:     **for all**  $s^A \in E_A, s^C \in E_C$  **do**
  - 7:       Let  $x \in \{0, 1\}^n$  be the characteristic vector of  $s^A \cup s^B \cup s^C \cup s^D$
  - 8:       **if**  $Ax = d$  **then return**  $x$  **end if**
  - 9:     **end for**
  - 10:   **end if**
  - 11: **end while**
- 

make this approach feasible, we rely on GPU acceleration for the validation loop line 6 to line 9 in Algorithm 2. After collecting all solutions in line 5, we offload the solution validation to the GPU while the CPU continues searching for 1-SSP solutions. This creates a CPU-GPU hybrid pipeline interleaving collection and validation operations. The GPU validation algorithm is shown in Algorithm 3. Instead of naively checking each pair in the validation loop, we hash the partial

---

#### Algorithm 3 GPU-accelerated solution validation

---

**Input:**  $E_A, E_C, s^B$ , and  $s^D$  as in Algorithm 2 line 6;  $A, b$  as in Algorithm 2

- 1: Compute  $Q := \{Ax \mid x \text{ is characteristic vector of } s^A \cup s^B, s^A \in E_A\}$
  - 2: Compute  $R := \{b - Ax \mid x \text{ is characteristic vector of } s^C \cup s^D, s^C \in E_C\}$
  - 3:  $Q_{\text{enc}} \leftarrow \text{encode\_kernel}(Q)$
  - 4:  $R_{\text{enc}} \leftarrow \text{encode\_kernel}(R)$
  - 5:  $\text{sort\_kernel}(Q_{\text{enc}})$
  - 6:  $\text{parallel\_binary\_search\_kernel}(Q_{\text{enc}}, R_{\text{enc}})$
- 

subset sum vectors in the `encode_kernel`, sort one hash set, and perform a parallel binary search for elements of the unsorted set. For lines 1 and 2, we pre-compute and buffer the partial vectors  $Ax$  for all characteristic vectors. Sorting and parallel binary search are implemented using CUDA thrust<sup>1</sup>. Encoding uses a custom hash kernel, simplified shown in Algorithm 4. For large  $m$ , the number of 1-SSP solutions passed to Algorithm 3 grows rapidly, potentially exceeding GPU memory. We then validate  $Q$  and  $R$  quadratically in chunks by partitioning both arrays. This

---

<sup>1</sup><https://nvidia.github.io/cccl/thrust/>

---

**Algorithm 4** `encode_kernel`: parallel hash encoding

---

**Input:**  $d_i \in \mathbb{N}^m, i = 0, \dots, N - 1$ ; array hash of size  $N$ 

```
1: if threadId <  $N$  then  
2:    $h \leftarrow 0$   
3:   for  $j = 0, \dots, m - 1$  do  $h \leftarrow \text{hash\_two}(h, (d_{\text{threadId}})_j)$  end for  
4:    $\text{hash}[\text{threadId}] \leftarrow h$   
5: end if
```

---

creates a bottleneck that could be addressed using multiple GPUs, though our current implementation uses a single GPU.

## 4 Computational Results

Our implementation is available on GitHub<sup>2</sup>. Experiments were conducted on a NVIDIA GH200 Grace-Hopper super-chip<sup>3</sup>, with an ARM Neoverse-V2 CPU (72 cores), 480 GB of memory, and one NVIDIA H200 GPU with 96 GB of device memory. We used the fMSP instances provided in QOBLIB<sup>4</sup> [6]. QOBLIB contains for each  $m \in \{3, \dots, 15\}$ ,  $K \in \{50, 100, 200\}$ , four feasible fMSP instances with coefficients uniformly drawn in  $[0, K)$ . We reflect this using the extended notation  $(m, n, K)$ . We ran each instance with our CPU-GPU hybrid algorithm and, formulated as a linear integer program (as in the original MSP) using Gurobi 11 [3]. Additionally, we ran instances for  $m \in \{3, \dots, 8\}$  with a CPU only parallel implementation of our algorithm relying on OpenMP<sup>5</sup> using all 72 available cores (also available on GitHub). We used a time limit of three days. We present our results in table 1. The table shows solution times of our CPU-GPU hybrid algorithm per instance, as well as the average runtimes of our CPU-GPU variant, our CPU-only variant, and Gurobi. The CPU-GPU algorithm solves all instances up to  $m = 10$ . For rows where the **Class** is annotated by “\*”, we first applied a surrogate constraint dimension reduction, reducing the first 2 constraints into one. The reduction shows speed-ups for (10, 90, 100) decreasing runtime to about 66 000 seconds. Generally, instances with a smaller coefficient range solve faster, and solution time grows rapidly with increasing  $m$ . We could not solve the instances (10, 90, 200) or any instance larger than (11, 100, 50). Compared to the parallel CPU implementation, the hybrid implementation is much more competitive and suffers less from an increase in complexity. It outperforms the CPU-only variant by roughly a factor of 10. We could not solve instances larger than (8, 70) efficiently on CPU. Gurobi ran out of memory for all instances larger than (7, 60)

A performance analysis of our hybrid algorithm reveals that its scalability is primarily constrained by GPU memory capacity for storing intermediate results and hashes and by CPU single-thread performance for managing the one dimensional search. Further gains would therefore depend more on increased GPU memory bandwidth and capacity than on higher clock speeds. This confirms that the method’s advantage stems from exploiting the GPU’s architectural strengths, massive parallelism and high memory throughput, rather than raw clock speed alone. On the other hand, the CPU-only variant would strongly benefit from an increased memory bandwidth, as solution validation time dominates the runtime.

Table 2 extends the comparison from [10] to include our approach (note, the basis

---

<sup>2</sup><https://github.com/NCKempke/MarketShareGpu>

<sup>3</sup><https://www.nvidia.com/en-us/data-center/grace-hopper-superchip/>

<sup>4</sup><https://git.zib.de/qopt/qoblib-quantum-optimization-benchmarking-library/>

<sup>5</sup><https://www.openmp.org/>

Table 1: Solutions times in seconds for fMSPs with Schroeppel-Shamir and Gurobi

Class	Instance 1	Instance 2	Instance 3	Instance 4	Avg.	Avg. CPU	Avg. Gurobi
(7, 60, 50)	0.37	0.37	0.39	0.35	0.37	2.79	243.32
(7, 60, 100)	1.66	0.88	1.38	0.86	1.20	19.94	1 086.08
(7, 60, 200)	2.07	2.45	2.35	1.19	2.02	24.15	3 158.37
(8, 70, 50)	1.37	1.12	1.00	1.50	1.25	21.97	MEM
(8, 70, 100)	5.80	8.07	9.19	8.28	7.84	386.21	MEM
(8, 70, 200)	15.60	20.13	8.45	27.03	17.80	879.25	MEM
(9, 80, 50)	23.25	10.78	14.68	15.40	16.03	574.54	MEM
(9, 80, 100)	472.88	101.52	300.37	69.51	236.07	15 212.71	MEM
(9, 80, 200)	548.83	505.44	486.97	541.80	520.76	34 391.54	MEM
(10, 90, 50)	153.50	288.41	74.13	155.29	167.83	16 127.69	MEM
(10, 90, 50)*	2 957.91	2 234.68	3 866.84	2 144.57	2 803.80	203 164.16	MEM
(10, 90, 100)	152 323.86	209 021.22	176 957.14	31 544.88	148 663.34	-	MEM
(10, 90, 100)*	104 230.63	30 141.04	56 104.74	76 068.47	66 636.22	-	MEM
(10, 90, 200)	-	-	-	-	-	-	MEM
(10, 90, 200)*	-	-	-	-	-	-	MEM
(11, 100, 50)	110 032.34	5 313.42	42 940.19	52 118.21	52 601.04	-	MEM
(11, 100, 50)*	34 151.77	34 669.73	54 455.52	44 320.57	41 399.39	-	MEM

Table 2: Literature results in seconds for fMSPs ( $m, n, 100$ )

Prob. size	B&B [2]	B&C [2]	DP	Basis [2]	Group [2]	Sort (ours)	LLL [1]	Enum [10]	Gurobi
(3,20)	10.3	178.67	1.11	239.40	0.12	0.17	-	0.01	0.1
(4,30)	2 271.65	-	19.67	-	17.96	0.20	-	0.08	0.7
(5,40)	-	-	-	-	1 575.58	0.22	62.4	1.01	0.8
(6,50)	-	-	-	-	22 077.32	0.29	2 190.0	2.16	79.0
(7,60)	-	-	-	-	-	1.20	-	28.2	1 086.0
(8,70)	-	-	-	-	-	7.84	-	678	
(9,80)	-	-	-	-	-	236.07	-	9 733	
(10,90)	-	-	-	-	-	66 636.22	-	(655 089)	

enumeration result in brackets for (10, 90) was obtained with a single instance). The table combines results from different papers run on different (and sometimes quite outdated) hardware. Our algorithm updates the *Sort* column and currently provides the fastest reported results for fMSPs. However, it would be worth re-running all experiments with up-to-date implementations and on state-of-the art hardware to provide a more complete picture, which is out of the scope of this paper.

## 5 Conclusion

In this paper, we presented a novel hybrid CPU-GPU approach for solving the  $n$ -SSP by accelerating a variant of the Schroepel–Shamir algorithm. Our implementation efficiently solves feasibility market split instances of size up to (10, 90) in under one day on average, representing, to our knowledge, the best published results for this problem class. This work demonstrates the potential of heterogeneous computing for exact combinatorial search, where algorithms effectively leveraging both CPU and GPU remain rare. For future work, we plan to explore GPU-accelerated lattice enumeration methods, as described in [10], which we believe offer a promising direction for further performance improvements on MSP.

## Acknowledgements

The work for this article has been conducted in the Research Campus MODAL funded by the German Federal Ministry of Education and Research (BMBF) (fund numbers 05M14ZAM, 05M20ZBM, 05M2025).

## References

- [1] Aardal et al. (1999). Market Split and Basis Reduction: Towards a Solution of the Cornuéjols–Dawande Instances. In *Lecture Notes in Computer Science* (pp. 1–16). Springer Berlin Heidelberg. [https://doi.org/10.1007/3-540-48777-8\\_1](https://doi.org/10.1007/3-540-48777-8_1)
- [2] Cornuéjols, G., & Dawande, M. (1998). A Class of Hard Small 0–1 Programs. In *Lecture Notes in Computer Science* (pp. 284–293). Springer Berlin Heidelberg. [https://doi.org/10.1007/3-540-69346-7\\_22](https://doi.org/10.1007/3-540-69346-7_22)
- [3] Gurobi Optimization, LLC. (2023). Gurobi (Version 11). <https://www.gurobi.com>
- [4] Horowitz, E., & Sahni, S. (1974). Computing Partitions with Applications to the Knapsack Problem. *Journal of the ACM*, 21(2), 277–292. <https://doi.org/10.1145/321812.321823>
- [5] Karp, R. M. (1972). Reducibility among Combinatorial Problems. In *Complexity of Computer Computations* (pp. 85–103). Springer US. [https://doi.org/10.1007/978-1-4684-2001-2\\_9](https://doi.org/10.1007/978-1-4684-2001-2_9)
- [6] Koch et al. (2025). Quantum Optimization Benchmark Library – The Intractable Decathlon (Version 1). arXiv. <https://doi.org/10.48550/ARXIV.2504.03832>
- [7] Schroepfel, R., & Shamir, A. (1981). A  $T = O(2^{n/2})$ ,  $S = O(2^{n/4})$  Algorithm for Certain NP-Complete Problems. *SIAM Journal on Computing*, 10(3), 456–464. <https://doi.org/10.1137/0210033>

- [8] Vogel, H. (2012). Solving market split problems with heuristical lattice reduction. *Annals of Operations Research*, 196(1), 581–590. <https://doi.org/10.1007/s10479-012-1143-0>
- [9] Wang et al. (2009). Solving the market split problem via branch-and-cut. *International Journal of Mathematical Modelling and Numerical Optimisation*, 1(1/2), 121. <https://doi.org/10.1504/ijmmo.2009.030091>
- [10] Wassermann, A. (2002). Attacking the Market Split Problem with Lattice Point Enumeration. *Journal of Combinatorial Optimization*, 6(1), 5–16. <https://doi.org/10.1023/a:1013355015853>
- [11] Williams, H. P. (1978). *Model Building in Mathematical Programming*. John Wiley & Sons Ltd.
- [12] Wu et al. (2013). Solving the market split problem using a distributed computation approach. In 2013 *IEEE International Conference on Information and Automation* (pp. 1252–1257). <https://doi.org/10.1109/icinfa.2013.6720486>