



NILS-CHRISTIAN KEMPKE¹, THORSTEN KOCH²

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

¹  0000-0003-4492-9818
²  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
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†

July 7, 2025

Abstract

The market split problem (MSP), introduced by Cornuéjols and Dawande (1998), is a challenging binary optimization problem that performs poorly on state-of-the-art linear programming-based branch-and-cut solvers. 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 efficiently solves 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.

1 Introduction

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


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


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 $x_j \in \{0, 1\}^n$ such that

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

Despite its compact notation, n –SSP is NP-complete [5]. As in [2], n –SSP can be reduced to 1–SSP by introducing the *surrogate constraint*: Given $D \in \mathbb{N}, D > a_{ij}$, we replace the set of equations in eq. (1) with

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

*  0000-0003-4492-9818

†  0000-0002-1967-0077

and obtain the 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.

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 the 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].

In this paper, we present a novel GPU-accelerated sorting-based method for solving n -SSP not relying on the surrogate constraint. Our approach solves instances up to (9, 80) and (10, 90) in often less than fifteen minutes or one day, respectively.

2 Enumerating solutions of 1-SSP

Our algorithm is based on the exhaustive enumeration of all solutions of 1-SSP. We define the subset sum of a set $s \subset S = \{a_1, \dots, a_n\}$, $a_i, n \in \mathbb{N}$ as $a(s) := \sum_{a_i \in s} a_i$. A classic way to find a solution to 1-SSP is Horowitz-Sahni's two-list algorithm [4], also used in [2] combined with the 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 $O(2^{\frac{n}{2}})$ 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, which reduces the space complexity to $O(2^{\frac{n}{4}})$. In algorithm 1, we modified the original algorithm to collect all solutions of 1-SSP. The extraction in line 14 can be done via linear iteration.

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). 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 make this approach feasible, we rely on GPU acceleration for the validation loop in algorithm 2. After collecting all solutions in line 3, 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 $Q \times R$, we hash the partial 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

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

Input: $S = \{a_1, \dots, a_n\}$, $a_i, n \in \mathbb{N}$, $d \in \mathbb{N}$, $a : 2^S \rightarrow \mathbb{N}$

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

Algorithm 2 Schroepfel–Shamir for n -SSP

Input: $A \in \mathbb{N}^{m \times n} = (a_{i,j}), d \in \mathbb{N}^m$

- 1: Set up heaps H_1, H_2 as in algorithm 1 for $S := \{a_{1,1}, \dots, a_{1,n}\}$ and d_1
 - 2: **while** H_1, H_2 not empty **do**
 - 3: Extract tuples $Q \subset H_1, R \subset H_2$ with equal combined weights α, β
 - 4: **for all** $(s^A, s^B) \in Q, (s^C, s^D) \in R$ **do**
 - 5: Let $x \in \{0, 1\}^n$ be the characteristic vector of $s^A \cup s^B \cup s^C \cup s^D$
 - 6: **if** $Ax = d$ **then return** x **end if**
 - 7: **end for**
 - 8: Advance in H_1 or H_2 depending on $\alpha + \beta$ and d
 - 9: **end while**
-

Algorithm 3 GPU-accelerated solution validation

Input: $Q = \{(s_i^A, s_j^B)\}, R = \{(s_k^C, s_l^D)\}, A \in \mathbb{N}^{m \times n}, b \in \mathbb{N}^m$

- 1: Compute $a(Q) := \{Ax \mid x \text{ is characteristic vector of } s_i^A \cup s_j^B, (s_i^A, s_j^B) \in Q\}$
 - 2: Compute $a(b-R) := \{b-Ax \mid x \text{ is characteristic vector of } s_k^C \cup s_l^D, (s_k^C, s_l^D) \in R\}$
 - 3: $Q_{\text{enc}} \leftarrow \text{encode_kernel}(a(Q))$
 - 4: $R_{\text{enc}} \leftarrow \text{encode_kernel}(a(b-R))$
 - 5: $\text{sort_kernel}(Q_{\text{enc}})$
 - 6: $\text{parallel_binary_search_kernel}(Q_{\text{enc}}, R_{\text{enc}})$
-

pre-compute and buffer the partial vectors Ax for all characteristic vectors. Sorting and parallel binary search are implemented using CUDA thrust¹. Encoding uses a custom hash kernel, simplified shown in algorithm 4. For large m , the number of

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** $\text{threadId} < N$ **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**
-

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 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 GitHub². Experiments were conducted on a NVIDIA GH200 Grace-Hopper super-chip³, with an ARM Neoverse-V2 CPU (72 cores), 480 GB of memory, and one NVIDIA H200 GPU with 96 GB of device

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

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

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

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

Class	Instance 1	Instance 2	Instance 3	Instance 4	Average	Average Gurobi
(7, 60, 50)	0.37	0.37	0.39	0.35	0.37	243.32
(7, 60, 100)	1.66	0.88	1.38	0.86	1.20	1 086.08
(7, 60, 200)	2.07	2.45	2.35	1.19	2.02	3 158.37
(8, 70, 50)	1.37	1.12	1.00	1.50	1.25	MEM
(8, 70, 100)	5.80	8.07	9.19	8.28	7.84	MEM
(8, 70, 200)	15.60	20.13	8.45	27.03	17.80	MEM
(9, 80, 50)	23.25	10.78	14.68	15.40	16.03	MEM
(9, 80, 100)	472.88	101.52	300.37	69.51	236.07	MEM
(9, 80, 200)	548.83	505.44	486.97	541.80	520.76	MEM
(10, 90, 50)	153.50	288.41	74.13	155.29	167.832	MEM
(10, 90, 50)*	2 957.91	2 234.68	3 866.84	2 144.57	2 803.80	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	Gurobi	DP	Group	Sort	LLL	Bas. Enum
(3,20)	0.13	1.11	0.12	0.17	-	0.01
(4,30)	0.78	19.67	17.96	0.20	-	0.08
(5,40)	0.81	-	1 575.58	0.22	62.4	1.01
(6,50)	79.09	-	22 077.32	0.29	2 190.0	2.16
(7,60)	1 086.08	-	-	1.20	-	28.2
(8,70)	-	-	-	7.84	-	678
(9,80)	-	-	-	236.07	-	9 733
(10,90)	-	-	-	66 636.22	-	(655 089)

memory. We used the fMSP instances provided in QOBLIB⁴ [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 algorithm and, formulated as a linear integer program (as in the original MSP) using Gurobi 11 [3]. We used a time limit of three days. We present our results in table 1. Our 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 exponentially with increasing m . We could not solve the instances $(10, 90, 200)$ or any instance larger than $(11, 100, 50)$. Gurobi ran out of memory for all instances larger than $(07, 60, .)$. Table 2 extends the

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

comparison from [10] to include our approach (note, the basis enumeration result in brackets for (10, 90) was obtained with a single instance). We replaced branch-and-bound/cut with the Gurobi results and dropped the basis reduction proposed in [2]. Our algorithm updates the *Sort* column and currently provides the fastest reported results for fMSPs.

5 Conclusion

In this paper, we presented a novel approach for solving n -SSP using a GPU-accelerated variant of Schroepfel-Shamir's algorithm. Our GPU-CPU hybrid algorithm solves instances up to (10, 90) in, on average, less than one day, representing, to our knowledge, the best published results on fMSP. For further work, we plan to explore parallelized and GPU accelerated lattice enumeration approaches as described in [10].

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
- [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
- [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
- [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). *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>