Adapted parallel Quine-McCluskey algorithm using GPGPU

Vladimír Siládi
Faculty of Natural Sciences
Matej Bel University
Banská Bystrica, Slovakia
vladimir.siladi@umb.sk

Michal Povinský
Faculty of Natural Sciences
Matej Bel University
Banská Bystrica, Slovakia
michal.povinsky@umb.sk

Ľudovít Trajteľ
Faculty of Natural Sciences
Matej Bel University
Banská Bystrica, Slovakia
ludovit.trajtel@umb.sk

Maxatbek Satymbekov
Institute of Information systems
Kazakh National University named after Al-Farabi
Almaty, Kazakhstan
m.n.satymbekov@gmail.com

***Abstract*—This paper deals with parallelization of the Quine-McCluskey algorithm. This algorithm is a method used for minimization of boolean functions. The algorithm has a limitation when dealing with more than four variables. The problem computed by this algorithm is NP-hard and run-time of the algorithm grows exponentially with the number of variables. It is possible to adapt the Quine-McCluskey algorithm for running on a parallel computing system. The graphics processing units (GPUs) brings acceleration of computing process. This solution differs from the previous solution in the use of bit fields. Parallelization of the algorithm is implemented through Compute Unified Device Architecture (CUDA).**

***Index Terms*—Graphics processing unit, Minimization methods, Digital circuits, Boolean functions, Hardware**

I. Introduction

English mathematician and philosopher G. Boole invented his algebra in 1854 [1]. A few decades later C. E. Shannon showed how the *Boolean algebra* can be used in the design of digital logic circuits [21]. A boolean function is a function that produces a boolean value output by logical calculation of boolean inputs. It plays key roles in programming algorithms and design of digital logic circuits. Minimization of boolean function (using Boolean laws) is able to optimize the algorithms and digital logic circuits. Finding and minimization of boolean functions (normal forms) is one of the crucial problems in logic circuits design.

Two popular techniques for minimization/simplification of boolean functions are used:

- Karnaugh map (the method based on a graphical representation of boolean functions) [12],
- the Quine-McCluskey algorithm (the Method of Prime Implicants; sometimes referred to as the Tabulation Method) [16].

Both of these methods are mechanical in nature (their efficiency depend on the designer's abilities). The methods designed for computer and human (e.g. Quine-McCluskey method, Karnaugh, Svoboda, Veitch, Marquand maps,...) has got advantages and disadvantages. We can find some comparison of them [17]. Karnaugh map is a diagrammatic and mechanical technique based on a special form of Venn diagram [24]. Usually it is used to handle boolean functions (expressions) with no more than six variables. But, Karnaugh map could be used for another purpose e.g. artificial intelligence to map human mind, high performance computing to build networking logics [15].

The Quine-McCluskey algorithm is a method used for minimization of boolean functions. The method was formulated by Quine and later improved by McCluskey [19] [20] [16]. The Quine-McCluskey method is a tabulation method. It is functionally identical to Karnaugh Map [12], but this method makes it more efficient (the tabular form overcomes the limitation associated with graphical representation) for use in computer algorithms (method is *Computer Based Technique* for minimization of boolean functions; it gives a deterministic way to check that the minimal form of a boolean function has been reached).

The method involves two steps:

1) determination of prime implicants (finding all prime implicants of the function),
2) selection of essential prime implicants (use those prime implicants in a prime implicant chart to find the essential prime implicants of the function, as well as other prime implicants that are necessary to cover the function).

The combining minterms with other minterms as a part of the Quine-McCluskey algorithm is natural parallel. In this paper we present an improved parallel algorithm for the first step of the Quine-McCluskey method.

II. Related work

The Quine-McCluskey method, a tool of discrete mathematics, is very simple and systematic technique for minimizing boolean functions [11]. However, the computational complexity of the Quine-McCluskey algorithm is $O\left(N^{\log_2 3} \log_2 N\right)$, with input length $N = 2^n$ [23]. The minimization of a boolean function is (probably) a hard

978-1-5386-0889-0/17/$31.00

problem [23] [25]. Prasad, Beg and Singh analysed the behavior of Quine-McCluskey simplification for different number of product terms and introduced a mathematical model to predict the boolean space complexity. They mathematically modelled boolean function complexity by the following equation:

$$N = \alpha \cdot t^{\beta} \cdot e^{-t \cdot \gamma} + 1 \tag{1}$$

where,

N is the number of literals,

t is the number of non-repeating product terms in the boolean function,

α, β and γ are three constants depend on the number of variables [18]. A couple of solutions were introduced to quickly and automatically simplify boolean functions [9] [10] [11]. Majumder et al. designed a technique based on decimal values. This technique decreases the probability of an error occurrence [13]. Non-deterministic, stochastic and heuristic computational models offer interesting tools to find unconventional alternative solutions [7] [8]. El-Bakry et al. used neural networks [2], fast neural networks [3], modular neural networks [4] for the minimization (simplification).

Another question of this minimization problem is the choice of a suitable data structure for representing boolean functions and an associated set of manipulation algorithms. Graph-based solution have shown Bryant [6].

In our previous work [22] we presented an parallel implementation of Quine-McCluskey algorithm on graphics processing unit through Compute Unified Device Architecture (CUDA). We designed parallel function, which naive transcribed natural parallelism of the Quine-McCluskey algorithm. The parallel implementation utilised a memory inefficiently.

III. Novel algorithm for simplification

We process the data in multiple step. Basically, the original Quine-McCluskey algorithm is adopted. New approach differs in terms representation. The goal of this algorithm is more intensive memory usage. A bitmap representation is used.

In each step of the algorithm, implicants are first partitioned by the positions of dashes in the terms. Each partition is then scanned for mergeable terms. Small partitions use a simple $O\left(N^2\right)$ algorithm, while large partitions use our $O\left(N\right)$ algorithm.

In our term merging algorithm the list of terms is first transformed into a bitmap (bit array in term of programming language C), representing each term as one set bit. Dashes in the term are treated as zeroes when calculating the bit's index, for example:

$$\underset{minterm}{(1-10-0)} \longmapsto \underset{array\ of\ bits}{101000} \longmapsto \underset{index.}{(40)_{10}}$$

Then the list of terms is scanned and for each term all suitable terms are looked up in the bitmap to merge with. Two conditions are applied:

- the current term differs from the term to merge with in exactly one bit,
- the different bit is zero in the current term.

This means only a small number of positions in the bitmap are checked for each term. The partitioning ensures the correct merging (dashes are represented with zeros). Duplicate terms are also never generated. The merged rows are then written to a new list to be used by the next step of the algorithm and both original terms are marked as used. If both of the original rows were optional, the new row will also be marked as optional.

IV. CUDA implementation

The Quine–McCluskey algorithm has got naturally parallel parts, which can be implemented on the SIMD architecture and therefore on the GPU (SIMT architecture). Our term merging algorithm contains this parallel part. Therefore, another significant improvement is expected to achieve a speed up of computation by implementation on a GPU. The design of the sequential CPU implementation was based on the parallel GPU implementation (*see* Listing 1). Listing 1 shows the GPU kernel used for finding merge-able implicants in a partition.

Listing 1. Part of the parallel implementation on GPU

```
1  static __global__ void bitarray(int *vars, int bits,
2   int dashes, int *bitarray,
3   uint32_t *used, uint32_t *output)
4  {
5    .
6    .
7    .
8   if (row_in_block == 0 &&
9    bit_in_row < ROWS_PER_BLOCK) {
10   rows[bit_in_row] = bitarray[rownumber+bit_in_row];
11   usedbuf[bit_in_row] = 0;
12  }
13
14  __syncthreads();
15  found[bit_in_block] = 0;
16  if (rows[row_in_block] & (1<<bit_in_row)) {
17   for (j=0; j<bits; j++) {
18    uint32_t m = 1 << j;
19    if (dashes & m) continue;
20    uint32_t otheridx = bitnumber^m;
21    if (!(otheri & (1<<(otheridx%32)))) continue;
22    atomicOr(usedbuf+row_in_block, (1<<bit_in_row));
23    if (bitnumber & m) continue;
24    if ((m-1) & dashes) continue;
25    found[bit_in_block] |= m;
26   }
27  }
```

First of all, the bitmap is loaded into shared memory and flag of use is set on zero for each representation of minterm (rows 7–9). After threads synchronisation, the combining minterms with other minterms is executed by commands on rows 14–24. These parts of the program code are implementation of above mentioned natural parallelism by CUDA. The kernel runs on GPU (device) and other parts of the algorithm run on CPU (host). This requires the transfer of data from the host to the device and vice versa.

The kernel is called with the following arguments:

- status variable array (`int *vars`),
- term size in bits (`int bits`),

- positions of dashes in the current partition (`int dashes`),
- term bitmap (`int *bitarray`),
- used row flag array (`uint32_t *used`),
- output array (`uint32_t *output`).

Each block processes a number of terms. Each group 32 threads processes 32 terms, each thread looking up a term differing in a bit at one position. The results are then written to an output array.

V. Experiments and results

The number of bits in results is the number of inputs to each function, so the number of output values for each function is $2^{number\ of\ bits}$. Each optimized function was randomly generated and the value of fill is the probability of each bit being set to 1. Higher fill values cause more terms to be created, which significantly increases both time and memory requirements, *see* Fig. 1.

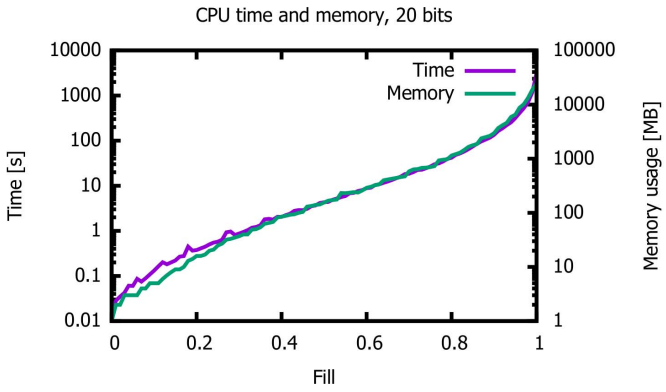

Fig. 1. CPU time and memory usage for a 20 bit function for different fill values

This limited our experiments to either small functions or low fill values. We were experimenting with functions of 20 input variables (*see* Fig. 2), 24 input variables (*see* Fig. 3), 28 input variables (*see* Fig. 4), 32 input variables(*see* Fig. 5).

For experiments on the GPU, three computers each with 64GB RAM, two Xeon E5-2650v2 CPUs and two Tesla K20m graphics cards with compute capability 3.5 (denoted *GPU1*), one computer with Geforce 9800GT GPU with compute capability only 1.1 (denoted *GPU2*) and one computer with Geforce GTX260M GPU with compute capability 1.3 (denoted *GPU3*) were used. For CPU experiments, computers with two Xeon X5670 CPUs and 48GB RAM and computers with two Xeon E5-2650v2 CPUs and 128GB RAM were used (in both cases denoted *CPU*).

For larger experiments, we were able to use on-disk storage for temporary implicant lists, *see* Fig. 5. Unfortunately we were unable to use this with GPU version of the algorithm. While for the smaller tested problem instances running on the GPU is slower than running on the CPU, we expect that GPU would be significantly faster for larger instances.

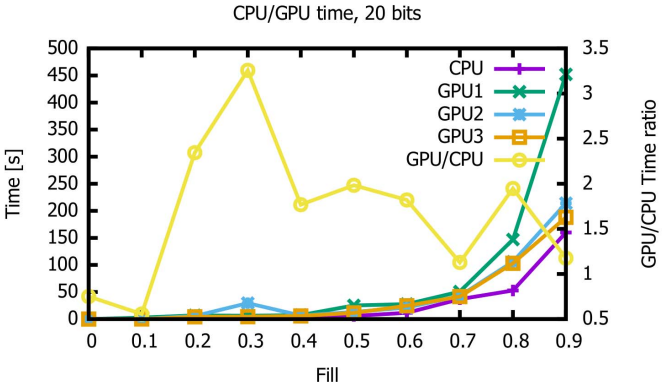

Fig. 2. CPU and GPU time for 20 bit functions

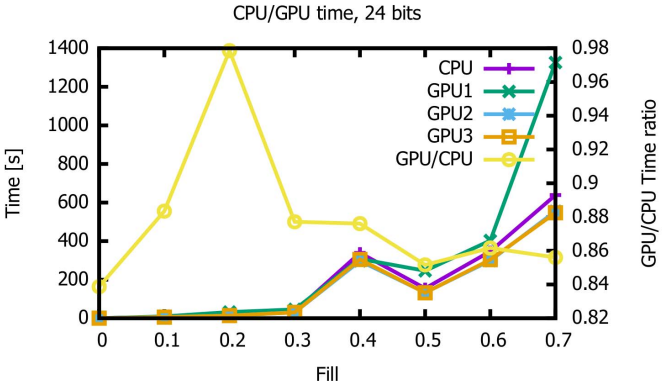

Fig. 3. CPU and GPU time for 24 bit functions

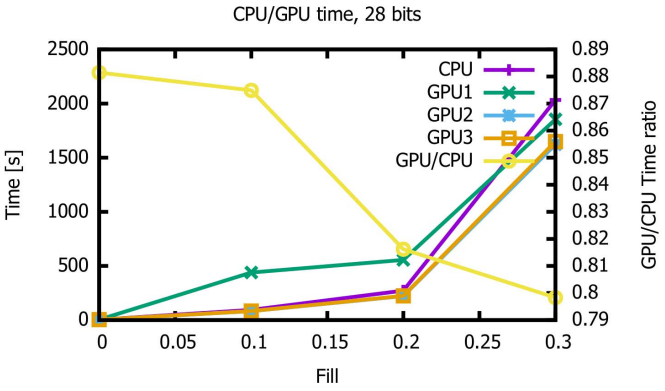

Fig. 4. CPU and GPU time for 28 bit functions

We were unable to test larger instances, as the host memory requirements grow exponentially.

Finally, one of the experiments ran on more powerful graphics card GeForce GTX 1080 Ti with computing capability 6.1 to prove weaknesses in the algorithm. We measured GPU memory usage for different problem sizes using nvidia-smi. GPU memory usage is constant during execution of the program, so no advanced memory profiling

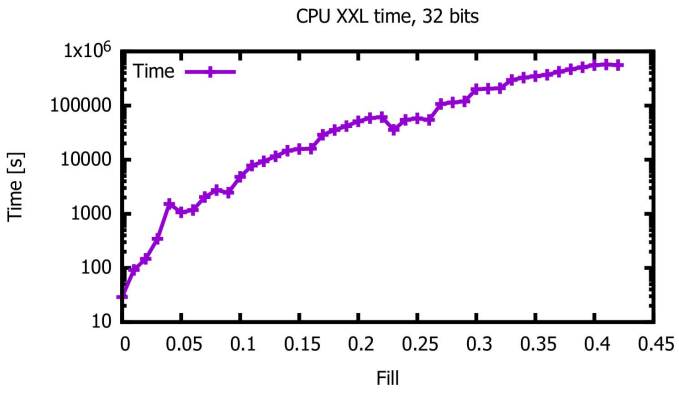

Fig. 5. CPU times for 32 bit functions with out-of-core storage

is required, *see* Fig. 7.

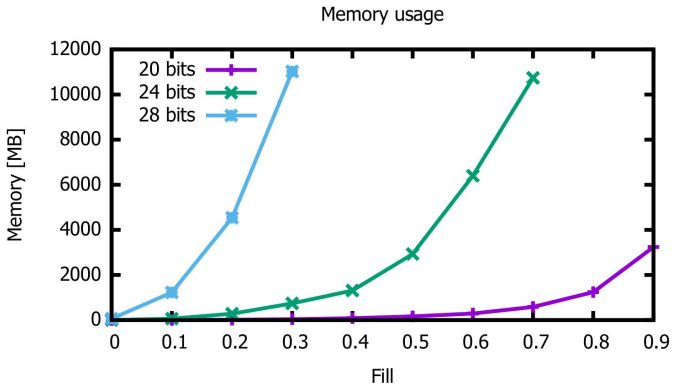

Fig. 6. On-disk storage using for temporary data – memory usage

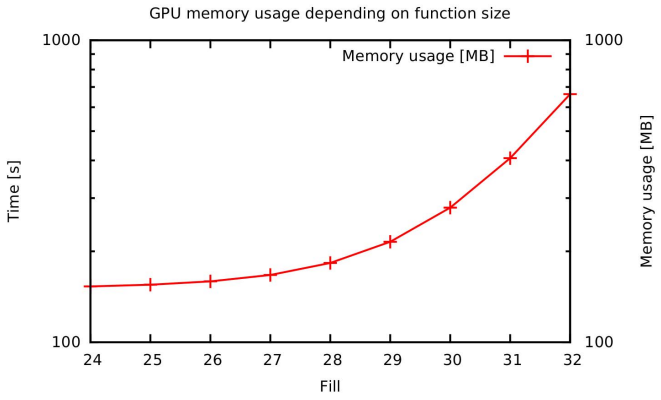

Fig. 7. GPU memory usage on GeForce GTX 1080 Ti depending on function size

Unfortunately, our improved algorithm has not saved memory of the host, but it saved memory of the device. In addition, the profiler conclusion is: "This kernel exhibits low compute throughput and memory bandwidth utilization relative to the peak performance of GeForce GTX 1080 Ti. These utilization levels indicate that the performance of the kernel is most likely limited by the latency of arithmetic or memory operations. Achieved compute throughput and/or memory bandwidth below 60% of peak typically indicates latency issues." (Fig. 8).

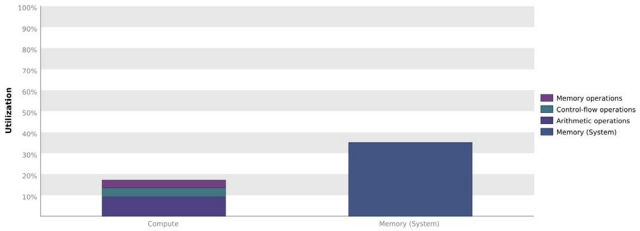

Fig. 8. Kernel performance by instruction and memory latency

The kernel's achieved occupancy of 48.1% is significantly lower than its theoretical occupancy of 100%. Most likely this indicates that there is an imbalance in how the kernel's blocks are executing on the shared memories so that all shared memories are not equally busy over the entire execution of the kernel.

The experiment has shown that this parallelization concept is still not very suitable for implementation on GPU.

VI. Conclusions

The CUDA implementation has been modified according to the new algorithm. Unfortunately, the algorithm still requires a large amount of CPU memory, which often limited the size of problems, which we could solve on our hardware. While for the smaller tested problem instances running on the GPU is slower than running on the CPU, we expect that GPU would be significantly faster for larger instances. We were unable to test larger instances, as the host memory requirements grow exponentially. Although presented parallel algorithm utilises the device memory more efficiently than our previously published parallel algorithm, it is still not very suitable for implementation on GPU. This way of adapting the natural parallel parts of the Quine-McCluskey algorithm on GPU still does not improve significantly the speed-up of the minimisation process. In future, research effort should be focused on optimisation of host's part of the algorithm. There is the weakness of this implementation on GPU.

Acknowledgment

Part of the computing was performed in the High Performance Computing Centre of the Matej Bel University in Banská Bystrica using the HPC infrastructure acquired in project ITMS 26230120002 and 26210120002 (Slovak infrastructure for high-performance computing) supported by the Research & Development Operational Programme funded by the ERDF.

REFERENCES

[1] G. Boole, "An Investigation of the laws of thought: On which are founded the mathematical theories of logic and probabilities," Watchmaker Publishing, 2010.

[2] H. M. El-Bakry, "Fast Karnough map for simplification of complex boolean functions," In: Proc. of 10th WSEAS International Conference on Applied Computer Science (ACS'10), Japan, pp. 478–483, 2010.

[3] H. M. El-Bakry, A. Atwan, "Simplification and implementation of boolean functions," International Journal of Universal Computer Sciences, vol. 1, Issue 1, pp. 19–33, 2010.

[4] H. M. El-Bakry, N. Mastorakis, "A fast computerized method for automatic simplification of boolean functions," In: Proceedings of the 9th WSEAS International Conference on Systems Theory and Scientific Computation (ISTASC '09), Moscow, Russia, pp. 99–107, 2009.

[5] H. M. El-Bakry, A. Atwan, N. Mastorakis, "A new technique for realization of boolean functions," In: Proc. of Recent Advances in Artificial Intelligence, Koweledge Engineering and Databases, Cambridge, UK, pp. 260–270, 2010.

[6] R. E. Bryant, "Graph-based algorithms for boolean function manipulation," IEEE Transaction on Computers, vol. C-35, no. 8, pp. 677–691, 1986.

[7] I. Dirgová Luptáková, M. Šimon, L. Huraj, J. Pospíchal, "Neural gas clustering adapted for given size of clusters," Mathematical Problems in Engineering, vol. 2016, Article ID 9324793, p. 7, 2016.

[8] I. Dirgová Luptáková, J. Pospíchal, "Maximum Traveling Salesman Problem by Adapted Neural Gas," In: Advances in Intelligent Systems and Computing book series (AISC, volume 576), International Conference on Soft Computing-MENDEL. Springer, Cham, pp. 168–175, 2016.

[9] A. Dușa, A. Thiem, "Enhancing the minimization of boolean and multivalue output functions with eQMC," The Journal of Mathematical Sociology, 39:2, pp. 92–108, 2015, DOI:10.1080/0022250X.2014.897949

[10] B. Gurunath, N.N. Biswas, "An algorithm for multiple output minimization," IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems, vol. 8, Issue 9, pp. 1007–1013, IEEE, 1989.

[11] T. K. Jain, D. S. Kushwaha, A. K. Misra, "Optimization of the Quine-McCluskey method for the minimization of the boolean expressions," Fourth International Conference on Autonomic and Autonomous Systems (ICAS'08), Gosier, 2008, pp. 165–168.

[12] M. Karnaugh, "The map method for synthesis of combinational logic circuits," Transactions of the American Institute of Electrical Engineers, vol. 72 part I,pp. 593–598, 1953.

[13] A. Majumder, B. Chowdhury, A. J. Mondai, K. Jain, "Investigation on Quine McCluskey method: A decimal manipulation based novel approach for the minimization of boolean function," International Conference on Electronic Design, Computer Networks & Automated Verification (EDCAV), 2015, DOI: 10.1109/EDCAV.2015.7060531.

[14] V. Manojolovič, "Minimization of Switching Functions using Quine-McCluskey Method," International Journal of Computer Application, vol. 82, no. 4, pp. 12–16, 2013.

[15] R. Mehta, R. Saini, N. Mudgal, M. Dhankar, "Delivering High Performance Result with Efficient Use of K-Map," International Journal of Control and Automation, vol. 9, no. 2, pp. 307–312, 2016.

[16] E. J. McCluskey, "Minimization of boolean functions," Bell System Technical Journal, vol. 35, Issue 6, pp. 1417–1444, 1956.

[17] M. Petrík, "Quine–McCluskey method for many-valued logical functions," Soft Computing, vol. 12, Issue 4, pp. 393–402, Springer-Verlag, 2007.

[18] P. W. Chandana Prasad, Azam Beg, Ashutosh Kumar Singh, "Effect of Quine-McCluskey simplification on boolean space complexity," In: Innovative Technologies in Intelligent Systems and Industrial Applications, CITISIA 2009, IEEE, 2009.

[19] W. V. Quine, "A way to simplify truth functions," The American Mathematical Monthly, vol. 62, no. 9, pp. 627–631, Mathematical Association of America, 1955.

[20] W. V. Quine, "The problem of simplifying truth functions," The American Mathematical Monthly, vol. 59, no. 8, pp. 521–531, Mathematical Association of America, 1952.

[21] C. E. Shannon, "A symbolic analysis of relay and switching circuit," Electrical Engineering, vol. 57, Issue 12, pp. 713–723, IEEE, 1938.

[22] V. Siládi, T. Filo, "Quine-McCluskey algorithm on GPGPU," In: AWERProcedia Information Technology and Computer Science, vol. 4 3rd World Conference on Innovation and Computer Science (INSODE-2013), pp. 814–820, 2013.

[23] S. P. Tomaszewski, I. U. Celik, G. E. Antoniou, "WWW-based boolean function minimization," International Journal of Applied Mathematics and Computer Science,vol. 13, no. 4, pp. 577–583, 2003.

[24] J. Venn, "On the diagrammatic and mechanical representation of propositions and reasonings," Philosophical Magazine, Series 5, 10:59, 1–18, DOI: 10.1080/14786448008626877.

[25] I. Wegener, "The complexity of boolean functions," John Wiley & Sons, Inc. New York, NY, USA,1987.