2024 Design, Automation & Test in Europe Conference (DATE 2024)

An Efficient Logic Operation Scheduler for Minimizing Memory Footprint of In-Memory SIMD Computation

Xingyue Qian[1], Zhezhi He[2], and Weikang Qian[1,3]
[1]University of Michigan-SJTU Joint Institute, [2]School of Electronic Information and Electrical Engineering, and [3]MoE Key Lab of AI, Shanghai Jiao Tong University, Shanghai, China
Emails: {qianxingyue, zhezhi.he, qianwk}@sjtu.edu.cn

***Abstract*—Many in-memory computing (IMC) designs based on *single instruction multiple data (SIMD)* concept have been proposed in recent years to perform primitive logic operations within memory, for improving energy efficiency. To fully exploit the advantage of SIMD IMC, it is crucial to identify an optimized schedule for the operations with less intermediate memory usage, known as *memory footprint* (*MF*). In this work, we implement a recursive partition-based scheduler which consists of our scheduler-friendly partition algorithm and a modified optimal scheduler. Compared to three state-of-the-art heuristic strategies, ours can reduce MF by 56.9%, 46.0%, and 31.9%, respectively.**

I. Introduction

In-memory computing (IMC) is a technique that reduces data transfer between processor and memory, improving energy efficiency. A popular IMC design style enables memory to perform primitive operations such as majority (MAJ) and XOR and also leverages the *single instruction multiple data (SIMD)* concept, where the same operation is applied to some rows of the memory *bitwise* in each cycle and generates the output stored in another row. A SIMD IMC can implement an arbitrary high-level function by two steps, synthesis and scheduling. The former converts a high-level function into a netlist of the supported operations, while the latter determines the *execution sequence* (*ES*) of the operations. Since cross-array communication is costly, building a good scheduler that can effectively use the limited rows within a *single* array to accomplish the target computation is crucial to obtain a good end-to-end performance. Typically, the primary inputs (PIs) of a function stay in the memory throughout the computation process, so our target is to minimize the *memory footprint* (*MF*), which is the number of rows needed to store the intermediate variables and the primary outputs (POs). By minimizing MF, we can compute functions with larger netlists within a single array (*i.e.,* eliminate cross-array communication).

There are several existing schedulers aiming at minimizing the MF for SIMD IMC. OptiSIMPLER [1] formulates a Boolean satisfiability (SAT) problem that is further solved by a satisfiability modulo theories (SMT) solver. It can obtain the minimum MF for a given netlist, but only works for small netlists due to its long run time. Hence, the other works turn to heuristic methods [2]–[4]. Although they can work well for some netlists, they fail to give satisfactory results for the others. In this paper, we propose a scheduling-friendly graph partition technique that can be combined with the optimal scheduler from [1] to further reduce the MF.

This work is supported by National Natural Science Foundation of China under Grants T2293700, T2293701, T2293704, and 62102257, and National Key R&D Program of China under grant number 2020YFB2205501 and 2022YFB4500200. Corresponding authors: Zhezhi He and Weikang Qian.

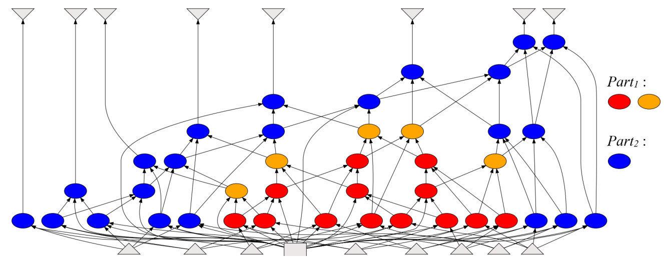

Fig. 1. An example of partitioned netlist.

II. Method

A. Scheduling-Friendly Bi-Partition

Our graph partition method is based on a scheduling-friendly bi-partition. We use the netlist in Fig. 1, produced by logic synthesis, to illustrate the idea of the bi-partition.

Let $\mathcal{N}$, $\mathcal{E}$, and $\mathcal{PO} \subseteq \mathcal{N}$ denote the sets of nodes, edges, and PO nodes, respectively. Note that each node $n \in \mathcal{N}$ corresponds a supported operation, so PIs are not in set $\mathcal{N}$. Each directed edge $(i, j) \in \mathcal{E}$ indicates that the output of node i is an input of node j. The netlist in Fig. 1 is divided into two partitions: $Part_1$ contains the red and orange nodes, and $Part_2$ contains the blue nodes. All the edges between $Part_1$ and $Part_2$ point to $Part_2$. An orange node is a node in $Part_1$ that is a PO or has a fanout in $Part_2$. We call it a *boundary node* (*BN*) of $Part_1$. Under this setting, it is easy to schedule the two partitions sequentially. We can first ignore $Part_2$ and schedule $Part_1$, treating the BNs, *i.e.*, orange nodes, as the POs. Then, with the BNs stored in memory, we can go on to schedule $Part_2$ without concerning all the red nodes. The overall MF is just the larger one of the MFs of the two smaller scheduling problems.

As the final MF depends on the bi-partitions, we propose an integer linear programming (ILP)-based method to find a scheduling-friendly bi-partition. For each node n, we use a binary variable p_n to denote whether it is in $Part_2$ and b_n to denote whether it is a BN of $Part_1$. Our ILP constraints are:

$$p_i \leq p_j, \ \forall (i, j) \in \mathcal{E}, \tag{1}$$

$$(1-\varepsilon)|\mathcal{N}|/2 \leq \sum_n p_n \leq (1+\varepsilon)|\mathcal{N}|/2, \tag{2}$$

$$b_i = \bar{p}_i, \ \forall i \in \mathcal{PO}, \tag{3}$$

$$b_i = \bar{p}_i \wedge \left(\bigvee_{j:(i,j)\in\mathcal{E}} p_j \right), \ \forall i \in \mathcal{N} \setminus \mathcal{PO}, \tag{4}$$

979-8-3503-4859-0/DATE24/© 2024 EDAA

where $\varepsilon \in [0, 1]$ is a parameter. Eq. (2) requires the two partitions to have approximately the same size (*i.e.*, the number of nodes) so that the sub-netlists shrink quickly, and Eqs. (3) and (4) are based on the definition of a BN.

The ILP objective is to minimize the number of BNs, *i.e.*, $\min \sum_n b_n$. This is because the BNs are treated as POs of $Part_1$, and hence, all of them must stay in the memory when the schedule of $Part_1$ finishes. To minimize the MF, it is important to first minimize the number of BNs. We use Gurobi [5] to solve this ILP problem.

B. Recursive Partition-Based Scheduler

We recursively apply the above bi-partition algorithm to partition a large netlist, while a few modifications have to be made. When partitioning $Part_2$, the orange nodes in $Part_1$ can be viewed as the PIs of $Part_2$. However, unlike PIs that always reside in the memory, their memory occupation can be freed when no longer needed. Thus, we call them *temporary inputs* (*TIs*). When partitioning $Part_2$, its TIs may become the BN of the first partition of $Part_2$. Thus, we also add the variables b_i for the TIs in the ILP formulation and include them in the objective function. Furthermore, we add the constraints on the variables b_i for the TIs based on the definition of a BN.

Hereby, we denote a (sub-)netlist as $G = (\mathcal{N}, \mathcal{E}, \mathcal{PO}, \mathcal{TI})$, consisting of node set $\mathcal{N}$, edge set $\mathcal{E}$, PO set $\mathcal{PO}$, and TI set $\mathcal{TI}$. The bi-partition algorithm is implemented as a function *Partition*, whose input is G and outputs are two partitions, *i.e.*, $Part_1$ and $Part_2$, and the boundary node set $\mathcal{BN}$.

The pseudocode of our recursive partition-based algorithm is shown in Algorithm 1. The output of the algorithm is a minimized MF $M_{\min}$ and its corresponding ES $E_{\min}$. The function *Optimal* is the optimal scheduler modified from [1], where the modification is to handle the TIs.

Algorithm 1: Recursive partition-based algorithm.

Function: $Scheduler(G = (\mathcal{N}, \mathcal{E}, \mathcal{PO}, \mathcal{TI}))$

1 **if** $|\mathcal{N}| \leq N_{node}$ **then**

2 $\quad (M_{\min}, E_{\min}) = Optimal(G)$; **return** $(M_{\min}, E_{\min})$;

3 $(Part_1, Part_2, \mathcal{BN}) = Partition(G)$;

4 $(M_{\min 1}, E_{\min 1}) = Scheduler((Part_1, \mathcal{E}_1, \mathcal{BN}, \mathcal{TI}))$;

5 $(M_{\min 2}, E_{\min 2}) = Scheduler((Part_2, \mathcal{E}_2, \mathcal{PO}, \mathcal{BN}))$;

6 **return** $(\max(M_{\min 1}, M_{\min 2}), \{E_{\min 1}, E_{\min 2}\})$;

In the recursive algorithm, when the size of the netlist is no larger than a user given bound N_{node}, we call the optimal scheduler directly in Lines 1–2. Otherwise, we call our bi-partition algorithm in Line 3. The first partition is scheduled in Line 4, using $Part_1$ as node set, $\mathcal{BN}$ as PO, and $\mathcal{TI}$ as TI. The second partition is scheduled in Line 5, using $Part_2$ as node set, $\mathcal{PO}$ as PO, and $\mathcal{BN}$ as TI. $\mathcal{E}_1$ and $\mathcal{E}_2$ contain the edges within the two sub-netlists respectively. Line 6 combines the schedules of the two sub-netlists and returns MF as $\max(M_{\min 1}, M_{\min 2})$ and ES as $\{E_{\min 1}, E_{\min 2}\}$, which means that the operations in $Part_2$ are put after those in $Part_1$. Note that the initial call of *Scheduler* sets $\mathcal{TI} = \emptyset$.

III. Experimental Results

This section shows the experimental results. We implement our algorithm and the algorithms for comparison in C++. All the benchmarks are converted to *XOR-majority graph (XMG)* supported in XMG-GPPIC [2] and optimized with Mockturtle [6]. As mentioned before, we do not include the PIs into MF. Also, for some benchmarks, several POs are directly the PIs, so we do not include those POs into the netlist. Balancing performance and run time, we set $N_{node} = 80$ in Algorithm 1 and $\varepsilon = 0.1$ in Eq. (2).

Table I. Performance comparison of various schedulers.

	#PI	#PO	$\lvert\mathcal{N}\rvert$	MF [2]	[3]	[4]	**Ours**
int2float	11	7	213	33	23	24	16
c880	60	26	250	63	30	37	26
router	60	3	201	29	22	25	17
cavlc	10	11	616	119	85	81	58
c3540	50	22	764	87	75	67	44
priority	128	8	594	118	65	52	17
c6288	32	32	748	177	89	47	38
c7552	207	53	803	134	144	115	53
systemcdes	512	126	1791	231	377	165	152
bar	135	128	2796	393	293	308	145
des_area	496	64	3465	490	373	219	130
sin	24	25	3538	527	366	331	255
max	512	130	2031	316	263	263	264
tv80	732	360	6724	989	692	518	360
arbiter	256	129	11839	467	635	443	301
voter	1001	1	3894	61	133	53	52
square	64	126	9067	1267	469	927	393
sqrt	128	64	17188	194	252	192	191
aes_core	1319	532	15650	856	2144	631	532
multiplier	128	128	14303	1594	398	342	253
log2	32	32	19812	1996	1146	1081	765
GEOMEAN				251	200	159	**108**

We test all 44 benchmarks with netlist sizes less than 20,000 in the popular benchmark sets including EPFL [7]. For 23 of them, the heuristic methods (*i.e.*, [2]–[4]) can achieve good result, *i.e.*, the MF is greater than the number of POs by only 1 or 2. For those benchmarks, we also test them with our scheduler, which can achieve the same or better result. Table I shows the performance, *i.e.*, MF, of the algorithms for the other 21 benchmarks. Compared to competing works of XMG-GPPIC [2], SIMPLER [3], and STAR [4], our method can reduce the MF by 56.9%, 46.0%, and 31.9%, respectively. We can see that our scheduler can achieve less MF for all benchmarks except *max*. Compared with the best result of the competing methods for each benchmark, ours still can reduce the MF by 28.2% on average.

IV. Conclusion

This work proposes a scheduler for SIMD IMC to minimize MF for a given netlist. It is based on a recursive scheduling-friendly graph partition algorithm. The experiments show that our scheduler outperforms the states of the art.

References

[1] D. Bhattacharjee *et al.*, "Synthesis and technology mapping for in-memory computing," in *Emerging Computing: From Devices to Systems: Looking Beyond Moore and Von Neumann*, 2022, pp. 317–353.

[2] C. Nie *et al.*, "XMG-GPPIC: Efficient and robust general-purpose processing-in-cache with XOR-Majority-Graph," in *GLSVLSI*, 2023, pp. 183–187.

[3] R. Ben-Hur *et al.*, "SIMPLER MAGIC: Synthesis and mapping of in-memory logic executed in a single row to improve throughput," *TCAD*, vol. 39, no. 10, pp. 2434–2447, 2020.

[4] F. Wang *et al.*, "STAR: Synthesis of stateful logic in RRAM targeting high area utilization," *TCAD*, vol. 40, no. 5, pp. 864–877, 2021.

[5] Gurobi Optimization, LLC., "Gurobi optimizer reference manual," 2021.

[6] M. Soeken *et al.*, "The EPFL logic synthesis libraries," 2022, arXiv:1805.05121v3.

[7] L. Amarù *et al.*, "The EPFL combinational benchmark suite," in *IWLS*, 2015.