FineMap: A Fine-grained GPU-parallel LUT Mapping Engine

Tianji Liu
CSE Department, CUHK
tjliu@cse.cuhk.edu.hk

Lei Chen, Xing Li, Mingxuan Yuan
Huawei Noah's Ark Lab, Hong Kong SAR
{lc.leichen, li.xing2, yuan.mingxuan}@huawei.com

Evangeline F.Y. Young
CSE Department, CUHK
fyyoung@cse.cuhk.edu.hk

***Abstract*—Lookup-table (LUT) mapping is an indispensable step in FPGA design flows, and also serves as a building block in many technology-independent optimization algorithms. Therefore, it is crucial to accelerate LUT mapping in order to satisfy the demand for synthesizing high-quality, large-scale VLSI designs. Previous work on GPU LUT mapping suffers from low speedup due to limited degree of parallelism. In this paper, we propose an ultra-fast GPU-parallel LUT mapping engine named FineMap, which is composed of a novel fine-grained mapping phase with a high degree of parallelism, a parallel cut expansion phase and a parallel timing analysis pass. The mapping phase is enhanced by specifically tailored cut evaluation and memory management algorithms for GPUs that enable fast mapping of large circuits with limited GPU memory. Experiments show that compared with the high-performance mapper implemented in ABC, FineMap achieves $128.7\times$ speedup with better quality in terms of area on large benchmarks.**

I. Introduction

Technology mapping is a crucial step in logic synthesis. During technology mapping, a circuit represented in a technology-independent format (e.g., And-Inverter Graph, AIG) is transformed into a technology-dependent representation in which the circuit is realized by standard cells (for ASICs) or k-input lookup-tables (k-LUTs, for FPGAs). In particular, the algorithm for mapping into k-LUTs (or *LUT mapping*) has been widely adopted in modern logic synthesis flows as a fundamental building block, besides its typical usage as a mapper in an FPGA design flow. For example, in some logic optimization algorithms [1], [2], LUT mapping acts as a technology-independent optimizer in which the costs are computed from the technology-independent point of view, making it also applicable in ASIC design flows.

With the increasing scale and complexity of VLSI designs, logic synthesis algorithms including LUT mapping have become highly time-consuming [3]. Meanwhile, in order to obtain good quality designs, there emerges high-effort logic synthesis flows in which LUT mapping plays an important role and can be invoked many times. For instance, in [2], optimization algorithms including LUT mapping are repeatedly applied to a circuit for incrementally improving its area. In [4], LUT mapping is used as an evaluator to guide the exploration of logic transformation sequences for FPGA synthesis. Such extensive and frequent usage makes LUT mapping an important candidate to be accelerated, which enables synthesizing large-scale designs by high-effort algorithms within practical time budgets.

There have been previous works on accelerating LUT mapping through parallelization [4]–[6], which can be classified into two categories: partition-based and data-parallel mapper. In a partition-based mapper, the circuit is split into several parts, and each part is mapped individually into a small LUT network. The final mapped network is obtained by reassembling all the small networks. Although the partition-based methods are flexible, they suffer from both limited acceleration and quality: in order to achieve high speedup, the number of partitions needs to be sufficiently large, but this in turn leads to quality degradation, since the optimization opportunities during mapping are restricted in small local subgraphs [4]. Moreover, the overhead of partitioning and reassembling becomes more significant with the increasing number of partitions.

A more promising paradigm for accelerating LUT mapping is data-parallelism. This is because data-parallel algorithms can be efficiently executed on GPUs which are powerful massively parallel processors, and we have witnessed many successful applications of GPUs in the field of EDA [7]–[9]. A typical GPU-parallel algorithm relies on many threads and each thread usually processes a small sub-problem of a constant size (e.g., one element in an array, or one node in a graph). The design of GPU architecture enables tens of thousands of threads to be scheduled and executed concurrently, which is infeasible on multi-core CPUs due to large threading overhead. However, it is non-trivial to design data-parallel algorithms that can exploit the full power of GPUs. For example, the GPU LUT mapper proposed in [5] only makes use of a coarse-grained parallelization opportunity that originates from the AIG structure and ignores the opportunities reside in the computation patterns of mapping, so there is still room for obtaining further speedup. We will further discuss [5] in Section II-C.

This paper presents a fully GPU-accelerated data-parallel LUT mapping engine named FineMap. FineMap is equipped with a novel fine-grained parallel mapping algorithm, as well as parallel cut expansion and timing analysis which are important components of a LUT mapper. Specifically, the contributions of this paper are:

- An ultra-fast, highly parallel mapping algorithm exploiting parallelization opportunities reside in both the global AIG structure and the computation patterns of processing a single AIG node.
- A new cut evaluation strategy that guarantees determinism and improves the efficiency of mapping.
- A dynamic memory management scheme enabled by a specifically designed GPU-based memory pool for the storage of priority cut sets.
- Parallel cut expansion and timing analysis algorithms for fully accelerating the overall mapping flow.

We conduct extensive experiments to evaluate the performance of the proposed LUT mapping engine. Compared with the high-performance LUT mapper implemented in the academic logic synthesis tool ABC [10], FineMap achieves an averaged overall acceleration ratio of $128.7\times$ with slightly better quality-of-results.

The rest of the paper is organized as follows. Section II introduces some backgrounds and reviews the previous work on GPU LUT mapping. Section III describes our LUT mapping engine. Section IV shows the experimental results. Section V concludes the paper.

II. Preliminaries

A. Backgrounds

A *Boolean network* is a directed acyclic graph (DAG) representation of a combinational circuit in which each node corresponds to a Boolean function and edges represent input/output signals of the nodes. The *transitive fanins* and *transitive fanouts* of a node are the predecessors and successors of the node respectively. In particular, the direct predecessors and successors of a node are called *fanins* and *fanouts* respectively. The *primary inputs* (PIs) and *primary outputs* (POs) of a Boolean network are the sources and sinks of the DAG respectively. The logic function at each node, taking its fanin signals as inputs, is denoted as the *local function* of the node.

An *And-Inverter Graph* (AIG) is a Boolean network where all nodes are two-input AND gates with optionally complemented fanin signals. A *k-LUT network* (or simply *LUT network*) is a Boolean network in which every node represents an arbitrary logic function with at most k inputs. In an AIG or a LUT network, the *delay of a node* is the length of the longest path from any PI to this node, which can be computed recursively as $d_N(n) = 1 + \max_{n' \in \text{fanins}(n)} d_N(n')$, where 1 represents the signal delay at this AND/LUT node. The *delay/level of the network* is the maximum delay of all POs.

For a node n in a Boolean network, a *cut* of n is a set of nodes in the network such that any path from a PI to n passes through at least one node in the set. The *logic cone* associated with a cut of n is a set of nodes including n, and the intersection of all the transitive fanins of n and all the transitive fanouts of the cut nodes.

LUT mapping transforms a Boolean network of a particular representation into a LUT network. In this paper, we consider LUT mapping from AIGs but the proposed approach can also be applied to other representations. During LUT mapping, a *representative cut* for each AIG node will be computed and a subset of the nodes will be *selected in the mapping*, such that the associated logic cones of their representative cuts cover all the logic (i.e., non-PI nodes) in the AIG. There is a one-to-one correspondence between a LUT and the associated logic cone of the representative cut of an AIG node, so we will use these two concepts interchangeably. An illustration of LUT mapping is shown in Figure 1.

B. Overview of LUT Mapping

In this section, we briefly introduce one of the state-of-the-art open-source LUT mappers which are implemented in ABC [10] (command `if`, referred to as ABC LUT mapper) as an overview of LUT mapping. We use ABC LUT mapper as a template for constructing our GPU LUT mapping engine. Although many algorithms for LUT mapping have been proposed over the years [11]–[13], in general their high-level ideas are similar, so our methods can be extended to new mapping flows without much effort.

ABC LUT mapper consists of two kinds of phases that are applied repeatedly in the framework: mapping, and cut expansion. The overall flow is shown in Figure 2.

Both the mapping and cut expansion phases update the representative cut of each AIG node in a topological order, such that the quality of the derived LUT network can be incrementally improved. The mapping phases constitute the major part of the mapper, in which each node is assigned a new representative cut chosen from a set of candidates. In particular, a mapping phase can be delay-oriented or area-oriented. Delay-oriented mapping phases are applied first in the mapping flow to reduce the delay of the LUT network, followed by area-oriented mapping passes for reducing the area without affecting the network delay. Both types of the mapping phase share the same algorithmic framework, and the major difference lies in the criteria for representative cut selection. The cut expansion phase expands the representative cut of the selected nodes towards PIs and encourages LUT sharing, which further reduces the area of the LUT network.

After each mapping or cut expansion phase, a timing analysis pass is performed in which the required time of each AIG node is updated. The required time of a node specifies the maximum feasible delay of this node in the derived LUT network such that the delay of the current LUT network will not be increased. This information is useful

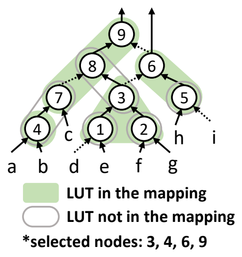

Fig. 1: An example of LUT mapping.

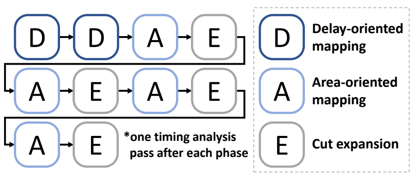

Fig. 2: The overall flow of ABC LUT mapper.

in subsequent phases where the nodes on the non-critical paths can have representative cuts with larger delays (bounded by the required time) but smaller contributions of area to the LUT network.

C. GPU-parallel LUT Mapping

Similar to other EDA algorithms, LUT mapping cannot be trivially GPU-parallelized by dividing the circuit into many parts and performing computation for each part concurrently, because the processings of different sub-circuits are interdependent. For instance, the computation of a node's candidate representative cut relies on the candidate cuts of the node's two fanins. This is just one of the many examples of interdependence in parallel LUT mapping, and we will see more in Section III.

The previous work [5] proposed a GPU-parallel mapping phase that follows the paradigm of level-wise parallellism [9]: the AIG nodes with delay i form a batch, where all the nodes in the batch are processed concurrently and the batches are processed one by one (i.e., batch $i+1$ follows i, etc.). However, the internal mapping procedures (cut enumeration and evaluation, representative cut selection) for a particular node are done sequentially by one thread, which ignores more fine-grained parallelization opportunities. Moreover, the cut expansion phase and the timing analysis pass are not parallelized in [5]. As will be shown in our experiments and analyses, these two drawbacks significantly restrict the speedup of [5].

III. GPU LUT Mapping Engine

This section introduces our GPU LUT mapping engine. First, we elaborate on our fine-grained parallel mapping algorithm. We then present novel solutions for resolving problems encountered in cut metric computation and memory management during parallel mapping. Finally, we introduce the parallel cut expansion phase and timing analysis pass.

A. Fine-grained Parallelism-enabled Mapping Phase

1) Backgrounds: During the mapping phase, a set of at most $C+1$ cuts (referred to as *priority cuts*, denoted by $P(n)$) for each AIG node n is computed [13]. The priority cuts of n includes its trivial cut $\{n\}$, and the best C cuts selected from a set of candidate cuts $E(n)$ computed by cut enumeration:

$$E(n) = \{u \cup v : u \in P(n_0), v \in P(n_1), |u \cup v| \leq k\}, \qquad (1)$$

which exhaustively examines every possible pair of cuts in the priority cut sets of the two fanins of n respectively. In particular, the best cut in $E(n)$ is assigned to be the representative cut of n.

The cut ranking is determined by certain criteria that are based on combinations of three cut metrics: delay, area-flow and exact area [13]. The delay of a cut $d_C(\cdot)$ is the delay of the corresponding LUT in the derived LUT network (if this LUT is selected in the mapping), computed as

$$d_C(c) = 1 + \max_{n' \in c} d_C(RC(n')),$$

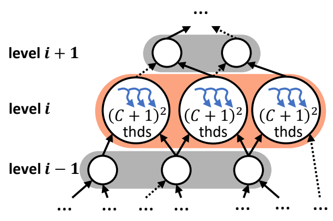

Fig. 3: Illustration of the fine-grained parallel mapping phase.

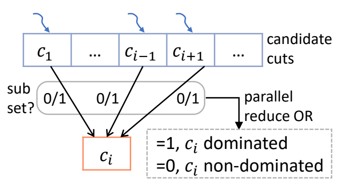

Fig. 4: Illustration of the parallel dominance checking.

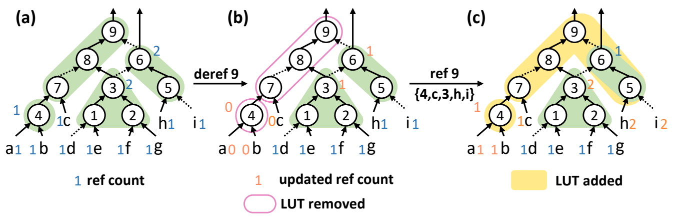

Fig. 5: (a) Illustration of the reference counter. (b) Dereferencing node 9. (c) Referencing node 9 with cut $\{4, c, 3, h, i\}$.

where $RC(\cdot)$ denotes the representative cut of a node. The area-flow of a cut c rooted at node n estimates the global area of the current sub-LUT network rooted at n, and can also be computed efficiently in a way similar to cut delay. Exact area is instead an area metric from a local viewpoint, and we will introduce exact area in Section III-B. For more related details about the cut metrics, we refer the readers to [13]. Roughly speaking, in a delay-oriented mapping phase, delay is the major criterion and area serves as a tie-breaker, whereas in an area-oriented mapping phase, the criteria order is reversed.

2) Approach: We describe our proposed fine-grained parallel algorithm for the priority cut set computation of *one* node in the mapping phase. For our GPU LUT mapping phase, the processing of *all* nodes follows the level-wise parallel paradigm introduced in Section II-C. An illustration for the overall mapping phase framework is shown in Figure 3. Although a sequential algorithm can be easily formulated for computing the priority cuts of a node, it is non-trivial to do this in parallel, since the threads need to jointly perform a single task.

Algorithm 1 shows the per-thread pseudocode of our algorithm. We assign one thread for the enumeration (Equation 1) and metric computation of a unique candidate cut (line 1-7), and thus $(C+1)^2$ threads are used in total (the largest possible value for $|E(n)|$). Then, we need to tackle two problems. First, after the cut evaluations are individually done by each thread, the best C cuts need to be collected. Intuitively, parallel sorting can be applied, but experimentally we found it induces a large overhead. Instead, our approach relies on the more efficient parallel reduction: we find the best cut C times, and in each iteration, parallel reduction selects the best cut from the candidate cuts (line 10-11). The obtained cut is masked out (line 17) so that it will not be selected again in subsequent iterations.

The second issue is that if there is a pair of candidate cuts such that one is a subset of (i.e., dominates) the other, we do not want the dominated cut to appear in the priority cut set, because the local function does not depend on some of the nodes in the dominated cut. To achieve this, we check the dominance of all the other cuts against the best cut once it is selected (line 13), and use parallel reduction (line 15) for aggregating the checking results to decide whether to keep or discard the selected best cut, as illustrated in Figure 4.

B. Fitting Local Area Evaluation into Parallel Mapping

1) Reference Counter and Exact Area: In ABC LUT mapper, each AIG node is assigned a reference counter representing the number of fanouts of the LUT rooted at this node in the derived LUT network. A node is selected in the mapping if it has a positive reference count and not selected otherwise. Figure 5(a) provides an illustration.

The local area metric, namely the exact area of a cut c rooted at a node n is computed along with a reference or dereference procedure in which n is added to or removed from the current mapping with c as the representative cut. During the (de)reference procedure, the reference counts of some nodes will be updated by a depth-first traversal. For example, in Figure 5(b), dereferencing node 9 updates the reference counts of node 3, 4, 6 and PI a, b, c. Note that whenever a node n' with zero reference count is encountered, it means that the logic of n' is no longer useful if n is removed (or in the referencing case, is needed if n is added), and thus n' should be recursively (de)referenced. Interested readers are referred to [14] for more details about (de)referencing. The exact area of a cut c is the number of LUTs traversed during (de)referencing. These LUTs are the ones dedicated to driving n in the mapping.

2) Race-free and Deterministic Exact Area Computation: In parallel mapping, since the exact areas of different candidate cuts are computed concurrently, data races could happen where different threads attempt to modify the reference count of the same node, which leads to incorrect results or non-determinism. For example, suppose that in the situation of Figure 5(b), the exact area of two candidate cuts of node 9 are computed concurrently by referencing. Suppose node 4 appears in both cuts, so both threads attempt to read and increment the reference count of 4. It could happen that one of the threads reads 1, if the other thread has already incremented the counter. Hence, this thread will not traverse the LUT rooted at 4 and its computed exact area will be smaller than the correct value.

To address this issue, we record a separate copy of the changed reference counts in the thread-local memory. The original reference counts in the global memory are read-only and kept static in the state before Algorithm 1 starts. When the thread attempts to change the reference count of a node, it first checks whether there is a record of this node in its local memory. If the record exists, it directly modifies the value. Otherwise, it retrieves the reference count from

Algorithm 1 Fine-grained Parallel Mapping (Per-thread)

```
Input: AIG node $n$, max cut size $k$, thread id $tid$
Updates: priority cut set $P(n)$, representative cut $RC(n)$
 1: $c_0 \leftarrow getCut(P(fanin_0(n)), tid\ /\ (C+1))$
 2: $c_1 \leftarrow getCut(P(fanin_1(n)), tid\ \%\ (C+1))$
 3: $c \leftarrow c_0 \cup c_1$
 4: $valid \leftarrow (c_0 \neq \emptyset$ and $c_1 \neq \emptyset$ and $|c| \leq k)$, $selected \leftarrow false$
 5: if $valid$ then
 6:     Compute the delay, area-flow or exact area of $c$
 7:     if $d_C(c) > t_{req}(n)$ then $valid \leftarrow false$
 8: if $tid = 0$ then $P(n) \leftarrow \{RC(n)\}$
 9: $sync\_threads()$                ▷ synchronizing the $(C+1)^2$ threads
10: while $|P(n)| < C$ do
11:     $c_b \leftarrow reduceBest(c, valid$ and $!selected)$     ▷ parallel reduction
12:     if $c_b = \emptyset$ then break        ▷ no valid and non-selected cut left
13:     if $valid$ and $c \neq c_b$ then $f \leftarrow isSubset(c, c_b)$
14:     else $f \leftarrow false$
15:     $dom \leftarrow reduceOr(f)$                ▷ parallel reduction
16:     if $tid = 0$ and $!dom$ then $P(n) \leftarrow P(n) \cup c_b$
17:     if $c = c_b$ then $selected \leftarrow true$            ▷ mask out $c_b$
18:     $sync\_threads()$
19: if $tid = 0$ then
20:     $P(n) \leftarrow P(n) \cup \{n\}$                ▷ add the trivial cut
21:     $RC(n) \leftarrow$ the first (best) cut in $P(n)$
```

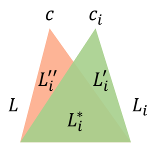

Fig. 6: An illustration of Lemma 1.

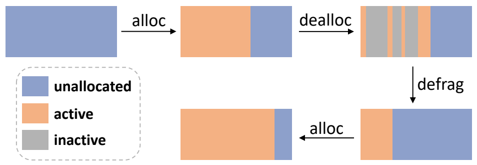

Fig. 7: Schematic of GPU memory pool.

the global memory and saves the changed value into its local memory. In this way, the reference counts can be independently manipulated by different threads without data race, and the correctness and determinism of exact area computation are guaranteed. An additional parallel procedure is applied after Algorithm 1 during the processing of each level, where the old representative cuts are dereferenced and the new ones are referenced for the nodes in the level, so that the reference counts in the global memory are refreshed and up-to-date.

3) A New Strategy for Local Area Evaluation: In the scenario of sequential computation, when computing the exact area of a candidate cut c' of node n during the mapping phase, the current representative cut c of n is dereferenced first, followed by referencing n with the tentative representative cut c'. We call this strategy deref-ref. In the GPU mapping scenario, the deref-ref strategy can be problematic: some cuts may have very large exact areas (e.g., hundreds), so the overall runtime of Algorithm 1 can be very long since the threads processing such cuts become the bottleneck of the parallel procedure, even though most of the threads run quite fast. In extreme cases, the local memory of such threads might be depleted due to having recorded too many DFS frontier nodes and changed reference counts during (de)referencing. To resolve this problem, we propose a new strategy for the local area evaluation of cuts, which is based on the following observation:

Observation 1. Different candidate cuts of a node tend to differ a little from each other (e.g., $\{4, c, 3, 6\}$ and $\{4, c, 3, h, i\}$ of node 9 in Figure 5), so the traversed LUTs usually have a large portion in common during exact area computation using the deref-ref strategy.

In our strategy, rather than dereferencing the representative cut c first and then reference a candidate cut c' as in the sequential scenario, we first reference c' and then dereference c, and use the difference between the number of LUTs traversed in referencing and that in dereferencing as the local area metric of c' (referred to as ref-deref). The ref-deref strategy improves runtime efficiency as well as reduces thread-local memory usage of cut evaluation, because only the non-overlapping parts of the traversed LUTs when computing the exact area of c' (using the deref-ref strategy) are traversed in the ref-deref case, which usually have small sizes according to Observation 1. This is illustrated in Figure 6 and formalized as Lemma 1.

Lemma 1. *Consider a node that is selected in the mapping with representative cut c and a candidate cut c_i. Let L and L_i respectively be the sets of LUTs traversed during dereferencing c and referencing c_i using the deref-ref strategy. Then, the sets of LUTs traversed during referencing c_i and dereferencing c using the ref-deref strategy are $L_i' := L_i \setminus L$ and $L_i'' := L \setminus L_i$ respectively.*

Proof. According to the description in Section III-B1, a LUT is traversed during referencing if and only if (1) it is not in the mapping before referencing, and (2) it needs to be selected in the mapping after referencing. Hence, when referencing c_i, the LUTs in L are already selected in the mapping and thus only those in L_i but not in L, i.e., LUTs in L_i' will be traversed. A similar argument proves the other side of the lemma about dereferencing c. □

Although the local area metric computed using the ref-deref strategy is different from the exact area computed using the deref-ref strategy, the candidate cut ranking is not affected, as shown below.

Proposition 2. *The candidate cut ranking computed by the ref-deref strategy is the same as that computed by the deref-ref strategy.*

Proof. Suppose we have a set of candidate cuts $\{c_i\}$. For an arbitrary candidate cut c_i, we use the notations in Lemma 1 and Figure 6, and define $L_i^* := L \cap L_i$. According to Lemma 1, the local area metric of c_i is computed as

$$m_i = |L_i'| - |L_i''| = (|L_i'| + |L_i^*|) - (|L_i''| + |L_i^*|) = |L_i| - |L|.$$

Note that $|L_i|$ is the exact area of c_i computed using the deref-ref strategy and $|L|$ is a constant. Hence, the ranking of $\{c_i\}$ will be the same as that computed by the deref-ref strategy. □

C. Dynamic Memory Management by GPU Memory Pool

High memory consumption is a known issue for LUT mapping [13], which is mainly due to the storage of the priority cut sets during the mapping phase. Statically allocating the cut sets once for all nodes simply leads to out-of-memory on GPUs. Hence, a dynamic memory management scheme is needed to control memory usage.

1) GPU Memory Pool: We design a GPU-based memory pool for efficiently managing the cut set memory during the mapping phase. The memory pool contains many entries (memory fragments of a constant size). Each entry is in one of the three states at any time: unallocated, active or inactive. We say that an entry is allocated if it is active or inactive. Active means that the entry is currently in use, and inactive means that the entry can be freed. We use a 0-1 binary array (denoted as the activeness mask) to indicate whether an entry is active (1) or inactive/unallocated (0).

There are three basic functionalities implemented by efficient GPU-parallel algorithms: batched allocation, batched deallocation and defragmentation. The schematic of the three functionalities is shown in Figure 7. A principle of our GPU memory pool is that the allocated memory entries are always consecutive. Following this principle, we will assign a consecutive memory segment located at the beginning of the unallocated part during batched allocation. For batched deallocation, we simply mark the corresponding locations in the activeness mask as zero in parallel. Hence, allocation and deallocation can both be done in $O(1)$ time. Defragmentation is to remove all inactive entries so that there will be more unallocated memory ready to be reused. Essentially, a parallel prefix-sum of the activeness mask is computed to obtain the new locations of the active entries, followed by concurrent memory copying.

2) Dynamic Cut Set Memory Management: Intuitively, to reduce memory usage, the lifetime of cut sets should be as short as possible, i.e., a cut set should be allocated just before it is used, and deallocated once it is no longer in use. Hence, for node n, the level for allocating its cut set is the level of n, and the level for deallocation is the maximum level of the fanout nodes of n, and thus the (de)allocation node lists for each level can be pre-computed accordingly. Batched allocation and deallocation of cut sets using the memory pool are respectively invoked once during the processing of each level in the mapping phase. In our strategy, defragmentation is only invoked before allocation when the unallocated entries are insufficient. One can also perform defragmentation more frequently and a smaller memory pool can be used to further reduce memory usage.

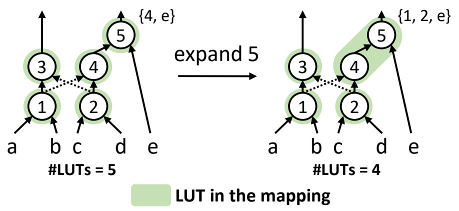

Fig. 8: An example of cut expansion.

D. Parallel Cut Expansion Phase

In the cut expansion phase, the representative cuts of the selected nodes in the mapping are improved by the following procedure [13]. For a node n, we repeatedly find a leaf n' in the representative cut of n and replace it by its two fanins n'_0, n'_1 if (1) n' exclusively drives n and (2) n'_0, n'_1 are shared by other LUTs in the mapping. Figure 8 shows an example where the representative cut of node 5 ($\{4, e\}$) is updated to $\{1, 2, e\}$ by expanding the leaf node 4. The effect of doing so is that some LUTs dedicated to driving n are merged into one, and thus the total area of the LUT network is reduced.

In the parallel settings, we concurrently compute all the improved representative cuts of the selected nodes. The leaf selection relies on manipulating the reference counts, so the strategy in Section III-B2 is applied for avoiding data race. Notably, the real update of the representative cuts are performed in level-wise parallel along with delay recomputation, and any improved cut whose delay exceeds the node's required time will be skipped so the delay of the mapped network will not increase after the update.

E. Parallel Timing Analysis Pass

In the timing analysis pass, the required times of AIG nodes are computed based on the following formula [13]

$$t_{req}(n) = \min_{n' \in ref(n)} (t_{req}(n') - 1),$$

where $ref(n)$ denotes the fanout nodes of n in the derived LUT network and the 1 represents the signal delay at the LUT of n'. The required time of a PO is set as the delay of the LUT network. It can be seen that the required time of a node is essentially propagated from its fanout nodes, and thus a reversed topological order sweeping is sufficient for computing the required times for all nodes.

Our parallel timing analysis is performed in the reversed level-wise parallel fashion (i.e., from high level to low level) in which the nodes in the same level concurrently propagate their required times to the leaves in their representative cut. Since different nodes may propagate their required times to a common leaf simultaneously, the min operation is applied atomically to avoid data race.

IV. Experimental Results

A. Set-up

The proposed GPU LUT mapping engine FineMap is implemented in CUDA C/C++, and the experiments are performed on an Ubuntu 18.04 server with Intel Xeon Silver 4114 CPU and NVIDIA GeForce RTX 3090 GPU with 24 GB dedicated DRAM. The benchmarks are selected from the EPFL Combinational Benchmark Suite [15] and the OpenCores designs in the IWLS 2005 Benchmarks [16], consisting of arithmetic, control and random circuits. The original benchmarks are too small to showcase the acceleration by parallelization, so they are enlarged following the approach in [9]. The statistics of the benchmarks are shown in Table I.

We mainly compare our algorithm with the ABC LUT mapper [13]. The parallel LUT mappers proposed in the previous works that have been mentioned in Section I are all non-open-sourced, and it is not fair and meaningful to compare with the reported runtime in their papers because their experimental set-ups (overall mapping flow, benchmarks, the hardware used, etc.) are quite different from ours. Still, we would like to compare with the GPU LUT mapping [5] (referred to as FPL'10) in order to analyze the effectiveness of individual components in our proposed engine. To this end, we re-implement their mapping function with inter-node parallelism and without our proposed fine-grained parallelism (referred to as the FPL'10 mapping function), and set up an FPL'10 mapping phase by replacing our mapping function (Algorithm 1) in our GPU mapping phase with the FPL'10 mapping function.

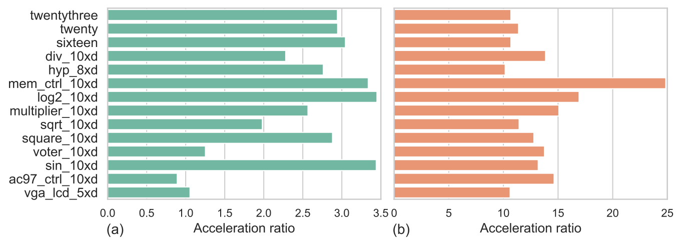

Fig. 9: (a) Acceleration ratios of the fine-grained mapping function vs. the inter-node-parallel-only mapping function. (b) Acceleration ratios of FineMap vs. the estimated full-flow time of FPL'10.

The maximum number of LUT inputs k and the number of priority cuts per node C is respectively set as 6 and 8 in all the experiments. All the generated results passed combinational equivalence checking.

B. Full-flow Comparison with ABC

To ensure a fair comparison, we follow the phase order shown in Figure 2 for FineMap. The reported runtime is the algorithm execution time which does not include the file I/O time. As shown in Table I, FineMap achieves 128.7× acceleration on average with slightly better quality-of-results in terms of area, when compared against ABC. The delays of the generated LUT networks by FineMap are identical to those by ABC mapping. Note that FineMap runs deterministically which is enabled by our algorithm design.

C. Comparison with FPL'10

In order to precisely analyze the efficiency contributed by the proposed fine-grained parallel mapping, we compare the total runtime of our mapping function (Algorithm 1) with the re-implemented FPL'10 mapping function. The runtime of other parts in the mapping phase such as update of reference counters (see Section III-B2) are not included. As can be seen in Figure 9(a), the fine-grained parallel mapping function obtains 2.3× speedup over the inter-node-parallel-only counterpart on average, which indicates that our fine-grained parallelism is able to further accelerate level-wise parallel mapping while retaining good result quality.

As mentioned in Section II-C, the full-flow LUT mapping acceleration of FPL'10 can be limited due to the sequential cut expansion and timing analysis. To demonstrate this point, we compare the runtime of FineMap against the estimated full-flow time of FPL'10, which is calculated by summing up the time of FPL'10 mapping phase and the time of cut expansion and timing analysis in the ABC mapper. Under this setting, the averaged speedup of FineMap is 13.2× over FPL'10, as shown in Figure 9(b). This is in fact an underestimation, because the GPU-host memory copying time between different phases/passes is not counted for the FPL'10 full-flow. To conclude, it is important to accelerate cut expansion and timing analysis in order to achieve a high overall speedup.

TABLE I: Results of full-flow LUT mapping.

Benchmarks	AIG Statistics		ABC `if`			FineMap (Ours)		
	#AIG Nodes	Levels	#LUTs	Levels	Time	#LUTs	Levels	Time
twentythree	23339737	176	6659071	36	2322.8	6646639	36	71.3
twenty	20732893	162	5929939	33	1888.4	5927717	33	49.4
sixteen	16216836	140	4486446	29	1358.3	4471454	29	37.1
div_10xd	58620928	4372	22559744	864	4100.5	22793216	864	23.2
hyp_8xd	54869760	24801	11392768	4194	3862.1	11461888	4194	23.5
mem_ctrl_10xd	47960064	114	12386304	25	2560.6	12402688	25	11.5
log2_10xd	32829440	444	8200192	77	2462.3	8056832	77	10.6
multiplier_10xd	27711488	274	6054912	53	1869.2	6000640	53	8.4
sqrt_10xd	25208832	5058	5857280	1033	1778.5	5919744	1033	11.1
square_10xd	18927616	250	4080640	50	1358.5	4007936	50	5.8
voter_10xd	14088192	70	2885632	17	760.9	2890752	17	4.2
sin_10xd	5545984	225	1492992	42	401.4	1483776	42	2.2
ac97_ctrl_10xd	14610432	12	2992128	4	441.4	2998272	4	3.2
vga_lcd_5xd	4054752	24	910912	7	191.6	910976	7	1.5
Geomean Ratio			1.000	1.000	128.7	0.998	1.000	1.0

"_$n$xd" means that the benchmark is generated by enlarging the original one using ABC `double` n times.

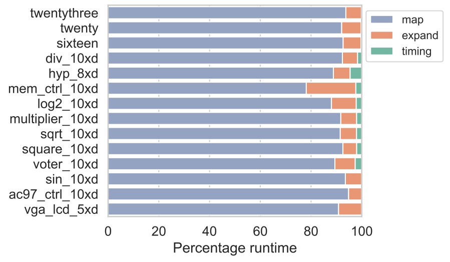

Fig. 10: Runtime breakdown of FineMap.

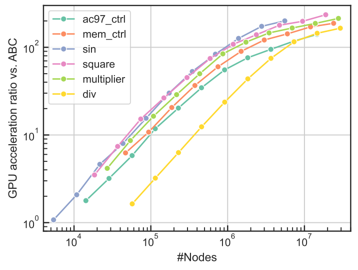

Fig. 11: Acceleration ratios of FineMap on AIGs of different sizes.

D. Analysis on Dynamic Memory Management

In our experiments, we use a constant memory pool size of 6.7 GB for FineMap. In contrast, the naive approach of allocating the cut sets once-for-all requires 39.3 GB for the largest benchmark. We found that the runtime induced by memory management (including allocation, deallocation and defragmentation) occupies less than 3% of the overall GPU mapping runtime on average. Hence, the proposed memory management approach reduces memory usage by at most 5.9× with negligible runtime overhead.

E. Runtime Breakdown

The fractions of the total GPU engine runtime for mapping, cut expansion and timing analysis are shown in Figure 10. The acceleration ratios of these three phases/passes are 138.4×, 101.8× and 323.9× over the ABC counterparts. It can be seen that mapping consists of the major part of the total runtime, which indicates the importance and necessity of our fine-grained parallelism.

F. Scaling Experiments

Figure 11 shows the speedup of FineMap over ABC on benchmarks that are enlarged to different scales. The acceleration ratio increases with the increasing AIG size, because more parallelization opportunities can be exploited. It is worth noting that even for benchmarks with small sizes or large delays (e.g., the leftmost points of sin and div), FineMap is still faster than ABC, which shows the effectiveness of our proposed framework.

V. Conclusion

This paper presents FineMap, an ultra-fast GPU-parallel LUT mapping engine. The major part of the engine is a fine-grained, highly parallel mapping phase constructed by combining level-wise parallelism with a novel parallel algorithm for the processing of a single node. A new strategy for the local area evaluation of a cut and a memory-pool-based memory management method ensure efficient mapping of large designs with moderate memory footprint. Furthermore, high full-flow acceleration of the engine is guaranteed by a parallel cut expansion phase and a parallel timing analysis pass. Experiments show that FineMap obtains an overall acceleration ratio of 128.7× over ABC with slightly better quality-of-results, and demonstrate the effectiveness and importance of individual components in the GPU mapping engine.

References

[1] W. Yang, L. Wang, and A. Mishchenko, "Lazy man's logic synthesis," in *Proc. ICCAD*, 2012, pp. 597–604.
[2] L. Amarú, V. Possani, E. Testa, F. Marranghello, C. Casares, J. Luo, P. Vuillod, A. Mishchenko, and G. De Micheli, "LUT-based optimization for ASIC design flow," in *DAC*, 2021, pp. 871–876.
[3] V. Possani, Y.-S. Lu, A. Mishchenko, K. Pingali, R. Ribas, and A. Reis, "Unlocking fine-grain parallelism for AIG rewriting," in *Proc. ICCAD*, 2018, pp. 1–8.
[4] G. Liu and Z. Zhang, "PIMap: A flexible framework for improving LUT-based technology mapping via parallelized iterative optimization," *ACM TRETS*, vol. 11, no. 4, pp. 1–23, 2019.
[5] D. Chen and D. Singh, "Parallelizing FPGA technology mapping using Graphics Processing Units (GPUs)," in *Proc. FPL*, 2010, pp. 125–132.
[6] C. Shen, Z. Lin, P. Fan, X. Meng, and W. Qian, "Parallelizing FPGA technology mapping through partitioning," in *FCCM*, 2016, pp. 164–167.
[7] Y. Lin, S. Dhar, W. Li, H. Ren, B. Khailany, and D. Z. Pan, "DREAMPlace: Deep learning toolkit-enabled GPU acceleration for modern VLSI placement," in *Proc. DAC*, 2019, pp. 1–6.
[8] L. Liu, B. Fu, M. D. Wong, and E. F. Young, "Xplace: an extremely fast and extensible global placement framework," in *Proc. DAC*, 2022, pp. 1309–1314.
[9] T. Liu and E. F. Young, "Rethinking AIG resynthesis in parallel," in *Proc. DAC*, 2023, pp. 1–6.
[10] R. K. Brayton and A. Mishchenko, "ABC: An academic industrial-strength verification tool," in *Proc. CAV*, 2010.
[11] D. Chen and J. Cong, "DAOmap: a depth-optimal area optimization mapping algorithm for FPGA designs," in *Proc. ICCAD*, 2004, pp. 752–759.
[12] A. Mishchenko, S. Chatterjee, and R. K. Brayton, "Improvements to technology mapping for LUT-based FPGAs," *IEEE TCAD*, vol. 26, no. 2, pp. 240–253, 2007.
[13] A. Mishchenko, S. Cho, S. Chatterjee, and R. Brayton, "Combinational and sequential mapping with priority cuts," in *Proc. ICCAD*, 2007, pp. 354–361.
[14] H. Riener, W. Haaswijk, A. Mishchenko, G. De Micheli, and M. Soeken, "On-the-fly and DAG-aware: Rewriting Boolean networks with exact synthesis," in *Proc. DATE*, 2019, pp. 1649–1654.
[15] L. Amarú, P.-E. Gaillardon, and G. De Micheli, "The EPFL combinational benchmark suite," in *IWLS*, 2015.
[16] C. Albrecht, "IWLS 2005 Benchmarks," https://iwls.org/iwls2005/benchmarks.html, 2005.