ESPRESSO-GPU: Blazingly Fast Two-Level Logic Minimization

Hitarth Kanakia*, Mahdi Nazemi*, Arash Fayyazi, and Massoud Pedram
Department of Electrical & Computer Engineering, University of Southern California, Los Angeles, CA, USA
{kanakia,mnazemi,fayyazi,pedram}@usc.edu

***Abstract*—Two-level logic minimization has found applications in new problems such as efficient realization of deep neural network inference. Important characteristics of these new applications are that they tend to produce very large Boolean functions (in terms of the supporting variables and/or initial sum of product representation) and have don't-care-sets that are much larger in size than the on-set and off-set sizes. Applying conventional single-threaded logic minimization heuristics to these problems becomes unwieldy. This work introduces ESPRESSO-GPU, a parallel version of ESPRESSO-II, which takes advantage of the computing capabilities of general-purpose graphics processors to achieve a huge speedup compared to existing serial implementations. Simulation results show that ESPRESSO-GPU achieves an average speedup of 97x compared to ESPRESSO-II.**

I. Introduction

Although logic synthesis has grown increasingly sophisticated, building complex optimization scenarios that span from regular fabrics of restricted depth up to multilevel and multi-valued logic realizations [1], two-level logic minimization still plays a central role as a key procedure in more complex logic optimization packages. The optimization of Boolean functions with tens of thousands of product terms takes hours to complete with existing two-level logic minimization methods. However, applications have arisen that produce large sparse Boolean functions with a large number of input variables and product terms. Examples of such applications are found in [2] and [3] where the problem of efficient processing of neural networks is formulated as a two-level logic minimization problem, which is optionally followed by multi-level logic minimization.

For example, the approach presented in [2] first discretizes input and output activations of artificial neurons to binary values while training a neural network. Next, during inference, it applies training data points to the neural network and for each neuron, records values of the binary inputs and outputs encountered when processing each data point. Finally, it creates an incompletely specified Boolean function for each neuron using the encountered binary inputs and outputs and employs logic synthesis to find a near-optimal realization of each neuron. This, in fact, is equivalent to sampling the algebraic function that represents each neuron and transforming that algebraic function to a Boolean function that approximates it. However, because neurons designed for state-of-the-art neural networks include tens to hundreds of inputs, the input space is huge and the samples only represent a tiny fraction of the input space that matters to the neural network, hence the approximation.

Consider the eighth convolutional layer of the VGG16 neural network [4] trained on the CIFAR-10 dataset [5]. This layer consists of 512 3×3 filters that are applied to an input volume of $4 \times 4 \times 256$. Therefore, the number of inputs to each filter is $3 \times 3 \times 256 = 2,304$ while the number of input patches is $4 \times 4 = 16$. Using the training data to sample the function of each filter, the number of minterms recorded for each filter is at most $16 * 50,000 = 800,000$ where 50,000 is the number of images in the training set. Therefore, to realize this layer using the approach presented in [2], one has to invoke ESPRESSO-II 512 times, once for each Boolean function which represents a filter, where each function has 2,304 inputs and 800,000 minterms. Optimizing Boolean functions in this one layer with ESPRESSO-II would take weeks to complete.

This work introduces ESPRESSO-GPU, a parallel two-level logic minimization heuristic based on ESPRESSO-II [6], which benefits from computing capabilities of general-purpose graphics processing units (GPUs) to achieve a considerable speedup compared to existing serial implementations when optimizing large sparse incompletely specified Boolean functions. In particular, we detail how the EXPAND step of ESPRESSO-II can be parallelized on GPUs to speed up its computations by about two orders of magnitude. Our GPU version is fully parallelized with CUDA [7] and optimized for large sparse incompletely specified Boolean functions.

The remainder of this paper is organized as follows. Section II reviews preliminaries and related work while Section III details the proposed parallelization techniques applied to ESPRESSO-II. After that, Section IV presents experimental results and finally, Section V concludes the paper.

II. Preliminaries & Related Work

This section reviews some basic definitions related to logic minimization, explains different steps of ESPRESSO-II, details its EXPAND step in addition to its internal data representation, and provides background on capabilities of compute unified device architecture (CUDA) for parallelizing computer programs on GPUs.

A. Two-Level Logic Minimization

A Boolean function ff of n input variables $\boldsymbol{x} = [x_1 \dots x_n]$ and m output variables $\boldsymbol{y} = [y_1 \dots y_m]$ is a function

$$ff : B^n \rightarrow Y^m,$$

in which $B = \{0, 1\}$ and $Y = \{0, 1, 2\}$, where each output variable y_i can assume a *don't-care* value 2 in addition to the usual 0 and 1 values. For each output variable y_i, its on-set is defined as $X_i^{\mathrm{ON}} \subseteq B^n$, the set of all input values $\boldsymbol{x}$ such that $ff_i(\boldsymbol{x}) = 1$. Similarly, the off-set of output y_i is defined as $X_i^{\mathrm{OFF}} \subseteq B^n$, the set of all input values $\boldsymbol{x}$ such that $ff_i(\boldsymbol{x}) = 0$. Finally, the don't-care-set is defined as $X_i^{\mathrm{DC}} \subseteq B^n$, the set of all input values $\boldsymbol{x}$ such that $ff_i(\boldsymbol{x}) = 2$.

*Hitarth Kanakia and Mahdi Nazemi contributed equally to this work.

978-3-9819263-5-4/DATE21/©2021 EDAA

A cover $\mathcal{F}$ of the Boolean function *ff* is a set of cubes $\{c^1, \ldots, c^k\}$ that contains every minterm of the on-set of the function. We define the cover of the off-set $\mathcal{R}$ of the function as sets of cubes that contain every minterm of the off-set. Moreover, a cover of the don't-care-set $\mathcal{D}$ is defined as the set of cubes that contain a subset of the minterms of the don't-care-set of the originally specified Boolean function. Obviously, $\mathcal{R}$ = COMPLEMENT($\mathcal{F} \cup \mathcal{D}$), where COMPLEMENT does the function complementation. The goal of two-level logic minimization is to find a minimal cover of *ff*, i.e. a cover $\mathcal{F}$ such that no proper subset of $\mathcal{F}$ is also a cover of *ff*.

ESPRESSO-II [6], which was developed in 1982, is the most popular two-level logic minimization heuristic. ESPRESSO-II is an iterative heuristic which is comprised of seven high-level steps. The first step, COMPLEMENT, calculates $\mathcal{R}$. The second step, EXPAND, expands each (uncovered) implicant of *ff* to a prime implicant and marks all implicants that are covered by the said prime as covered (this step only requires access to $\mathcal{F}$ and $\mathcal{R}$ to perform its calculations). The result is a prime cover of *ff* such that no cube of *ff* contains any other. The third step, IRREDUNDANT_COVER, finds a minimal irredundant cover by using $\mathcal{F}$ and $\mathcal{D}$. The result is a cover of *ff* where no proper subset of it is also a cover. The fourth step, ESSENTIAL_PRIMES, finds essential primes, removes them from $\mathcal{F}$, and puts them in the $\mathcal{D}$ (the essential primes will be added back onto the cover in the end). The fifth step, REDUCE, reduces each implicant to a minimum essential implicant by removing from it minterms that are covered by other implicants of the cover. Next, steps 2, 3, and 5 are repeated until there is no reduction in the number of implicants or literals. Finally, LAST_GASP performs reduction, expansion, and irredundant cover one more time using a different strategy and MAKESPARSE puts essential primes back into the cover and makes the structure of the optimized circuit sparse.

ESPRESSO-II has been extended by some of the prior work such as [8] and [9]. ESPRESSO-MV [8] reduces computational and memory complexities of ESPRESSO-II while it adds support for multiple-valued input, multiple-output functions. SAT-ESPRESSO [9] formulates REDUCE, ESSENTIAL_PRIMES, and IRREDUNDANT_COVER steps of ESPRESSO-II as satisfiability problems and solves them using a SAT solver. SAT-ESPRESSO can achieve 5 – 20 times lower optimization time compared to ESPRESSO-II [9].

This work mainly deals with large sparse incompletely specified Boolean functions where sizes of $\mathcal{F}$ and $\mathcal{R}$ are very small compared to the size of the input space while the $\mathcal{D}$ is enormous. For such functions, among the three steps that are repeated in the main optimization loop of ESPRESSO-II , EXPAND can be executed as is whereas IRREDUNDANT_COVER and REDUCE, which utilize $\mathcal{D}$, must be modified to use only the on-set (as was done in [9]). This results in some loss in the quality of the optimized Boolean function, although the loss tends to be small. Additionally, single-core execution of EXPAND for such large functions may take hours to complete, which effectively makes it impossible to iterate over different designs. The focus of this work is on parallelizing the EXPAND and IRREDUNDANT_COVER, which take most of the optimization time in ESPRESSO-II. Based on our profiling data, we did not find much value in trying to parallelize REDUCE.

EXPAND step: The goal of the EXPAND procedure is to make the cubes of $\mathcal{F}$ prime and remove as many cubes possible from $\mathcal{F}$ without losing on-set coverage. It does so by processing cubes of $\mathcal{F}$ sequentially, expanding each while maximizing the number of cubes that are covered by the expanded cube. Note that cubes of the cover that are contained in the expanded cube are deleted. As a result the final cover is also minimal with respect to single cube containment. The bulk of computations in the EXPAND procedure occur in four *nested* loops (the runtime complexities described in this section assume a single-output function is being optimized by the EXPAND step). Note that initially all cubes of $\mathcal{F}$ are marked as uncovered.

Algorithm 1 Espresso EXPAND

Input:
  $\mathcal{F}$ //Cover of the ON-set
  $\mathcal{R}$ //Cover of the OFF-set
Output:
  $\mathcal{F}_p$ //Prime cover of *ff*
1: $\mathcal{F} = \text{mini\_sort}(\mathcal{F})$
2: $\mathcal{F}_p$ = Empty cover
3: **for each** uncovered cube $c \in \mathcal{F}$ **do** //LOOP-1
4: Mark uncovered cubes of $\mathcal{F}$ as ACTIVE and INFEASIBLE
5: **do** //LOOP-2
6: feasible_count = 0
7: $c^+ = \text{essen\_raise}(c, \mathcal{R})$
8: $\mathcal{F}$ = distill_cubes($\mathcal{F}$, ACTIVE)
// LOOP-3.1 and LOOP-4.1: find all feasibly covered cubes
9: **for each** uncovered cube $p \in \mathcal{F}$ **do** //LOOP-3.1
10: **if** p is covered by c^+ **then** Mark p as COVERED and INACTIVE
11: **else**
12: **if** feasibly_covered($c^+, p, \mathcal{R}$) **then** //LOOP-4.1 (loop is inside the function call)
13: Mark p as FEASIBLE
14: feasible_count = feasible_count + 1
15: Record lowering_set[p] //This is the set of literals that cannot be raised any longer if c^+ is expanded to minimally cover p
16: **else**
17: Mark p as INFEASIBLE and INACTIVE
18: **end if**
19: **end if**
20: **end for**
21: $\mathcal{F}$ = distill_cubes($\mathcal{F}$, FEASIBLE)
// LOOP-3.2 and LOOP-4.2: count feasibly covered cubes that are disjoint from the lowering set of a given feasibly covered cube
22: **for each** FEASIBLE cube $p \in \mathcal{F}$ **do** //LOOP-3.2
23: count[p] = calc_disjoint_cnt(lowering_set[p], $\mathcal{F}$) //LOOP-4.2 (loop is inside the function call)
24: **end for**
25: $p^* = \arg\max_p \text{count}[p]$
26: c^{++} = c^+ expanded to minimally cover p^*
27: Mark cubes in $\mathcal{F}$ which are covered by c^{++} as INACTIVE
28: $c = c^{++}$
29: **while** feasible_count > 0
30: **while** there exists any ACTIVE cubes of $\mathcal{F}$ **do** //We may still drop literals from c in order to make it PRIME
31: Apply the MINI strategy of raising c //This strategy raises part of c that is common in the largest number of cubes of $\mathcal{F}$
32: **end while**
33: Mark c as PRIME and INACTIVE
34: Add c to $\mathcal{F}_p$
35: **end for**
36: return $\mathcal{F}_p$

At the beginning of the EXPAND step, `mini_sort` is applied on $\mathcal{F}$ to sort the minterms in the order of being less likely to be covered by the expansion of other cubes. Next, LOOP-1 iterates over each uncovered cube c in the previously sorted order of $\mathcal{F}$, and therefore, takes $\mathcal{O}(|\mathcal{F}|)$ to complete. In every iteration we get an uncovered cube c and convert it into a prime implicant. Next, we mark the cubes of $\mathcal{F}$ and $\mathcal{R}$ with appropriate flags and only use a subset of the cubes from these

Algorithm 2 feasibly_covered

Input:
c //The cube being expanded
p //The cube that we wish to cover
$\mathcal{R}$ //Cover of the OFF-set
Output:
is_feasible //Can c be expanded to cover p without intersecting $\mathcal{R}$?
1: c^+ = expanded version of c to minimally cover p
2: **for each** cube r in $\mathcal{R}$ **do** //LOOP-4.1
3: **if** c^+ intersects with r **then**
4: return false
5: **end if**
6: **end for**
7: return true

Algorithm 3 calc_disjoint_cnt

Input:
lp //Lowering set of a cube
$\mathcal{F}$ //Cover of the ON-set
Output:
count //Number of FEASIBLE cubes in $\mathcal{F}$ that are disjoint from lp
1: count = 0
2: **for each** cube q in $\mathcal{F}$ **do** //LOOP-4.2
3: **if** q is FEASIBLE and q is disjoint from lp **then**
4: count = count + 1
5: **end if**
6: **end for**
7: return count

covers which have specific values of flags set for them. For instance, we only deal with ACTIVE cubes in the algorithm. In LOOP-2, c is expanded by dropping a set of literals in every iteration. In `essen_raise` (line 6 of Algorithm 1), if some part of the Boolean space is not blocked by any cube of $\mathcal{R}$, then this part is raised in c to get c^+. Following this, a one-step look-ahead heuristic is used in order to select the cube p^* from $\mathcal{F}$ and c^+ is expanded to minimally cover p^*. Using LOOP-3.1 and LOOP-4.1 c^+ is tentatively expanded to cover each uncovered cube. Then, using LOOP-3.2 and LOOP-4.2 p^* is selected as i) the cube that can be covered by expanding c^+ without intersecting $\mathcal{R}$ and ii) covering the largest number of other uncovered cubes after this expansion. After every iteration, at least one additional cube of $\mathcal{F}$ is covered and at least one literal of c is dropped. Hence,this loop runs in $O(min(n, |\mathcal{F}|))$. However, large sparse incompletely specified Boolean functions have $n << |\mathcal{F}|$ and hence this loop can be said to run $O(n)$ times.

LOOP-3.1 iterates over each uncovered cube p of $\mathcal{F}$ and uses LOOP-4.1 to see if c can be expanded to cover p without intersecting $\mathcal{R}$. If so, it calculates lowering_set[p], which is the set of literals that cannot be dropped if c^+ is expanded to minimally cover p. Otherwise, it marks the cube as INACTIVE and INFEASIBLE and such cubes are removed from consideration. This loop runs $O(|\mathcal{F}|)$ times. LOOP-4.1 iterates over cubes of $\mathcal{R}$ to check if the tentatively expanded version of c^+ intersects with $\mathcal{R}$. The check runs in $O(n)$ since it calculates the distance between the 2 cubes. Hence, LOOP-4.1 has a total complexity of $O(n|\mathcal{R}|)$. At this point in the algorithm we have identified every uncovered cube of $\mathcal{F}$ that can be feasibly covered along with their corresponding lowering_set. LOOP-3.2 iterates over each feasibly covered cube p to get count[p], which is the number of feasibly covered cubes in $\mathcal{F}$ which are disjoint from the lowering_set[p]. This loop runs $O(|\mathcal{F}|)$ times.

LOOP-4.2 iterates over each feasibly covered cube q of $\mathcal{F}$ and checks whether it is disjoint from lowering_set[p]. Disjoint check runs in $O(n)$. Hence, LOOP-4.2 has a total complexity of $O(n|\mathcal{F}|)$. The overall time complexity is: $\mathcal{O}(n^2|\mathcal{F}|)^2|\mathcal{R}|)$.

Algorithm 4 distill_cubes

Input:
$\mathcal{C}$ //Cover of a function
$flag$ //Flag that should be present on a cube to be filtered in
Output:
$\mathcal{C}_f$ //Cover of the function with only containing cubes that have the specified $flag$
1: $\mathcal{C}_f$ = Empty cover
2: **for each** cube $c \in \mathcal{C}$ **do**
3: **if** c has $flag$ **then**
4: Add c to $\mathcal{C}_f$
5: **end if**
6: **end for**
7: return $\mathcal{C}_f$

IRREDUNDANT_COVER step: ESPRESSO-II splits $\mathcal{F}$ into three sets, relatively essential $\mathcal{F}_1$, totally redundant, and partially redundant $\mathcal{F}_2$. $\mathcal{F}$ consists of cubes covering some minterms of the function not covered by any other cubes. Clearly, they must be included in any cover of *ff*. The totally redundant set includes cubes that are covered by $\mathcal{F}_1 \cup \mathcal{D}$. Since $\mathcal{F}$ is included, there is no need to include these totally redundant cubes. The remaining cubes are included in $\mathcal{F}_2$ (each of these cubes may be individually removed from $\mathcal{F}$ without destroying the covering property). $\mathcal{F}_2$ is determined by the PARTIALLY_REDUNDANT procedure of ESPRESSO-II, which takes $\mathcal{O}(n|F|^2)$. Next function MINIMAL_IRREDUNDANT is called to find maximum subset of partially redundant cover to remove from the $\mathcal{F}$.

Internal data representation: The software implementation of ESPRESSO-II represents each input/output variable using two bits: 01 if the variable has a value of zero, 10 if the variable has a value of one, and 11 if the variable has a value of 2 (don't-care). It then packs every 16 variables into an unsigned integer to minimize memory complexity. Additionally, it stores the mete-data corresponding to each cube in a separate unsigned integer. The meta-data includes information about whether the cube is prime, covered, etc. The software implementation of ESPRESSO-II stores all cubes that constitute $\mathcal{F}$, $\mathcal{R}$, or $\mathcal{D}$ in an array of unsigned integers where each contiguous sub-array of size k represents an individual cube (the value of k is determined by the number of input variables and output variables). Therefore, the $\mathcal{F}$ will be represented with an array of size $k|\mathcal{F}|$ ($\mathcal{R}$ and the $\mathcal{D}$ are represented with similar arrays). Fig. 1 illustrates an example of such array.

B. Review of the CUDA Platform

A GPU is implemented as a set of multiprocessors as illustrated in [7]. Each multiprocessor has a single instruction, multiple thread (SIMT) architecture where at any given clock cycle, different processors of a multiprocessor execute the same instruction on different data. A portion of an application which is executed many times, but independently on different data, can be cast as a function, which is executed on different threads on a GPU. To that effect, such a function is mapped to a set of instructions chosen from the instruction set of the GPU and the

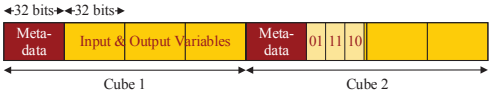

Fig. 1. An array of unsigned integers representing two cubes. Each cube consists of four unsigned integers where the first one stores the meta-data and the rest store values of input and output variables.

resulting program, called a kernel, is downloaded to the GPU. The batch of threads that realize a kernel is organized as a grid of thread blocks (a.k.a. blocks). Each thread block is processed by only one multiprocessor so that the shared memory space resides in the on-chip shared memory of that multiprocessor. This, in turn, leads to very fast memory accesses.

CUDA is a hardware-software architecture which exposes the parallel data processing capabilities of GPUs. CUDA allows users to view the GPU as a highly multi-threaded co-processor that offloads the CPU when executing compute-intensive applications. CUDA provides general DRAM memory addressing on GPUs for more programming flexibility and supports both scatter and gather memory operations. CUDA also provides access to a parallel data cache or an on-chip shared memory with very fast read and write accesses. Additionally, CUDA provides atomic operations like `atomicAdd()` and `atomicSwap()` to enable developers avoid race conditions while accessing/updating shared data structures.

III. Proposed Parallel Implementation

As detailed in Section II-A, the bulk of computations in the EXPAND step happens in four nested loops. This section explains which loops are parallelized on GPUs and why as well as details of how parallelizations are achieved.

The first loop iterates over uncovered cubes of $\mathcal{F}$ and expands each uncovered cube c into a prime implicant. Parallelizing this loop by expanding multiple uncovered cubes at the same time has three main disadvantages. First and foremost, it is likely that some of the cubes under expansion can cover each other. In other words, if those cubes were expanded serially, some of them would have been covered by previous expansions and never considered independently for expansion. As a result, the chances of performing wasteful computations is increased. Second, because different cubes are expanded independently of each other, they may cover the same cubes of $\mathcal{F}$ multiple times. Therefore, the number of prime implicants found at the end of the EXPAND step is likely to increase. While this phenomenon may also occur in the serial version of the EXPAND step, the heuristic behind the EXPAND step is designed to favor covering uncovered cubes. Consequently, the serial version of the EXPAND step is expected to have fewer prime implicants. Third, because the expansion of each cube requires modifying the meta-data corresponding to both $\mathcal{F}$ and $\mathcal{R}$, these covers must be replicated for each cube under expansion and later merged properly to a consistent state. Such replication of $\mathcal{F}$ and $\mathcal{R}$ increases memory requirements and reduces efficient use of the DRAM bandwidth.

The second loop, which performs an iterative expansion, cannot be parallelized due to loop-carried dependencies. As a result, this work implements the second loop serially. The remaining loops of the EXPAND step, which implement a single iteration of the iterative EXPAND procedure, are ones that can be parallelized effectively. In terms of memory accesses, it is required to transfer $\mathcal{F}$ and $\mathcal{R}$ from the CPU to the GPU before the iterative EXPAND procedure starts (the GPU implementation uses the exact same data representation as ESPRESSO-II). Additionally, the updated $\mathcal{F}$ and $\mathcal{R}$ as well as the expanded cube need to be transferred from the GPU to the CPU when the expansion ends. Because the number of computations performed on $\mathcal{F}$ and $\mathcal{R}$ in the innermost loops is very large, the amortized cost of these data transfers is negligible.

It is important to note that parallelization is performed for different reasons for different blocks of code in the two innermost loops. Some blocks of code or functions like distance calculation are inherently parallelizable while some other are serial in nature and/or include conditional statements. The blocks of code which are inherently parallelizable are processed by GPUs to boost performance. However, the blocks of code which are serial in nature should preferably be mapped to CPUs due to their advanced capabilities such as branch prediction, out-of-order execution, and multi-level caching. Mapping those blocks to GPUs will lead to performance degradation unless a very large number of threads is launched. In that case, the GPU's performance will be on a par with that of a CPU. To achieve performance gains in processing the EXPAND step and to avoid multiple data transfers between the CPU and the GPU, different functions of the EXPAND step need to be restructured carefully to enable efficient parallelization for both inherently serial and parallel blocks of code. Parallelizing computations of the said loops requires defining multiple CUDA kernels, some of which can be reused for different parts of the computations.

Filtering cubes: As illustrated in Section II-A, the iterative EXPAND procedure only deals with active cubes of $\mathcal{F}$ to reduce the number of computations. Extracting active cubes requires two steps. First, one needs to look at the meta-data corresponding to each cube and check the status of the active flag. Next, one should store the indices of active cubes in an array for future use.

The CUDA kernel which extracts active cubes proceeds as follows (similar CUDA kernels can be designed to find feasible or covered cubes). It first creates one or more thread blocks where each thread block deals with a subset of all cubes by examining a contiguous sub-array in the array that represents $\mathcal{F}$. Next, the threads inside each thread block examine the active flag of their corresponding cube. At this point, all active cubes are identified. However, locations in the output array in which each thread has to write its active cubes are yet to be determined. There are two pieces of information that are required for finding the locations (a.k.a. indices) of each active cube in the output array. The first one is the number of actives cubes in each thread block while the second one is the index of an active cube within each thread block. The total number of active cubes in thread blocks that precede a certain thread block determines an index from which the thread block has to write its active cubes. Similarly, the index of each active cube inside that thread block determines the offset that should be added to the start index to find the actual index corresponding to the active cube. These indices have to be calculated dynamically because the number of active cubes in each thread block may change from one iteration to another.

To determine the number of active cubes in each thread block, each thread atomically increments a counter local to the thread block when its corresponding cube is active. Concurrently, it reads the previous value which was stored in the local counter and uses that value as the offset (all counters are initialized to zero). Next, the thread with an index of zero in each thread block (the leader thread) atomically adds value of the local counter corresponding to its thread block to a global counter while reading the value which was previously stored in

the global counter. The read value determines the start index of that thread block. Each thread corresponding to an active cube adds the value read by the leader thread to its offset to find its resulting index. Finally, all threads write their active cubes in the output array in parallel. Fig. 2 illustrates an example execution of a CUDA kernel which implements parallel filtering.

Finding feasibly covered cubes (LOOP 3.1 & 4.1): By definition, feasibly covered cubes are cubes of $\mathcal{F}$ which can be covered by raising some literals of the cube under expansion and without intersecting $\mathcal{R}$. ESPRESSO-II iterates over all uncovered, non-prime cubes of $\mathcal{F}$ and temporarily expands the cube under expansion to cover that cube. It then calculates the distance between the temporarily expanded cube and all cubes of $\mathcal{R}$ to find possible intersections (in this context, distance reflects the number of variables where one cube has a value of zero while the other has a value of one). If no such intersections exist, the uncovered, non-prime cube is marked as feasible. For each feasible cube, a lowering set is defined as a set of variables of the cube under expansion which cannot be raised if it is expanded to cover the uncovered, non-prime cube.

Since the temporary expansion of the cube under expansion for each uncovered non-prime cube is independent of that of other cubes, this step can be parallelized on GPUs. However, naïve parallelization would lead to reading the whole $\mathcal{R}$ from the DRAM for each uncovered, non-prime cube which, in turn, defeats the purpose of parallelization. ESPRESSO-GPU takes advantage of tiling to maximize data reuse in shared, on-chip memory and reduce the number of DRAM accesses.

Assume $\mathcal{F}$, $\mathcal{R}$, and a distance matrix that keeps track of pairwise distances between temporarily expanded cubes and cubes of $\mathcal{R}$ as illustrated in Fig. 3. The proposed implementation partitions the distance matrix and assigns the distance calculations of each part to a grid of thread blocks. Next, it assigns each thread block in the grid to a subset of the part that corresponds to the grid as shown in Fig. 3 (each of these subsets is referred to as a tile). After that, the leftmost threads of a thread block read the cubes of the on-set that are required for distance calculation while the topmost threads read the required cubes of the off-set and store them in the shared memory. Finally, different threads of a thread block quickly calculate distances using shared data. Tiling results in significant savings in memory bandwidth because of its inherent sharing. For example, designing 32×32 tiles leads to $32\times$ reduction in DRAM accesses for reading the cubes of the on-set and off-set i.e., each fetched cube of the on-set/off-set will be used in 32 distance calculations with 32 cubes of the off-set/on-set.

ESPRESSO-GPU assigns multiple distance calculations to each thread to amortize the cost of launching threads across multiple computations. We refer to the number of distance calculations a thread performs as *thread reuse factor* (TRF).

Counting feasibly covered cubes (LOOP-3.2 & 4.2): As

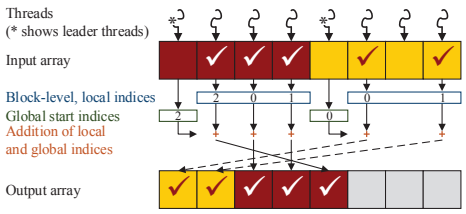

Fig. 2. Parallel distillation of cubes. Check marks indicate active cubes.

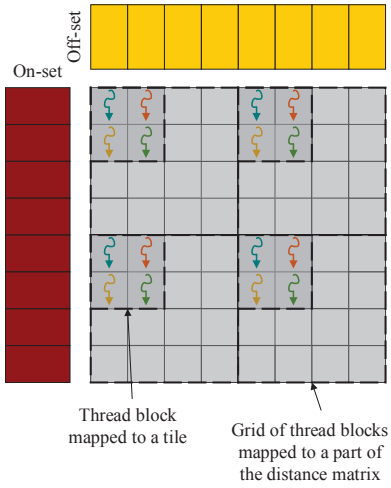

Fig. 3. Parallel distance calculation using tiling. In this example, $|\mathcal{F}| = |\mathcal{R}| = 8$ and therefore, the distance matrix is 8×8. This matrix is partitioned into four parts (a.k.a. grids), each of which is 4×4. Each grid consists of four thread blocks where each thread block processes a 2×2 sub-matrix. Each thread processes one distance value in each grid as shown by the color-coded threads. This amortizes the cost of launching a thread across four computations.

described in Algorithm 1, for each feasible cube, it counts the number of feasible cubes that are disjoint from the lowering set of that feasible cube. Next, by picking the feasible cube with the highest count, it enables a large number of uncovered, non-prime cubes to remain feasible and contribute to further expansions. Similar to the CUDA kernel that finds feasible cubes (see Fig. 3), the CUDA kernel for finding the number of disjoint feasible cubes formulates this problem as a two-dimensional count matrix where both dimensions are associated with feasible cubes. It then launches a two-dimensional grid of thread blocks, which is responsible for counting plus data transfer while adopting tiling to minimize data transfer between the GPU and the off-chip memory.

Doing Argmax over disjoint counts: The CUDA kernel that finds the feasible cube with the highest number of disjoint cubes partitions the count matrix and assigns the computations for finding the feasible cube with the highest number of disjoint cubes in each part to a thread block. The threads inside each thread block perform a tree-based reduction to find such a feasible cube as shown in Fig. 4.

Assuming a thread block with t threads has to find a feasible cube in an array of size $2t$, each thread i loads the data at indices i and $i+t$ and returns the cube with a higher number of disjoint cubes. This halves the size of the array in each iteration until the feasible cube with the maximum number of disjoint cubes is found inside each thread block. Finally, different thread blocks write their output feasible cubes into a secondary array which is passed to a tree-based reduction thread block to find the target feasible cube across all thread blocks.

It is important to note that this work parallelizes the EXPAND step such that its output is exactly the same as the output of ESPRESSO-II's EXPAND step which facilitates debugging and end-to-end testing.

IRREDUNDANT_COVER step: We leverage CUDA's dynamic parallelism facility to create and synchronize new nested work for steps of IRREDUNDANT_COVER where we can have a hierarchy of parallel algorithms. At the splitting stage, we check

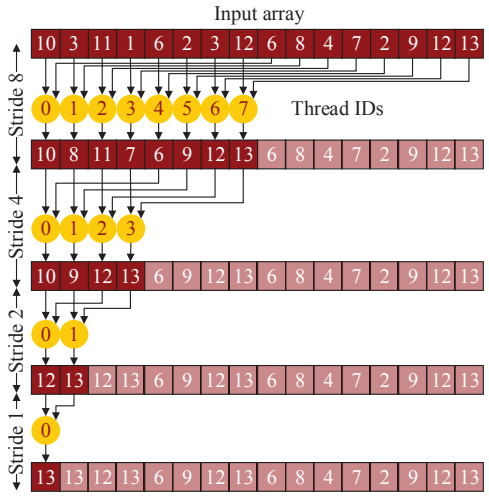

Fig. 4. Parallel tree-based reduction for finding the maximum value (or the index thereof) in an array.

for every cube of $\mathcal{F}$ whether it is covered by other cubes of $\mathcal{F}$ in parallel since these checks are independent. A cube c is covered by the given cover if the cover obtained after co-factoring with respect to c is tautology. The co-factorization is parallelizable very similar to the distillation of cubes in EXPAND step. The tautology check uses the unate recursive paradigm that can proceed in parallel using the dynamic parallelism capability of CUDA. A Boolean expression is a a tautology if its matrix representation has a row of all 2's; it is not a tautology if the matrix has a column of all 1's or all 0's or it has deficient vertex count. Checks for these special (terminal) cases are parallelizable. Furthermore, the binate variable selection as well as co-factor operations are parallelizable.

IV. Results & Discussion

To evaluate the efficacy of the proposed parallel two-level logic minimizer, we find Boolean functions using the approach presented in [2] for neurons in a multi-layer perceptron trained on the MNIST dataset [10] in addition to neurons in a convolutional neural network trained on the CIFAR-10 dataset [5]. Next, we create three classes of Boolean functions for our experiments where each class includes 10 functions: a *small* class where each function has 10,000 minterms, a *medium* class where each function has 60,000 minterms, and a *large* class where each function has 100,000 minterms.

The first set of experiments compares the average time per call to the distance calculation kernel for different values

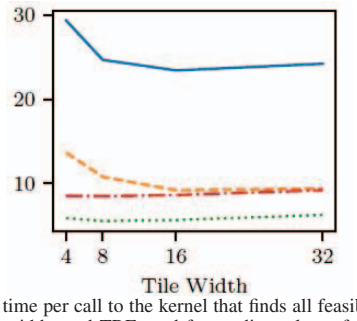

Fig. 5. Average time per call to the kernel that finds all feasibly covered cubes for different tile widths and TRFs and for medium class of functions.

TABLE I
Comparison of the time taken by ESPRESSO-II vs ESPRESSO-GPU and the observed speedup

PLA class	Mean ± Std ESPRESSO-II runtime (s)	Mean ± Std ESPRESSO-GPU runtime(s)	Speedup
Small	92.7 ± 37.7	3.4 ± 1.07	25.7
Medium	8177.8 ± 1702.5	72.7 ± 27.6	126.4
Large	107056 ± 21512.03	757.6 ± 129.86	140.4

of tile width and thread reuse factor and for the medium class of Boolean functions. As illustrated in Fig. 5, increasing TRF reduces computation time up to a point but increases it afterwards. In fact, the proper value for the TRF balances between amortization of the cost of launching threads and the degree of parallelization. If a thread is reused too many times, parallelization will be hampered. It is observed that when TRF is 16 and tile width is 8, the lowest execution time is achieved. Therefore, we set these values for later experiments (we have run similar experiments for other kernels to find their best set of hyperparameters that minimize the execution time).

The next set of experiments compare the execution time of the entire EXPAND step between ESPRESSO-GPU and ESPRESSO-II. These experiments were run on 10 PLAs for each class. Table I shows the Mean/Std of runtimes for ESPRESSO-II vs ESPRESOO-GPU and gained Speedup. We note that as the size of PLA increases the speedup also increases. This is because the number of computations in large PLAs help in generating extremely high throughput that benefits from the massively parallel GPU architectures. With the gained speedup, PLAs that take more than a day to run a single EXPAND step on ESPRESSO-II, can be given to ESPRESSO-GPU to generate the same output in 10-15 minutes.

V. Conclusions

We presented ESPRESSO-GPU, a parallel two-level logic minimization heuristic based on ESPRESSO-II for dealing with Boolean functions with tens of thousands of minterms. Our parallel implementation provides, on average, a speedup of 97x compared to a serial implementation.

References

[1] R. K. Brayton, M. Gao, J. R. Jiang, Y. Jiang, Y. Li, A. Mishchenko, S. Sinha, and T. Villa, "Optimization of multi-valued multi-level networks," in *Int'l Symp on Multiple-Valued Logic*, 2002, pp. 168–179.
[2] M. Nazemi, G. Pasandi, and M. Pedram, "Energy-efficient, low-latency realization of neural networks through Boolean logic minimization," in *Asia and South Pacific Design Automation Conf*, 2019, pp. 274–279.
[3] Y. Umuroglu, Y. Akhauri, N. J. Fraser, and M. Blott, "LogicNets: Co-designed neural networks and circuits for extreme-throughput applications," in *Int'l Conf on Field-Programmable Logic and Applications*, 2020, pp. 291–297.
[4] K. Simonyan and A. Zisserman, "Very deep convolutional networks for large-scale image recognition," in *Int'l Conf on Learning Representations*, 2015.
[5] A. Krizhevsky, G. Hinton *et al.*, "Learning multiple layers of features from tiny images," 2009.
[6] R. K. Brayton, G. D. Hachtel, C. T. McMullen, and A. L. Sangiovanni-Vincentelli, *Logic Minimization Algorithms for VLSI Synthesis*, ser. The Kluwer Int'l Series in Engineering and Computer Science, 1984, vol. 2.
[7] J. Nickolls, I. Buck, M. Garland, and K. Skadron, "Scalable parallel programming with CUDA," *ACM Queue*, vol. 6, no. 2, pp. 40–53, 2008.
[8] R. L. Rudell, "Logic synthesis for VLSI design," Ph.D. dissertation, EECS Department, University of California, Berkeley, April 1989.
[9] S. Sapra, M. Theobald, and E. M. Clarke, "SAT-based algorithms for logic minimization," in *Int'l Conf on Computer Design*, 2003, p. 510.
[10] Y. LeCun and C. Cortes, "MNIST handwritten digit database," 2010. [Online]. Available: http://yann.lecun.com/exdb/mnist/