Int J Parallel Prog (2011) 39:639–661
DOI 10.1007/s10766-011-0164-7

Speeding Up Cycle Based Logic Simulation Using Graphics Processing Units

Alper Sen · Baris Aksanli · Murat Bozkurt

Received: 15 October 2010 / Accepted: 30 December 2010 / Published online: 22 February 2011

Abstract Verification has grown to dominate the cost of electronic system design, consuming about 60% of design effort. Among several verification techniques, logic simulation remains the major verification technique. Speeding up logic simulation results in great savings and shorter time-to-market. We parallelize logic simulation using Graphics Processing Units (GPUs). In the past, GPUs were special-purpose application accelerators, suitable only for conventional graphics applications. The new generations of GPU architecture provide easier programmability and increased generality while maintaining the tremendous memory bandwidth and computational power of traditional GPUs. We develop a parallel cycle-based logic simulation algorithm that uses And Inverter Graphs (AIGs) as design representations. AIGs have proven to be an effective representation for various design automation applications, and we obtain similar benefits for speeding up logic simulation. We develop two clustering algorithms that partition the gates in the designs into independent blocks. Our algorithms exploit the massively parallel GPU architecture featuring thousands of concurrent threads, fast memory, and memory coalescing for optimizations. We demonstrate up-to 5x and 21x speedups on several benchmarks using our simulation system with the first and second clustering algorithms, respectively. Our work ultimately results in significant reduction in the overall design cycle.

Keywords Graphics processing units (GPU) · Parallel logic simulation · Cycle based simulation · And inverter graph (AIG)

This article is an extended version of a conference paper that appeared at ISPDC 2010 [27].

A. Sen (✉) · B. Aksanli · M. Bozkurt
Department of Computer Engineering, Bogazici University, 34342 Istanbul, Turkey
e-mail: alper.sen@boun.edu.tr

1 Introduction

Complexity of electronic designs have been rapidly growing. Billions of transistors are commonly placed in designs to generate higher performance. Multicore and many core systems provide the performance by handling more work in parallel. However, the complexity in electronic designs poses a great challenge for building such systems, where designs continue to be released with latent bugs. Functional design verification is the task of establishing that a given design accurately implements the intended functional behavior (specification). Today, design verification has grown to dominate the cost of electronic system design, in fact, consuming about 60% of design effort [5]. Among several verification techniques, logic simulation remains to be the major verification technique due to its applicability to real designs and its relative ease of use. However, logic simulation of designs with millions of components is time consuming and has become a bottleneck in the design process. Any means to speedup logic simulation results in productivity gains and faster time-to-market.

We observe that electronic designs exhibit a lot of parallelism that can be exploited by parallel algorithms. In fact, there are parallel electronic design automation algorithms for almost all stages in the implementation of an electronic design such as logic optimization, floor-planning, routing, and physical verification stages. In this paper, we focus on parallelization of logic simulation of electronic designs using Graphics Processing Units (GPUs).

In the past, Graphics Processing Units (GPUs) were special-purpose application accelerators, suitable only for conventional graphics applications. Nowadays, GPUs are routinely used in applications other than graphics such as computational biology, computational finance and electronic designs, where huge speedups have been achieved [22]. This is a result of the availability of extremely high performance hardware (for example, a GTX 480 GPU provides 1.35 Tflops with 480 cores and is under $500), and the availability of general purpose programming models such as Compute Unified Device Architecture (CUDA) by NVIDIA [13]. CUDA is an extension to C language and is based on a few abstractions for parallel programming. CUDA has accelerated the development of parallel applications beyond that of the original purpose of graphics processing.

There are two main types of logic simulation; Cycle-Based Simulation (CBS) and Event-Based Simulation (EBS). In CBS, the evaluation schedule of gates in the design for each step of simulation is determined once at the compilation time of the simulator. EBS has a more complicated scheduling policy where a gate is simulated only if at least one of its input values have changed. Both CBS and EBS are commonly used in the industry. In this paper, we work with CBS since it has a less complicated static scheduling policy that is amenable to better parallelization. However, we also note that logic simulation has been classified as the most challenging Computer Aided Design pattern to parallelize [7]. The ratio of communication (between gate and gate) to Boolean evaluation is high. Since the communication pattern of logic simulation often defy a pattern, it is difficult to assign elements to processors so that communicating elements are always close. High performance loss due to communication overhead is reported [26].

Logic simulation proceeds in two phases; compilation and simulation. The compilation phase is part of any modern simulator and is required to convert the design into

an appropriate form for the simulation phase. Our GPU based solution is also composed of these two phases. In the compilation phase, we perform several operations namely combinational logic extraction, levelization, clustering, and balancing operations. This phase has several GPU architecture dependent optimizations such as the efficient usage of shared memory and memory coalescing. Levelization helps determine the dependency between gates, where gates in the same level can be simulated in parallel. Clustering helps partition the design into a collection of smaller parts where each part can be simulated independent of other parts. Balancing helps optimize the usage of threads for a SIMD instruction. These operations are necessary because in a gate level design, certain gates could have hundreds or thousands of fanouts while most will have a few fanouts resulting in irregular data access patterns. Clustering and balancing help organize access patterns for effective simulation. The simulation phase is where thousands of threads are available to simulate the compiled design in parallel. We use several optimizations in simulation phase in order to reduce the communication overhead between the GPU and the CPU. In both phases, we exploit the GPU memory resources as efficiently as possible in order to have low latency.

Our design representation for logic simulation is different from the other design representations. We use And-Inverter Graphs (AIGs), in particular we use the AIGER format [2]. AIG is an efficient representation for manipulation of boolean functions. It is increasingly used in logic synthesis, technology mapping, verification, and boolean satisfiability checking [6,10,20,21,28]. However, to the best of our knowledge, our work is the first GPU based electronic design automation solution using AIGs. Since we use AIGs with only a single type of combinational gate (and-gate), our algorithms can efficiently use the limited low latency memory spaces provided by the GPU.

We developed two clustering algorithms. The first one clusters gates using a given threshold, and the second one improves clustering with that of merging and balancing steps and incorporates memory coalescing and efficient utilization of shared memory. We validated the effectiveness of our parallel CBS algorithm for both types of clustering with several benchmarks from IWLS, and AIGER [2,17,24]. We compared our parallel CBS algorithm with that of a sequential CBS algorithm. Our experiments show that parallel CBS can speedup the simulation of designs over the sequential algorithm. In particular, we obtained up-to 5x speedup with the first clustering algorithm and up-to 21x with the second clustering algorithm. We obtain better speedups with larger circuits.

This paper is organized as follows. In the next section, we give an overview of related work in logic simulation and General Purpose computation on GPUs (GPGPU). We then describe background in CUDA, AIG format and logic simulation in Sect. 3. In Sect. 4, we describe our parallel CBS algorithm together with two clustering algorithms and CUDA optimizations. Our experiments are in Sect. 6. Finally, conclusions and future work are described.

2 Related Work

Catanzaro et al. [7] explain design patterns for parallelization in Computer Aided Design (CAD). The authors consider 17 different CAD algorithms and partition

these algorithms in three categories; graph algorithms, branch and bound search, and linear algebra. They also state that graph algorithms in CAD (of which logic simulation is a member) are the hardest to parallelize among these categories. Similarly, CAD case studies of GPU acceleration can be found in [12]. Some of these case studies are spice simulation, fault simulation, static timing analysis, boolean satisfiability, fault dictionary computation, and power grid analysis. The authors describe optimization techniques for irregular EDA applications on GPUs in [14]. In particular, they make use of memory coalescing and shared memory utilizations that improve the speedup of sparse matrix vector product and breadth first search.

There has been a lot of work on parallel logic simulation using architectures other than GPUs. There are several surveys on parallel logic simulation [4,19]. In particular, cycle-based simulation approaches are used by IBM and others [15]. The simulation algorithms in these works are aimed at loosely coupled processor systems.

Different partitioning algorithms for electronic designs are described in [3,18]. Some of these algorithms are based on performance, layout, clustering, network flow, and spectral methods. Our partitioning approach is similar to cone clustering described in [16], where a fanin cone of a circuit element embodies an area of combinational logic that has the potential to influence signal values provided by that element.

Several general purpose GPU applications can be found in [13,22]. The application domain ranges from physics to finance and the medical field. The work in [11] gives a performance study of general purpose applications using CUDA and compares them with that of applications written using OpenMP.

There is another general purpose programming environment for GPUs named OpenCL. OpenCL [23] is a relatively new standard that is very similar to CUDA in that it is also an extension of C language. CUDA is specific to NVIDIA GPUs, whereas OpenCL can be run on different architectures and gives you portability at the expense of potentially sub-optimal performance for any specific platform. Also, CUDA is a more mature environment with high-performance libraries and accompanying tools like debuggers and profilers.

Our CBS algorithm is most similar to the work by Chatterjee et al. [9]. However, there are several differences. We use AIGs as gate level representation whereas they support a generic library of gates. AIGs allow us to efficiently use the limited low latency memory spaces. We use a threshold value in our first clustering algorithm, whereas they start clusters from the primary outputs. We fix the number of blocks in our second clustering algorithm in order to maximize parallel thread execution. We encode variables in order to better utilize shared memory and we explicitly use memory coalescing, whereas these are not used in [9].

There is an earlier logic simulation algorithm using GPUs by Perinkulam [25]. However, this algorithm does not provide performance benefits since they do not optimize data transfer between GPU and CPU, use a different partitioning approach, and do not use the general purpose programming language CUDA. There is also a recent work on event-based simulation algorithm, which is also a commonly used simulation technique in the industry, using CUDA [8].

3 Background

In this section, we are going to present background on CUDA programming, AIGs, and sequential cycle based logic simulation of electronic designs.

3.1 CUDA Programming

Compute Unified Device Architecture (CUDA) is a small C library extension developed by NVIDIA to expose the computational horsepower of NVIDIA GPUs [13]. GPU is a compute *device* that serves as a co-processor for the *host* CPU. CUDA can be considered as an instance of widely used Single Program Multiple Data (SPMD) parallel programming models. A CUDA program supplies a single source code encompassing both host and device code. Execution of this code consists of one or more phases that are executed either on the host or device. The phases that exhibit little amount of parallelism are executed on the host and rich amount of parallelism are executed on the device. The device code is referred to as *kernel* code.

The smallest execution units in CUDA are *threads*. Thousands of threads can work concurrently at a time. GPU has its own device memory and provides different types of memory spaces available to threads during their execution. Each thread has a private local memory in addition to the registers allocated to it. A group of threads forms a CUDA *block*. A CUDA block can have at most 512 threads, where each thread has a unique id. A group of blocks forms a CUDA *grid*. Each thread block has a *shared memory* visible to all threads of the block within the lifetime of the block. Access to shared memory is fast like that of a register. Threads in the same block can synchronize using a barrier, whereas threads from different blocks can not synchronize. Each thread has access to the *global device memory* throughout the application. Access to the global memory is slow and around 400–600 cycles. There are also special memory spaces such as texture, constant, and page-locked (pinned) memories. All types of memory spaces are limited in size, and should therefore be handled carefully in the program.

Figure 1 displays general NVIDIA CUDA architecture with N streaming multiprocessors composed of M streaming processors. For an NVIDIA FX3800 GPU, $N = 24$ and $M = 8$. Each multiprocessor can execute 768 threads in parallel and shared memory size is limited where each multiprocessor can have 16 KB of shared memory.

3.2 And-Inverter Graph (AIG)

And-Inverter Graph (AIG) is a directed, acyclic graph that represents a structural implementation of the logical functionality of a design or circuit. AIGER format is an implementation of AIGs [2]. An AIG is composed of two-input and-nodes (combinational elements) representing logical conjunction, single input nodes representing memory elements (latches, sequential elements), nodes labeled with variable names

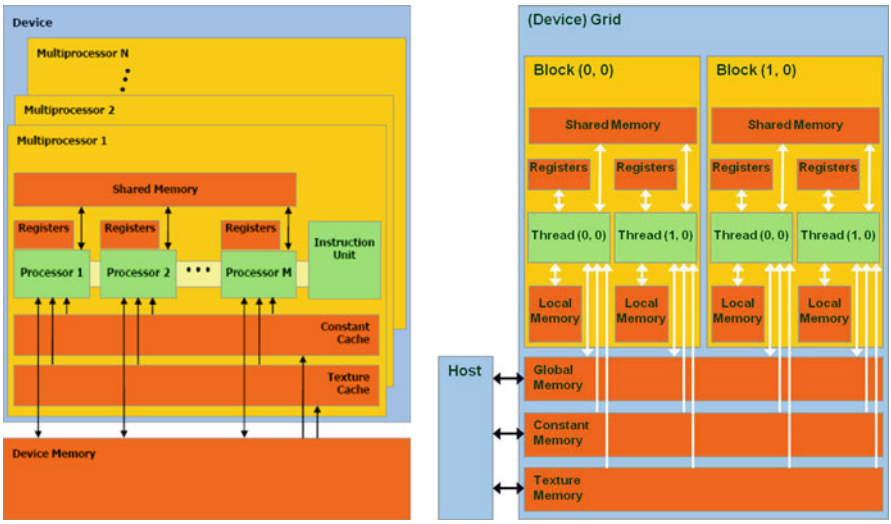


Fig. 1 CUDA hardware and memory architecture (*Source*: NVIDIA CUDA programming Guide [13])

representing inputs and outputs, and edges optionally containing markers indicating logical negation. We refer to and-nodes and latches as gates. AIG is an efficient representation for manipulation of boolean functions.

The combinational logic of an arbitrary Boolean network can be factored and transformed into an AIG using DeMorgans rule. The following properties of AIGs facilitate development of robust applications in synthesis, mapping, and formal verification. Structural hashing ensures that AIGs do not contain structurally identical nodes. Inverters are represented as edge attributes. As a result, single-input nodes representing inverters and buffers do not have to be created. This saves memory and allows for applying DeMorgans rule on-the-fly, which increases logic sharing. The AIG representation is uniform and fine-grain, resulting in a small, fixed amount of memory per node. The nodes are stored in one memory array in a topological order, resulting in fast, CPU-cache-friendly traversals.

There has been a growing interest in AIGs as a functional representation for a variety of tasks in Electronic Design Automation (EDA) such as logic synthesis, technology mapping, verification, and equivalence checking [6,10,20,21,28]. Especially, the recent emergence of efficient boolean satisfiability (SAT) solvers that use AIGs instead of the Binary Decision Diagrams (BDD) has made AIGs popular in EDA. A tool called ABC [6] features an AIG package, several AIG-based synthesis and equivalence-checking techniques, as well as an implementation of sequential synthesis.

Figure 2 displays an example gate level design. In this figure, and-gates are identified as *G#*, where *#* stands for the unique gate number; and latches are identified as *L#*, where *#* stands for the unique latch number. *i# and o#* stand for (primary) inputs and (primary) outputs, respectively.

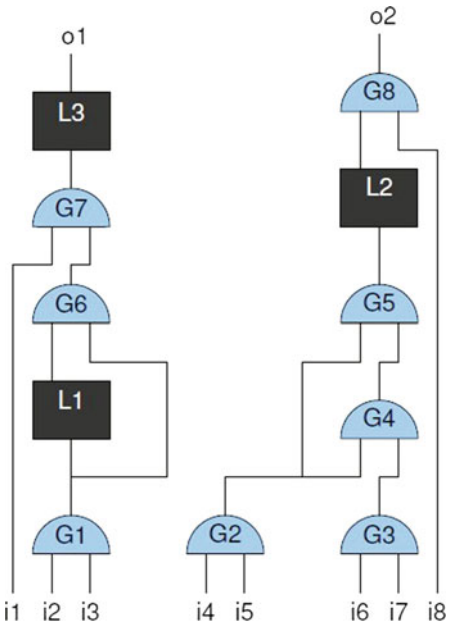


Fig. 2 An example gate level design

3.3 Sequential Cycle-Based Simulation

Cycle-based simulation (CBS) is a commonly used logic design simulation technique. Given a sequence of input vectors, the goal is to generate the output vectors of the design. CBS is a form of time driven or compiled mode simulation because the evaluation schedule of gates for each simulation step is determined once at the compilation time of the simulator. The time in CBS is not a physical time rather an integer that denotes the current cycle. CBS represents a fast alternative to event-based simulation. In CBS, logic values of *all* gates are calculated at clock defined cycle boundaries, whereas event-based simulators calculate logic values of a gate if a change occurs at the inputs of that gate. This property of CBS may bring some redundancy since inputs of a combinational element may not change at every cycle. However, it eliminates costly management of scheduling of events as in event-based simulation resulting in efficient implementation rules and better performance.

Algorithm 1 displays a pseudocode for a general sequential CBS algorithm. In our case, since we use AIGs, the only combinational elements are and-gates. At each step, both combinational and sequential elements are evaluated. The primary output values are calculated at every cycle using the current values of sequential elements and primary inputs. The combinational elements are evaluated generating the next latch values. The combinational elements can be simulated in any order as long as the inputs of an and-gate are ready for the current cycle. An optimization at this step would be to levelize the combinational elements and simulate them level by level, which we will explain in detail in the next sections. Next, sequential elements are simulated. The next latch values are stored in a second set of latch variables that will take the role of the

Algorithm 1 Sequential Cycle-Based Simulation Algorithm

Input: circuit, number of simulation cycles *num*, test-bench
Output: output values
1: **for** i = 0 to *num* **do**
2: execute test-bench and generate primary input values;
3: simulate all combinational elements;
4: simulate all sequential elements;
5: generate primary output values using primary input values and the current values of sequential elements;
6: **end for**

Table 1 Cycle-based simulation of design in Fig. 2

Cycle	Inputs	Latches	Outputs
	$i_1 i_2 i_3 i_4 i_5 i_6 i_7 i_8$	$L_1 L_2 L_3$	$o_1 o_2$
0	11111111	000	00
1	11111111	110	01
2	11111111	111	11

present (current) latch variables in the next cycle. For example, assuming that the latch values are initially all zero, the outputs of the design in Fig. 2 with respect to a given input sequence for three cycles is displayed in Table 1. In cycle 0, primary outputs are 00, whereas the next latch values are 110 based on the given primary inputs. In cycle 1, the next latch values of cycle 0 become the current latch values of cycle 1, that is, current latch values are 110, hence the primary outputs become 01. The next latch values become 111 based on the given primary inputs. In cycle 2, the next latch values of cycle 1 become the current latch values of cycle 1, that is, current latch values are 111, hence the primary outputs become 11.

4 Parallel Cycle-Based Simulation Using GPUs

Algorithm 2 displays the pseudocode for our parallel CBS algorithm. Our algorithm has compilation and simulation phases. In compilation phase, we extract combinational elements of a design since we can simulate combinational and sequential elements separately as described in Algorithm 1. Next, we levelize the extracted combinational design in order to simulate elements at a level in parallel. Then, we partition the levelized design into clusters, which are sets of gates that we will define later. We then balance these clusters in order to maximize CUDA thread usage and map these balanced clusters to CUDA blocks. We developed two different clustering algorithms that we will explain below. After obtaining CUDA blocks, we are ready for parallel simulation. In parallel simulation phase, we execute the test-bench in order to generate primary input values at each cycle, next we transfer these values to the GPU. Then, we simulate in parallel CUDA blocks for combinational and sequential elements. Below we provide the details of our algorithm.

Algorithm 2 Parallel Cycle-Based Simulation Algorithm

Input: circuit, number of simulation cycles *num*, test-bench
Output: output values
// compilation phase
1: extract combinational elements;
2: levelize combinational elements;
3: cluster combinational and sequential elements;
4: balance clusters;
// parallel simulation phase
5: **for** i = 0 to *num* **do**
6: execute test-bench on the host and generate primary input values;
7: transfer inputs from the host to the device;
8: simulate all clusters on the device and generate primary output values;
9: transfer outputs from the device to the host;
10: **end for**

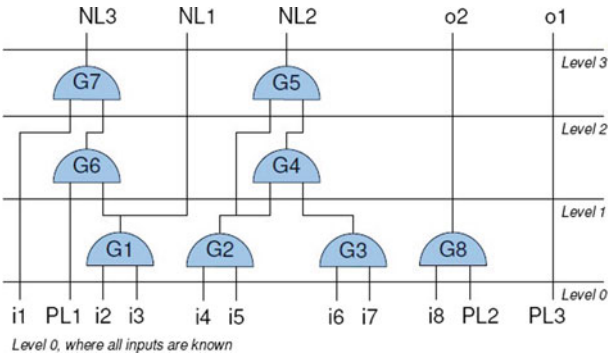


Fig. 3 Levelization of the circuit in Fig. 2. The *upper* levels are dependent on the *lower* levels, whereas the gates in the same level are independent of each other

4.1 Levelization

In CBS, we can simulate combinational and sequential elements separately. A speed-up technique for the simulation of combinational elements is to simulate them level by level, where the *level of a gate* is defined as the largest distance from the primary inputs and the sequential elements of the design. For example, the level of gate $G7$ in Fig. 2 is 3. The level of a gate also describes its evaluation order with respect to other gates, that is, a lower level gate is simulated before a higher level gate. Hence, a level encodes the dependency relation between gates in the design where the input of a gate can be the output of another gate from a lower level. This levelization enables parallelization of CBS since the simulation of gates in the same level are independent of each other and can be done in parallel.

Figure 3 displays the design in Fig. 2 after levelization step. Note that there are no latches displayed in the figure since levelization is done on the combinational extraction of the design. We also assume that all latch values and primary input values are given initially. Hence, latches and inputs can be considered to be in level 0. In the figure, we denote the present and next latch values by *PL#* and *NL#*, respectively.

4.2 Clustering and Balancing of Gates

Once the design is levelized, we can partition both the combinational and sequential elements into sets. We call each such set a *cluster*. Our goal is to generate clusters where each cluster can be simulated independently of other clusters. Then we can simulate each gate in a level of a cluster by a separate thread since simulation of gates in the same level are independent of each other.

We used several heuristics for clustering while exploiting the GPU architectural properties. In particular, we want to maximize the usage of available CUDA threads. Furthermore, CUDA allows thread synchronization, that is, threads within a block can be synchronized; however, threads in different blocks cannot be synchronized. An attempt to distribute all gates such that each level is simulated by a separate block would require the synchronization of different blocks since each level has to be completely simulated before the next level. Hence, this approach is not possible.

4.2.1 First Clustering Algorithm

Figure 4 is a conceptual view of the result of our first clustering algorithm. Each triangle represents a cluster. We use a threshold value, explained below, in order to control the number of CUDA blocks. The intersections of clusters represent the design elements that are duplicated. This duplication is necessary to ensure the simulation independence between the clusters.

Algorithm 3 displays the pseudocode of our first clustering algorithm. In this algorithm we assign each cluster to a CUDA block. First, starting from the gates at the highest level, we search for the level where the number of gates in that level is greater

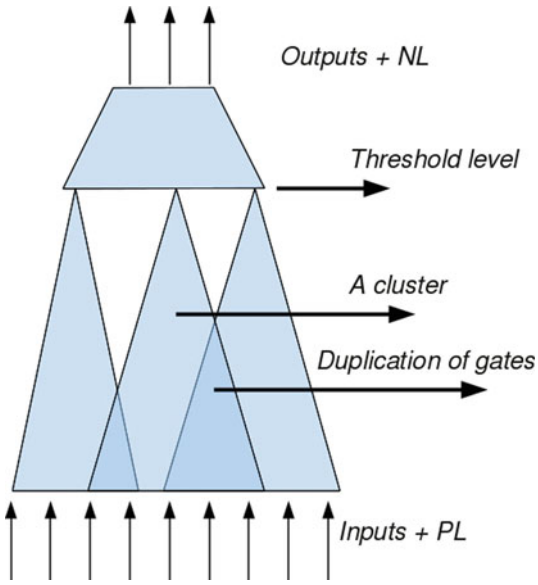


Fig. 4 Visualizing clustering Algorithm 1. The threshold level is the level where the number of gates is at least the number specified by clustering threshold. The top corner of each triangle represents a gate in the threshold level

Algorithm 3 Pseudocode for First Clustering Algorithm

Input: levelized circuit, threshold value *threshold*
Output: cluster of gates
1: *level* = the highest level of gates in circuit;
2: **repeat**
3: **if** number of gates in *level* > *threshold* **then**
4: add the gates from the highest level until the current level to *clusterH*;
5: break; // found threshold level
6: **end if**
7: level = level − 1;
8: **until** level = 0;
9: **if** *level* = 0 **then**
10: update threshold value and rerun above steps;
11: **end if**
12: **for** i = 1 to number of gates in *level* **do**
13: add the cone of logic for gate Gi to *clusterGi*;
14: **end for**
15: *clusterRem* = ⋃ remaining combinational gates;
16: *clusterSeq* = ⋃ sequential gates;

than or equal to a given threshold value. We call this level the *threshold level*. This threshold value is crucial in obtaining a good partitioning of the gates. We add all the gates from the highest level up-to but not including the threshold level into a single cluster (*clusterH*). Once we reach the threshold level, we determine the cone of logic of every gate in that level, where the *cone of logic* of a gate is the set of elements encountered during a backtrace from a gate to inputs or sequential elements. For example, the cone of logic of gate $G5$ in Fig. 3 includes $G5$, $G4$, $G3$, $G2$. Each cone of logic is a cluster for our purposes, named *clusterGi* for each gate Gi in threshold level. Note that since gates can be in the cone of logic of more than one cluster they may need to be duplicated for each cluster. This duplication allows us to simulate each cluster independent of other clusters. After these steps, there may still be some gates that are not part of any cluster. In particular, there may be combinational gates that are not in the cone of logic of any gate at the threshold level. For example, gate $G8$ is one such gate. Hence, we add such gates to *clusterRem*. Also, we add all sequential elements to a separate *clusterSeq*. At the end of the clustering algorithm we have the following clusters, *clusterH*, *clusterGi* for each gate Gi in threshold level, *clusterRem*, and *clusterSeq*.

Although most clusters can be simulated independent of each other, there is a *dependency* among some clusters. In particular, *clusterH* can be simulated only after all *clusterGi* have been simulated. This is because all *clusterGi* are in the cone of logic for gates in *clusterH*, hence *clusterH* is dependent on all *clusterGi*. Furthermore, *clusterSeq* for sequential elements can be simulated only after all combinational clusters have been simulated since the next latch values can only be calculated after all combinational clusters have been simulated.

In Fig. 5, we apply clustering algorithm on the design in Fig. 3. We assume that the threshold value is 2, in which case the top level (level 3) automatically becomes the *threshold level*. Clusters 1 and 2 are obtained by finding the cone of logic of gates in level 3. Whereas, cluster 3 is a cluster of remaining combinational elements. Similarly,

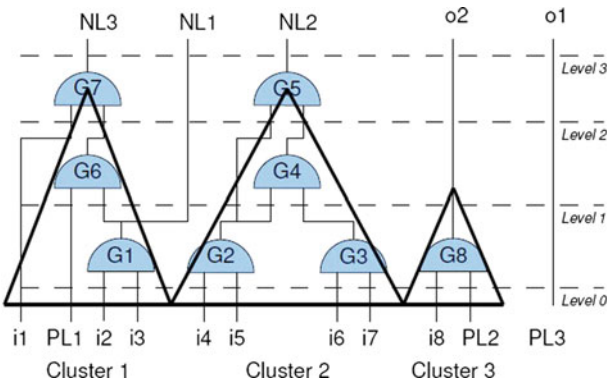
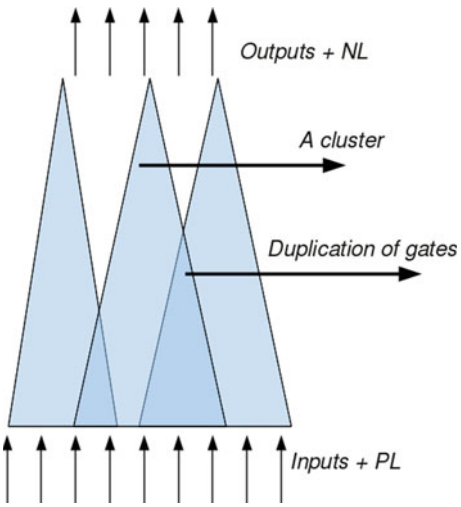


Fig. 5 Clustering Algorithm 1 on the levelized design in Fig. 3. Distinct clusters are independent of each other. However, there is a dependency between levels within a cluster

Fig. 6 Visualizing clustering Algorithm 2



there is a cluster 4 that includes all latches. Finally each cluster is assigned to a CUDA block.

It is clear that we can control the number of CUDA blocks using the threshold value with this clustering algorithm. However, this becomes a manual effort and some designs may not have as many gates as specified in the threshold value or may have many more gates than the threshold value. Furthermore, clustering of remaining gates needs extra effort of searching and finding out the gates that are not part of any cluster. In such cases, the first clustering approach may not be useful. Hence, we developed an optimized approach that allows us to better control the number of CUDA blocks, the shared memory and ultimately obtain better speedups.

4.2.2 Second Clustering Algorithm

In this algorithm, we build clusters starting from the primary outputs and latches instead of starting from the user given threshold level as in the first clustering algorithm (Fig. 6). This allows us not to have any remaining gates as was the case above. Also, since designs can have varying number of outputs and latches, the number of clusters and the sizes of clusters can vary. This necessitates further steps to optimize thread usage. For this purpose, we develop new steps for merging and balancing of clusters. This avoids assigning every cluster to a CUDA block as in the previous algorithm, rather we reshape clusters into an optimized format then assign those clusters to CUDA blocks.

Algorithm 4 shows our second clustering algorithm. We next describe the steps in the algorithm in greater detail.

- *Step 1. Obtain clusters* In this algorithm we do not assign each cluster to a CUDA block as was done in the previous clustering algorithm. Rather, our goal is to build many clusters starting from primary outputs and latches. We add the cone of logic of both primary outputs and latches and form clusters, named $clusterOL$. However, when going back from a gate, we stop at the latches. At the end of this step, the number of clusters is the sum of the number of primary outputs and the number of latches. Figure 7 displays application of this step on the levelized design in Fig. 3.
- *Step 2. Merge clusters* After the above step is completed, there may be clusters with an imbalance in distribution of gates to clusters. Hence, we need to merge

Algorithm 4 Pseudocode for Second Clustering Algorithm

Input: levelized circuit, number of CUDA blocks $numBlocks$
Output: merged and balanced cluster of gates
// Step 1. obtain clusters
for each primary output or latch OL **do**
  add the cone of logic for OL to $clusterOL$;
end for
// Step 2. merge clusters
$gate\_limit$ = total number of gates in all $clusterOL$/number of blocks;
compute overlaps of all possible pairs of clusters;
for $i = 1$ to $numBlocks$ **do**
  let $clusterCL$ be an arbitrary unmarked cluster and $mergedCL[i] = \emptyset$;
  $mergedCL[i] = mergedCL[i] \cup clusterCL$;
  mark $clusterCL$;
  repeat
    $clusterCLi$ = an unmarked cluster with maximum overlap with $clusterCL$;
    $mergedCL[i] = mergedCL[i] \cup clusterCLi$;
    mark $clusterCLi$;
  until number of gates in $mergedCL[i]$ reaches $gate\_limit$;
  delete duplicate gates in each $mergedCL[i]$;
end for
// Step 3. balance merged clusters
$level\_width$ = average width of all merged clusters;
for each merged cluster $mergedCL[i]$ **do**
  move gates in $mergedCL[i]$ such that the cluster has a $level\_width$ wide rectangular shape;
end for

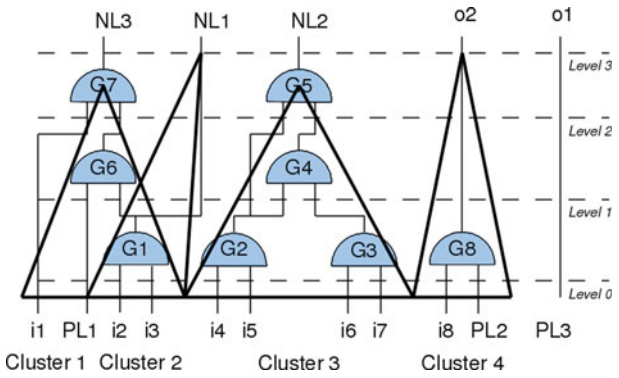


Fig. 7 Clustering Algorithm 2 before merging and balancing steps

clusters in order to reduce this imbalance in the sizes of the clusters. This step will also allow us to optimize the usage of shared memory as we describe below. We respect the following conditions for merging clusters.

1. The number of merged clusters is a given preset number of CUDA blocks.
2. The size of a merged cluster may not exceed a certain limit, called $gate\_limit$. The $gate\_limit$ is obtained by dividing the total number of gates in all clusters (including duplicate gates) by the number of CUDA blocks.
3. We merge clusters with maximum number of common gates (overlap).

We choose the preset number of CUDA blocks according to the properties of the circuit and the CUDA device at hand. In our case, we chose to generate 48 CUDA blocks for large circuits. This is because each multiprocessor can execute 768 threads in parallel and given that a CUDA block can not have more than 512 threads, we allocate 384 threads to two blocks for each multiprocessor for maximum thread utilization. For a CUDA architecture with 24 multiprocessors, this results in 48 blocks in total. This preset number of blocks is fine for most industrial designs as can be seen from experiments. However, for designs with fewer number of gates, fewer number of blocks can be allocated such as 12 or 24.

Next, we use a simple heuristic to merge clusters. We compute the overlaps of all possible pairs of clusters in a two-dimensional matrix. Then, for each CUDA block, we start with an arbitrary cluster and add the cluster with the maximum overlap to it. This addition procedure continues until we reach a given $gate\_limit$ for each merged cluster.

After merging the clusters based on the above heuristic, there may be duplicate gates in merged clusters, so we delete these duplications. Figure 8 displays the effect of the merging step.

– Step 3. *Balance merged clusters* At the end of the above step, we obtain merged clusters that are of triangular shape. In a triangular shape cluster, more threads will become idle as the level increases during execution, since there are more gates at the lower levels. Our goal is to utilize available threads in the most efficient way such that all threads will approximately have the same computation burden for

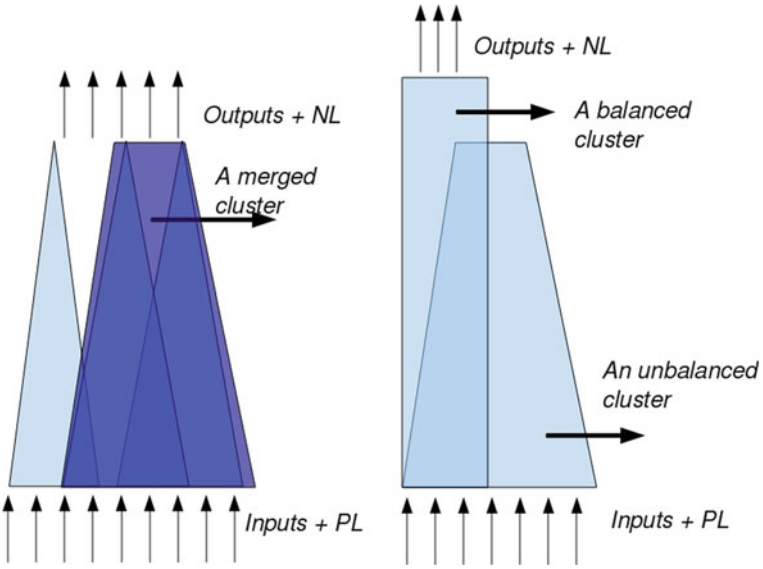


Fig. 8 Merging and balancing clusters results in using all available threads efficiently at all levels

each level. A uniform shape like a rectangular allows us to have the same number of gates at each level. Figure 8 displays the effect of the balancing step.
In order to balance merged clusters, we compute the average width of all merged clusters, called $level\_width$. This width will give us the number of threads that should be allocated to each CUDA block. We make sure that this is a multiple of 16 since for coalesced memory access half warp (16 threads) can work at the same time. If this width is greater than 384, which is the maximum number of threads that we allow for a block, we set 384 as the width value. Since $level\_width$ is determined as an average, there may be levels within the blocks having more gates than this width. We aim to reshape the merged clusters such that no level will have gates more than this width value. When we have a level having gates more than the width value, the extra gates will be moved to one level up. To preserve the dependency between gates, the levels of all gates that are dependent on those moved gates are also incremented. At the end of this iterative procedure, we will have rectangular shaped merged clusters with the same width but with differing heights. Note that there may be fewer gates than the width at a given level. For such levels, we add dummy gates that have no operational burden.

Finally, each merged and balanced cluster is assigned to a CUDA block.

4.3 Parallel Simulation Phase

Parallel simulation phase consists of several steps that can be seen in Algorithm 2. During each cycle, input values are transferred from host to device. The inputs are essentially generated by executing the test-bench on the host. In our experiments,

Algorithm 5 Parallel Block Simulation Kernel Pseudocode

```
1: _ _global_ _ void simulationKernel(...){
2:
3: _ _shared_ _ unsigned char vars[8000]; // keep encoded intermediate values
4: numberOfThreads = blockDim.x;
5: index = threadIdx.x; start = blockIdx.x * numberofThreads;
6: vars = coalesced read of encoded values from global device memory;
7: for i = 0; i < level; i++ do
8:    // coalesced read gate variables from global device memory
9:    i1 = firstInput[start+index];
10:   i2 = secondInput[start+index];
11:   o = output[start+index];
12:   decode variable i1 as v1, i2 as v2;
13:   val = v1 ∧ v2;
14:   write encoded val to vars array;
15:   index+ = numberOfThreads;
16:   _ _syncthreads();
17: end for
18: coalesced write vars to global device memory;
19: }
```

we mainly used random test-benches, hence the input generation could be optimized as described in the next section. Once the inputs are ready, all CUDA blocks can be executed by executing a kernel function. An execution of a block with combinational elements proceeds level by level and after each level is simulated the threads of the block synchronize using a barrier. This process continues until all levels are completed. We used coalesced memory access for reading and writing gate variables between the global memory and the CUDA processors. We increased the number of gates that can be efficiently simulated in a block by encoding the intermediate gate outputs and storing these in the shared memory. Also, the intermediate gate output values are stored in shared memory in an encoded fashion, which we describe in next section. At the beginning of a cycle, blocks fetch relevant design data (gate variables) and input values from device memory in a coalesced manner. At the end of the cycle, output and latch values are transferred from device to host. In particular, we stored the design structure, primary inputs, primary outputs and latches in the device memory and frequently accessed data structures such as intermediate and-gate values of a block in the shared memory. Our particular simulation kernel function is shown in Algorithm 5. We next describe our CUDA optimizations in more detail.

5 CUDA Optimizations

We applied several optimization methods to increase performance of our algorithms. To develop an effective CUDA program, one needs to have knowledge of the GPU architecture such as shared memory, memory coalescing and memory transfer overhead. Communication overhead is a major cost, so we exploited CUDA's faster shared memory, register and pinned memories in our implementation.

In CUDA, one can allocate memory space in host memory dynamically by using $CudaMallocHost$ function instead of the ordinary C $malloc$ function. This uses

the pinned (paged-locked) memory. Allocating memory with $CudaMallocHost$ increases the performance of $CudaMemcpy$ operation. We used these functions to allocate data structures for our designs.

One of the most important optimizations for CUDA programming is to make effective use of shared memory. Fetching a variable from device memory requires 400–600 cycles, whereas fetching it from the shared memory requires only a few cycles. During the execution of a CUDA block, we copied the most frequently accessed data structures to the shared memory such as intermediate and-gate output values, which are the values obtained after simulating one level of a block. Using the global memory for this purpose will result in a huge time penalty, hence we write these intermediate values to the shared memory instead.

For designs with random input generation (random test-benches), we moved the test-bench code into the GPU and enabled parallel input generation. This removes the communication need between CPU and GPU. For designs without a random input generation, we used another optimization method where the inputs of the design and outputs of the design are not transferred between GPU and CPU during every cycle, rather these data transfers are done in bulks such as after every 10,000 cycles. Although this optimization increases the data size to be copied between the host and the device, our experiments have proven that decreasing the number of $CudaMemcpy$ operations is more critical than increasing the data size.

Also, since we use AIGER format with only a single type of combinational gate (an and-gate), we do not need to store the truth table for other types of complex gates in the shared memory, wasting valuable space.

In addition to the above optimizations, we incorporated the following optimizations with our second clustering algorithm.

- We developed a new clustering and balancing algorithm that makes maximum use of all available threads such that every thread is doing the same task without divergent paths. We accomplished this by first merging the clusters and then balancing them to reshape into rectangular uniform structures as described in Sect. 4.2.2.
- We heavily made use of coalesced memory access for reading and writing variables between the global memory and the CUDA processors.
- We increased the number of gates that can be efficiently simulated by encoding the intermediate gate outputs and storing these in the shared memory.

We now describe these optimizations in more detail.

During parallel simulation, we need to access the inputs and outputs of the gates. To access this information inside the kernel in a time-efficient manner, we put gate information on the device global memory and access this information in a coalesced manner. When accessing device global memory, peak performance utilization occurs when all threads in a half warp (16 threads) access continuous memory locations. For coalesced access, we have to adjust the data structures that hold gate information. Therefore, we allocate new arrays for indices of each gate; one for first input, one for second input and one for the output. These data structures are such that all gates in a level are adjacent to each other so that when the threads access device global memory for gate information they are accessed in a coalesced manner. Figure 9 shows this coalesced memory access.

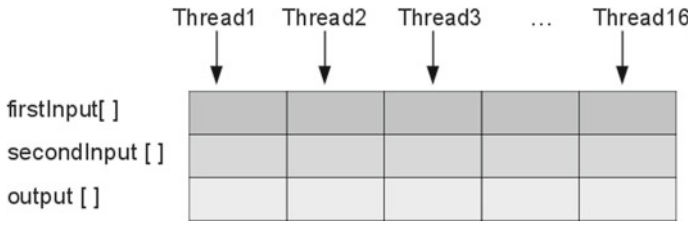


Fig. 9 Coalesced memory access

Above we described how we used shared memory for fast access. However, shared memory size is limited. Each multiprocessor can have 16 KB of shared memory, when we use 2 blocks for each multiprocessor then each will have 8 KB of shared memory. If we use one char variable for each gate output value, we can have at most 8,000 variables for a block (8 KB max shared memory for a block), and therefore the total number of variables in a design can be at most 48 blocks * 8,000 variables = 384,000 variables. This number may be small for large designs. Therefore, we developed an efficient representation to keep gate output values by using 1 unsigned char variable (8 bits) for 8 gate outputs since a gate output can only be 0 or 1. At the start of each block simulation, we bring these encoded variables from device global memory in a coalesced manner as well. With this representation, we can have 64,000 variables for a block and in total 48 blocks * 64,000 variables = 3,072,000 variables. This is equal to the sum of the number of and-gates, latches, primary inputs and outputs in a design. If we assumed that there is a 30% duplication of gates among blocks, then we can simulate designs with up to 2 million gates approximately. Similarly encoded variables are read and written in a coalesced manner.

6 Experimental Results

We validated the effectiveness of our two GPU based parallel CBS algorithms on several test cases. We used test cases from IWLS [17], OpenCores [24] and AIGER [2]. These test cases include ldpc-encoder (low density parity check encoder), des-perf (triple DES optimized for performance), wb-conmax (wishbone conmax IP core), aes-core (AES Encryption/Decryption IP Core) pci-bridge32, ethernet, vga-lcd (wishbone compliant enhanced vga/lcd controller), and several other designs. These test cases either had their own test benches or we generated random test benches for them. Table 2 displays the list of test cases that we used in our experiments and their design characteristics including the number of levels and the number of CUDA blocks used for their simulation for both clustering algorithms. For some test cases, we generated AIGER format of designs using ABC tool [1].

We performed our experiments on an Intel Xeon CPU with two 2.27 GHz multi-processors, 32GB of memory and a CUDA–enabled NVIDIA Quadro FX3800 GPU with 1 GB device memory and 24 streaming multiprocessors each with 8 streaming processors.

Table 2 Experimental test cases

Design	Vars	Inputs	Latches	Outputs	And-Gates	Levels	Blocks1	Blocks2
ldpc-encoder	220,684	1,723	0	2,048	218,961	19	288	48
vga-lcd	143,879	89	17,079	109	126,711	23	145	48
des-perf	100,500	17,850	0	9,038	82,650	19	373	48
ethernet	80,326	98	10,544	115	69,684	31	364	48
wb-conmax	49,753	1,130	770	1,416	47,853	26	261	24
tv80	7,665	14	359	32	7,292	39	34	24
aes-core	22,841	1,319	0	668	21,522	25	126	24
ac97-ctrl	12,624	84	2,199	48	10,341	8	495	24
pci-bridge32	26,305	162	3,359	207	22,784	29	146	24
system-cdes	2,813	132	190	65	2,491	22	48	24
pci-spoci-ctrl	880	25	60	13	795	14	29	24
sasc	740	16	117	12	607	7	49	24

Table 3 Experimental results for 100,000 cycles

Design	SEQ(sec)	PAR1(sec)	PAR2(sec)	Speedup1	Speedup2
ldpc-encoder	382.21	74.79	24.32	5.11	15.72
vga-lcd	223.15	45.91	25.45	4.86	8.77
des-perf	180.62	65.68	8.53	2.75	21.17
ethernet	155.51	52.89	14.96	2.94	10.40
wb-conmax	94.81	33.96	10.8	2.79	8.78
tv80	92.59	23.31	10.63	3.97	8.71
aes-core	83.64	35.63	7.97	2.35	10.49
ac97-ctrl	58.66	18.87	4.74	3.11	12.38
pci-bridge32	50.12	21.7	9.39	2.31	5.34
system-cdes	30.44	11.55	6.92	2.64	4.40
pci-spoci-ctrl	7.04	10.44	5.13	0.67	1.37
sasc	6.26	6.34	3.75	0.99	1.6

In Table 3, we demonstrate our experimental results. Column SEQ denotes the results for sequential simulation using the default simulator that is available with AIGER [2]. Columns $PAR1$ and $PAR2$ denote the results for our parallel simulation algorithms using the first and the second clustering algorithms, respectively. Columns $Speedup1$ and $Speedup2$ denote the speedup of parallel algorithms over the sequential algorithm. The times do not include the compilation phase in both cases but include the time required to transfer the data between the GPU and the CPU. Figure 10 graphically displays the speedups.

We simulated the designs for different number of cycles ranging from 100, 500 K to 1 M. Figure 11 shows that the speedups are similar for different number of cycles for

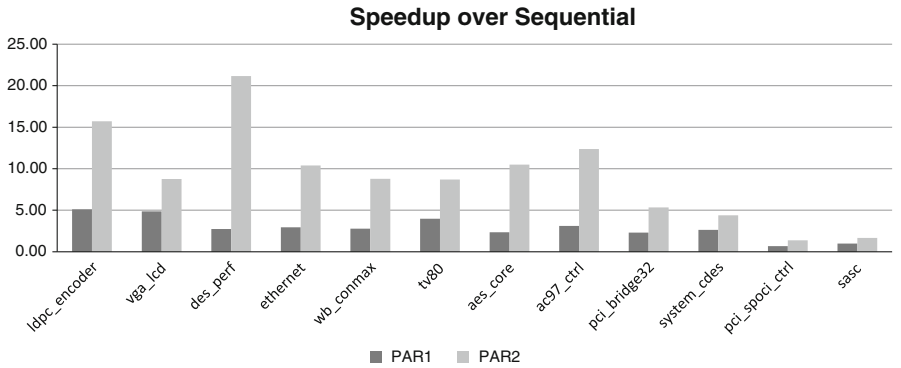
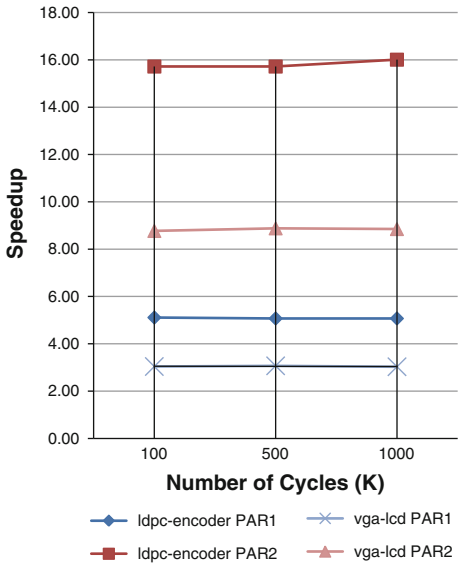


Fig. 10 Parallel simulation speedups over sequential simulation for 100,000 cycles

Fig. 11 Parallel simulation speedups versus number of cycles



ldpc-encode and vga-lcd. We obtained similar results for other testcases. We present results for 100K cycles in the rest of the experiments.

Different speedups are expected for logic simulation due to the fact that logic simulation is heavily influenced by the circuit structure. We obtained speedups up-to 5x for PAR1 and up-to 21x for PAR2. Both PAR1 and PAR2 outperform SEQ except for two cases where PAR1 performs worse than SEQ. In all test cases, PAR2 significantly outperforms both SEQ and PAR1.

In general, arbitrary circuit structures make it difficult to analyze designs and the corresponding speedups. However, Fig. 12 shows that both PAR1 and PAR2 speedups are proportional with the design size (the number of variables). Hence, further studies with larger designs hold promise. In fact, for smaller designs such as pci-spoci-ctrl and sasc, PAR1 performs worse than the sequential algorithm. Figure 13 shows that

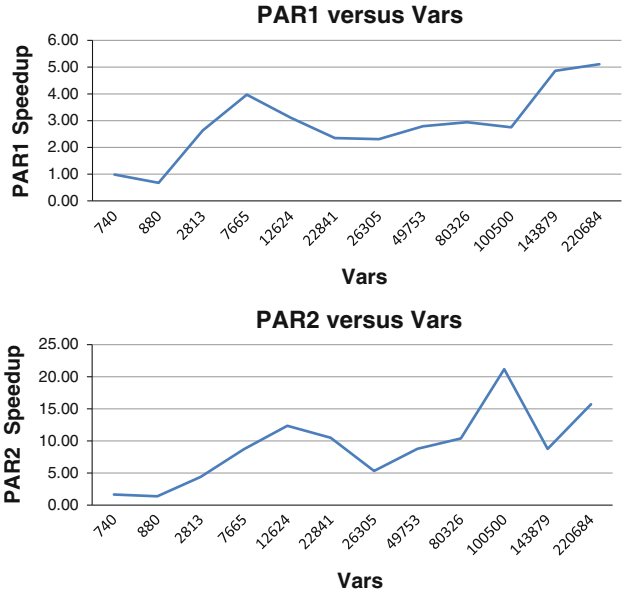


Fig. 12 Parallel simulation speedups versus design size

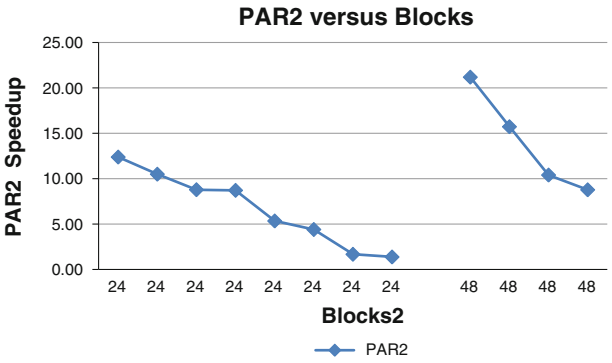


Fig. 13 Parallel simulation speedup using second clustering versus number of CUDA blocks

the speedup is better when the number of blocks is increased from 24 to 48 for PAR2. This is related to the fact that the higher number of blocks is chosen for larger designs in PAR2. We also observe from Fig. 10 that three of the top four speedups for PAR2 are obtained for designs with no latches such as ldpc-encoder, des-perf, and aes-core. This is expected since there is no need to start another GPU kernel for latches after and-gate simulation.

For PAR1, we experimented with the threshold values. We observed that the threshold value is dependent on the circuit structure as well. For example, for most testcases including sasc the execution time changes almost linearly with the threshold value.

We observed that the number of gates in the designs can be higher using AIGs than the other gate level formats. Although our experiments showed that AIGs are effective in reducing the shared memory size, AIGs may result in higher number of levels that may lead to execution overheads. So further synthesis steps in ABC tool can help reduce the number of levels.

Our parallel algorithm compilation phases may spend more time than the sequential algorithm. Once the compilation is done, it is saved so that simulations with different test benches or parameters or number of cycles can be done without incurring the compilation cost again. In practice, once the design matures, simulations are run until the design product is taped-out.

7 Conclusions

We introduced two novel parallel cycle-based simulation algorithms for digital designs using GPUs. This work has further progressed the state of the art in logic simulation. Our algorithms result in faster, more efficient parallel logic simulators that can run on commodity graphics cards allowing verified designs while obtaining significant reduction in the overall design cycle. Our approach leverages the GPU architecture by optimizing on low latency memory spaces, and reducing host and device communications during compilation and simulation phases of a cycle based simulation algorithm. Our approach is unique in that we use the AIG representation for electronic designs that proves to be very efficient for boolean functions.

We obtained speedups ranging from $5\times$ to $21\times$ with our first and second algorithms, respectively, for various benchmarks. Our first clustering algorithm uses a given threshold to generate independent CUDA blocks for parallel simulation. Whereas, our second clustering algorithm automatically generates CUDA blocks while using sophisticated merging and balancing steps in order to maximize the number of parallel threads that can be executed on the GPU. Our second algorithm also improves on the first by better utilization of shared memory and memory coalescing as well as supporting larger design sizes, which is crucial for practical applications. Since logic simulation is an activity that continues nonstop until the design is productized such speedups have a big impact on the verification and the overall design cycle. Our results show that we obtain better speedups with larger circuits. Hence, further studies with industrial designs hold promise.

As a future work, we want to investigate other gate level design formats and develop event-based simulation algorithms that are also commonly used in the industry. We also want to experiment with complex industrial test cases that may not fit on a single GPU device.

Acknowledgments We would like to thank Alan Mischenko from the University of California, Berkeley for suggesting and providing AIG benchmarks. This research was supported by a Marie Curie European Reintegration Grant within the 7th European Community Framework Programme and BU Research Fund 5483.

References

1. ABC web site. http://www.eecs.berkeley.edu/~alanmi/abc/
2. AIGER Format web site. http://fmv.jku.at/aiger/
3. Alpert, C.J., Kahng, A.B.: Recent directions in netlist partitioning: a survey. Int. VLSI J. **19**(1–2), 1–81 (1995)
4. Bailey, M.L., Briner, J.V. Jr., Chamberlain, R.D.: Parallel logic simulation of VLSI systems. ACM Comput. Surv. **26**(3), 255–294 (1994)
5. Bergeron, J.: Writing Testbenches—Functional Verification of HDL Models. Springer, Berlin (2003)
6. Brayton, R., Mishchenko, A.: ABC: an academic industrial-strength verification tool. In: Proceedings of the International Conference on Computer-Aided Verification (CAV) (2010)
7. Catanzaro, B., Keutzer, K., Su, B.Y.: Parallelizing CAD: a timely research agenda for EDA. In: Proceedings of the Design Automation Conference (DAC), pp. 12–17. ACM (2008)
8. Chatterjee, D., DeOrio, A., Bertacco, V.: Event-driven gate-level simulation with GP-GPUs. In: Proceedings of the Design Automation Conference (DAC), pp. 557–562 (2009)
9. Chatterjee, D., DeOrio, A., Bertacco, V.: GCS: High-performance gate-Level simulation with GPGPUs. In: Proceedings of the Conference on Design Automation and Test in Europe (DATE), pp. 1332–1337 (2009)
10. Chatterjee, S., Mishchenko, A., Brayton, R.K., Wang, X., Kam, T.: Reducing structural bias in technology mapping. IEEE TCAD **25**(12), 2894–2903 (2010)
11. Che, S., Boyer, M., Meng, J., Tarjan, D., Sheaffer, J.W., Skadron, K.: A performance study of general-purpose applications on graphics processors using CUDA. J. Parallel Distrib. Comput. **68**(10), 1370–1380 (2008)
12. Croix, J.F., Khatri, S.P.: Introduction to GPU programming for EDA. In: Proceedings of the International Conference on Computer Aided Design (ICCAD), pp. 276–280. ACM (2009)
13. NVIDIA CUDA web site. http://www.nvidia.com/CUDA
14. Deng, Y.S., Wang, B.D., Mu, S.: Taming irregular EDA applications on GPUs. In: Proceedings of the International Conference on Computer Aided Design (ICCAD), pp. 539–546. ACM (2009)
15. Hering, K.: A Parallel LCC Simulation System. In: Proceedings of the International Parallel and Distributed Processing Symposium (IPDPS) (2002)
16. Hering, K., Reilein, R., Trautmann, S.: Cone clustering principles for parallel logic simulatio. In: International Workshop on Modeling, Analysis, and Simulation of Computer and Telecommunication Systems, pp. 93–100 (2002)
17. IWLS 2005 Benchmarks. http://www.iwls.org/iwls2005/benchmarks.html
18. Johannes, F.M.: Partitioning of VLSI circuits and systems. In: Proceedings of the Design Automation Conference (DAC), pp. 83–87. ACM (1996)
19. Meister, G.: A survey on parallel logic simulation Technical report. Department of Computer Engineering, University of Saarland, Saarland (1993)
20. Mishchenko, A., Brayton, R., Jang, S.: Global delay optimization using structural choices. In: FPGA '10: Proceedings of the 18th Annual ACM/SIGDA International Symposium on Field Programmable Gate Arrays, pp. 181–184. ACM (2010)
21. Mishchenko, A., Chatterjee, S., Brayton, R.: DAG-aware AIG rewriting: a fresh look at combinational logic synthesis. In: Proceedings of the Design Automation Conference (DAC) (2006)
22. Nguyen, H.: Gpu Gems 3. Addison-Wesley Professional, Reading (2007)
23. OpenCL web site. http://www.khronos.org/opencl/
24. Opencores Benchmarks. http://www.opencores.org
25. Perinkulam, A.: Logic Simulation Using Graphics Processors. Master's thesis, University of Massachusetts Amherst (2007)
26. Pfister, G.: The Yorktown simulation engine: introduction. In: Proceedings of the Design Automation Conference (DAC) (1982)
27. Sen, A., Aksanli, B., Bozkurt, M., Mert, M.: Parallel cycle based logic simulation using graphics processing units. In: Proceedings of the International Symposium on Parallel and Distributed Computing (ISPDC) (2010)
28. Zhu, Q., Kitchen, N., Kuehlmann, A., Sangiovanni-Vincentelli, A.: SAT sweeping with local observability don't-cares. In: Proceedings of the Design Automation Conference (DAC) (2006)