Accelerating Gate Sizing using GPU

Yi-Hua Chung[1], Nahmsuk Oh[2],
Malleswara Gupta Balabhadra Naga Venkata[2], Aditya Shiledar[2], Sudipto Kundu[2], Vishal Khandelwal[2], and Tsung-Wei Huang[1]

[1] University of Wisconsin-Madison, Madison, WI, USA
{yihua.chung, tsung-wei.huang}@wisc.edu
[2] Synopsys Inc, Sunnyvale, CA, USA
{nahmsuk, mgupta, adityas, skundu, vishalk}@synopsys.com

Abstract. Gate sizing is important in VLSI design to optimize performance and meet timing constraints. Multi-core CPU-based approaches have been widely used to speed up the gate sizing algorithm, but their scalability is typically limited to 8–16 threads. To address this limitation, we propose a GPU algorithm to accelerate a time-consuming routine of gate sizing, namely the library cell (libcell) selection process, in an industrial-standard sizer. By leveraging both block- and warp-level parallelism, our algorithm can greatly accelerate the libcell selection time. Experimental results show that our GPU implementations achieve up to 38.13× speedup over a 16-core CPU baseline, while warp-level sizing can further achieve additional 4.77% improvement over block-level sizing.

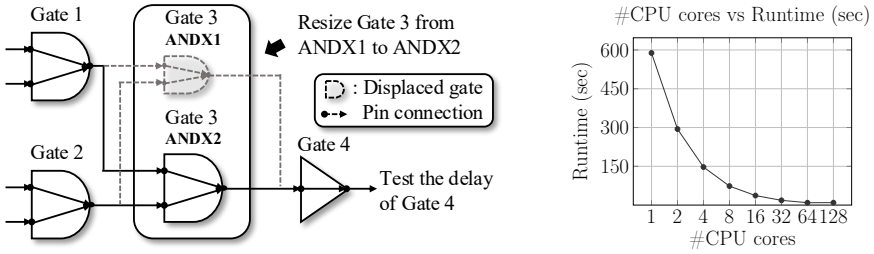
Keywords: Gate sizing, GPU parallel, Electronic Design Automation

1 Introduction

Gate sizing is a fundamental optimization step in VLSI design that directly impacts circuit performance, power consumption, and area [1, 2, 6]. It involves selecting the most appropriate standard cell size for each gate in a design to achieve optimal trade-offs between timing, power, and area constraints. Proper gate sizing helps ensure that critical timing paths meet setup and hold requirements while minimizing power overhead.

Given a set of independent standard cells, each with associated input and output pins, the goal of gate sizing is to select the optimal library cell (libcell) for each gate to minimize timing violations and meet design constraints [3, 4]. For example, as shown in Figure 1a, we assign the libcell for Gate 3 from size X1 (cell ANDX1) to size X2 (cell ANDX2) to minimize the overall timing delay.

As the circuit complexity continues to grow, solving a gate-sizing problem can be very time-consuming. To mitigate the runtime challenge, industrial sizers have leveraged CPU parallelism to speed up various steps in the gate-sizing algorithm [6, 8, 9]. In particular, the *libcell selection* process has been widely studied, as it exhibits a high degree of parallelism by selecting an optimal libcell for each gate independently [1, 6, 8, 9]. Despite performance improvement, the



(a) (b)

Fig. 1: (a) An example circuit illustrating the gate sizing optimization. Gate 3 acts as a driver, receiving inputs from Gate 1 and Gate 2 and driving the output to Gate 4. (b) Runtime performance of the CPU baseline with different numbers of CPU cores for processing 4090K driver pins, showing improved speedup with increasing cores, but saturation occurs beyond 64 cores.

scalability of CPU-parallel approaches is largely limited to 16 threads [9], as shown in Figure 1b. Compared to CPUs, modern GPUs provide a larger number of threads, enabling much higher parallel processing power [7]. This advantage inspires us to leverage GPU parallelism to accelerate the libcell selection process in gate sizing.

In this paper, we focus on accelerating the libcell selection routine from an industrial-standard gate sizer, which is originally implemented on multi-core CPUs. We collaborate with the vendor to directly integrate our GPU algorithm into the tool's environment. The libcell selection routine is executed within the inner loops of multiple steps in the sizer algorithm and accounts for a significant portion of the overall runtime. Due to intellectual property of the industrial tool, we cannot disclose other details of the algorithm but focus on our GPU kernel design.

2 ALGORITHM

We introduce two GPU kernels to accelerate the libcell selection process of gate-sizing: (1) block-level sizing and (2) warp-level sizing. These two approaches are designed to accommodate a wide range of GPU architectures while maximizing runtime efficiency. Block-level sizing targets GPUs without advanced warp-level primitives, focusing on distributing the workload at the thread-block level. Warp-level sizing targets GPUs that support warp-level primitives [5] to further achieve fine-grained synchronization and efficient communication during the libcell selection process.

2.1 Block-level Sizing

To accelerate the libcell selection process on GPU, we develop an efficient block-level sizing algorithm. Block-level sizing algorithm consists of two GPU kernels:

`update_delay` and `select_best_libcell`. We use the `update_delay` kernel to compute the delay of all libcells for each driver pin and `select_best_libcell` kernel to select the libcell with the smallest delay for each driver pin, where a driver pin is the output pin of a gate that connects to its downstream gates. The details of `update_delay` and `select_best_libcell` kernels are shown in Algorithm 1 and Algorithm 2.

As shown in Algorithm 1, the `update_delay` kernel calculates the delays for all libcells of driver pins in parallel by assigning each libcell to a separate thread (line 2 and line 4). All libcells of driver pins are stored in a 1D array, *libcells_of_driver_pins*, to ensure a coalesced memory access (line 4). After computing the delays using the `compute_delay` function, the results are stored in the output array *delay* (line 5). Next, in the `select_best_libcell` kernel, as shown in Algorithm 2, each thread processes the computed delays for its assigned driver pin, with all driver pins stored in the *driver_pins* array (line 2). Each thread identifies the libcell with the smallest delay and records its index as *idx* (line 4). Finally, the selected libcell at index *idx* is stored in the *best_libcell* array as the final selection (line 5).

Algorithm 1 update_delay

```
1: /* One thread handles one libcell of a driver pin */
2: #blocks = ⌈libcells_of_driver_pins.length() / 1024⌉; #threads = 1024
3: parallel for each thread tid {
4:     libcell = libcells_of_driver_pins[tid]
5:     delay[tid] = compute_delay(driver_pins, libcell)
6: }
```

Algorithm 2 select_best_libcell

```
1: /* One thread performs a reduction of all delays of a driver pin */
2: #blocks = ⌈driver_pins.length() / 1024⌉; #threads = 1024
3: parallel for each thread tid {
4:     idx = argmin_i{delay[i] | i ∈ driver_pins[tid]}
5:     best_libcell[tid] = the libcell at index idx within driver_pins[tid]
6: }
```

2.2 Warp-level Sizing

To further accelerate the process of reduction on GPU, we develop an efficient warp-level sizing algorithm, where each warp consists of 32 threads that execute instructions simultaneously. This algorithm consists of two GPU kernels: `update_delay` and `warp_select_best_libcell`. After using `update_delay`

kernel to get the computed delays, we use the `warp_select_best_libcell` kernel to perform a reduction operation to select the libcell with the smallest delay for each driver pin. Unlike the `select_best_libcell` kernel in block-level sizing, the `warp_select_best_libcell` kernel leverages highly optimized warp-level primitives, enabling fine-grained synchronization and efficient communication among threads within the same warp for faster reduction. The details of `warp_select_best_libcell` kernel are shown in Algorithm 3.

In Algorithm 3, each driver pin is processed by a single warp. The variable *min_violation*, initialized to infinity, keeps track of the smallest delay found by each thread (*tid*) within a warp (*wid*) (line 4). The *libcells_by_driver_pin* array is a 2D array that stores all libcells by driver pin, allowing efficient access during processing (line 5). To process all libcells, threads within the same warp iterate through them in rounds of 32 (line 6), updating *min_violation* and *min_id*, which track the smallest delay and its corresponding libcell index (lines 7–11). After comparing all delays, a warp-level reduction is performed using the CUDA warp-level primitives *__reduce_min_sync* to efficiently identify and store the smallest delay among threads in the warp as *w_min* (line 12). The algorithm then uses another *__reduce_min_sync* operation to identify the index of the libcell with the smallest delay (*w_min_index*) for the assigned driver pin (lines 13–14). Finally, the thread corresponding to *w_min_index* stores the selected libcell in the *best_libcell* array as the final selection (line 15). By sharing data among threads within the same warp, warp-level reduction achieves faster synchronization and lower communication overhead compared to the block-level reduction.

Algorithm 3 warp_select_best_libcell

1: $\#blocks = \lceil driver\_pins.length() \times 32/1024 \rceil$; $\#threads = 1024$
2: /* One warp handles a driver pin */
3: **parallel for each warp in blocks** *wid* **{**
4: $min\_violation = \infty$; tid = GPU thread's index in warp
5: $len = libcells\_by\_driver\_pin[wid].length()$
6: $rounds = \lceil libcells\_by\_driver\_pin[wid].length()/32 \rceil$
7: **for each** $r \in \{1...rounds\}$
8: $libcell = libcells\_by\_driver\_pin[wid][r \times 32 + tid]$
9: $tmp \leftarrow$ time violation of libcell $(r \times 32 + tid)$ by driver pin
10: $min\_id = (min\_violation > tmp)\ ?\ (r \times 32 + tid) : (min\_id)$
11: $min\_violation = \mathbf{min}(min\_violation, tmp)$
12: $w\_min =$ **__reduce_min_sync**(0xFFFFFFFF, $min\_violation$)
13: $min\_index = (min\_violation == w\_min)\ ?\ min\_id$: len
14: $w\_min\_index =$ **__reduce_min_sync**(0xFFFFFFFF, min_index)
15: $best\_libcell[wid] = libcells\_by\_driver\_pin[wid][w\_min\_index]$
16: **}**

3 EXPERIMENTAL RESULTS

We implemented the CPU and GPU baselines in CUDA and C++ and compiled it using nvcc v12.3 with `-O3` and `-std=c++20` enabled. All experiments ran on a 4.8 GHz 64-bit Linux machine with 32 Intel Core i5-13500 cores and an Nvidia RTX A4000 GPU.

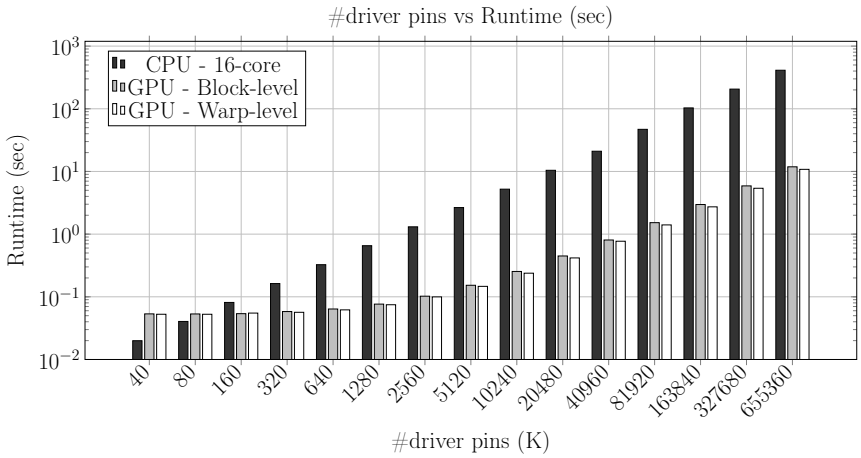


Fig. 2: Comparison of CPU and GPU runtime across different numbers of driver pins.

Figure 2 compares the runtime performance of CPU and GPU implementations across different numbers of driver pins. The figure presents results for three configurations: a 16-core CPU baseline, a GPU implementation with the block-level sizing algorithm, and an optimized GPU implementation with the warp-level sizing algorithm. The reported GPU runtime includes data transfer time between the CPU and GPU. In this experiment, we randomly generated driver pins and their associated libcells with varying counts and sizing parameters to have a simple and direct comparison against the CPU-based baseline. The number of driver pins is shown in the figure, while the number of libcells per driver pin follows distribution pattern: 70% of driver pins have between 1 and 32 libcells, while 30% have between 33 and 41 libcells.

The results show that the GPU-based implementations consistently outperform the 16-core CPU baseline when the number of driver pins exceeds 80K, with block-level sizing achieving an average 16.97× speedup and warp-level sizing achieving an average 18.17× speedup. As the number of driver pins increases, the speedup of GPU baselines grows larger, demonstrating their scalability and efficiency for handling larger workloads. For instance, at 655M driver pins, block-level sizing achieves a 34.75× speedup, while warp-level sizing achieves a 38.13× speedup over the CPU baseline. Additionally, warp-level optimization further im-

proves GPU performance, achieving an average 4.77% speedup over the block-level sizing, with the highest improvement of 8.88% observed at 655M driver pins.

4 Conclusion

In this paper, we introduced a GPU-accelerated algorithm to speed up the libcell selection routine in gate sizing, a critical step in VLSI design optimization. Our proposed algorithm addresses the scalability bottlenecks of multi-core CPUs by leveraging both block-level and warp-level GPU parallelism to independently evaluate and select the optimal libcell for each gate. Compared to a 16-core CPU baseline, our approach achieves up to 38× speedup, with warp-level sizing further improving performance by an additional 4.77% on average.

Acknowledgment

This project is supported by NSF grants 2235276, 2349144, 2349143, 2349582, and 2349141.

References

1. Cheng, C.K., Holtz, C., Kahng, A.B., Lin, B., Mallappa, U.: Dagsizer: A directed graph convolutional network approach to discrete gate sizing of vlsi graphs. ACM Transactions on Design Automation of Electronic Systems **28**(4), 1–31 (2023)
2. Coudert, O., Haddad, R., Manne, S.: New algorithms for gate sizing: A comparative study. In: Proceedings of the 33rd annual Design Automation Conference. pp. 734–739 (1996)
3. Flach, G., Reimann, T., Posser, G., Johann, M., Reis, R.: Effective method for simultaneous gate sizing and v th assignment using lagrangian relaxation. IEEE transactions on computer-aided design of integrated circuits and systems **33**(4), 546–557 (2014)
4. Hu, S., Ketkar, M., Hu, J.: Gate sizing for cell library-based designs. In: Proceedings of the 44th annual Design Automation Conference. pp. 847–852 (2007)
5. Lin, Y., Grover, V.: Using cuda warp-level primitives (August 2017), https://developer.nvidia.com/blog/using-cuda-warp-level-primitives/, accessed: 2025-01-28
6. Mangiras, D., Chinnery, D., Dimitrakopoulos, G.: Task-based parallel programming for gate sizing. IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems **42**(4), 1309–1322 (2022)
7. Nickolls, J., Buck, I., Garland, M., Skadron, K.: Scalable parallel programming with cuda: Is cuda the parallel programming model that application developers have been waiting for? Queue **6**(2), 40–53 (2008)

8. Sharma, A., Chinnery, D., Bhardwaj, S., Chu, C.: Fast lagrangian relaxation based gate sizing using multi-threading. In: 2015 IEEE/ACM International Conference on Computer-Aided Design (ICCAD). pp. 426–433. IEEE (2015)
9. Shi, B., Zhang, Y., Srivastava, A.: Accelerating gate sizing using graphics processing units. IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems **31**(1), 160–164 (2011)