Massively Parallel GPU Rasterizer for Next-Generation Computational Lithography

Loay Hegazy, Mohamed Taher, Sherif Hammouda
Siemens EDA, CFC A3, New Cairo, Egypt

SIEMENS

Introduction

Rasterization is the process of converting continuous geometric shapes, such as polygons or curves, into a discrete grid of pixels as shown in Figure 1, enabling digital processing, display, and simulation. While widely used in computer graphics and vision, rasterization is equally critical in Electronic Design Automation (EDA), where it underpins high-resolution mask synthesis, lithography simulation, and Optical Proximity Correction (OPC). On GPUs, rasterization can leverage massive parallelism, but challenges arise from load balancing, atomic contention, and the need for high precision when handling extremely fine geometries. In OPC, circuit layouts are repeatedly rasterized onto extremely dense grids—often tens of thousands of pixels per micron—to simulate light propagation and resist behavior with nanometer-scale accuracy. Unlike conventional graphics, this requires rasterization to be not only fast but also pixel-accurate and connectivity-preserving, since even small breaks in thin lines or contacts can lead to incorrect aerial image calculations, degraded OPC convergence, and yield loss. As semiconductor nodes continue to shrink below a few nanometers, the computational demands of OPC-driven rasterization grow dramatically, making the development of GPU-optimized, high-precision algorithms essential for both accuracy and efficiency.

Fig. 1. Example of converting a polygon into a pixel-based representation through rasterization.

Background and Motivation

Traditional Rasterization in Graphics.

- Polygons are rasterized using a binary coverage model, pixel = 1 if fully covered, and 0 if uncovered.
- Works well in computer graphics because small errors are visually imperceptible.

Limitations in Computational Lithography & OPC.

- Binary coverage is insufficient: polygon edges can cut through pixels at any sub-pixel location.
- Accurate fractional pixel coverage is required to model light intensity and resist behavior.

High-Precision Requirements.

- Nanometer-scale accuracy demands floating-point coverage computation.
- Thin features must remain connected to preserve manufacturable mask geometries.

Performance Challenge

- OPC involves billions of polygons per iteration and trillions of pixel evaluations per mask layer.
- GPUs provide parallelism but face issues:
  - Lack of accuracy.
  - Irregular memory access.

Our work tackles these issues by designing a GPU-friendly rasterization algorithm that:

- Operates in floating-point precision to compute accurate partial coverage.
- Preserves pixel connectivity for sub-pixel-width geometry.
- Uses tile-based, warp-cooperative strategies to achieve high throughput without sacrificing accuracy.

Methodology

1. **Initialization:**
   - Start with a grid where each element represents a pixel.
   - Initially, all pixels in the grid are set to a value of zero.
   - Reserve shared memory size using the polygon with the maximum number of vertices among all polygons.
2. **Polygon Assignment:**
   - Assign each polygon to a block of threads in the computing environment. This parallel approach enables simultaneous processing of multiple polygons.
   - Transfer the vertices of the polygon to the shared memory. This step is important for reducing global memory access latency, as it allows threads to access polygon vertices rapidly without repeated memory accesses, effectively accelerating the rasterization process.
3. **Bounding Box Calculation:**
   - Compute a rectangular bounding box as shown in Figure 2 that encompasses the entire polygon.
   - This bounding box simplifies the computational workload by narrowing the focus to a smaller section of the grid, directly enclosing the relevant area for processing.
   - $BB_{min} = (\min_{i=1} x_i, \min_{i=1} y_i)$
   - $BB_{max} = (\max_{i=1} x_i, \max_{i=1} y_i)$

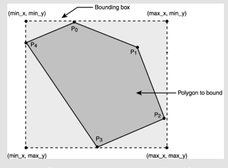

Fig. 2. Example of computing bounding box for a polygon.

4. **Thread-Pixel Allocation:**
   - Assign each computational thread to a unique pixel within the bounding box of the polygon.
   - This pixel-specific assignment allows for precise calculations as each thread individually assesses its designated pixel's relationship to the polygon, enhancing parallel processing capabilities.
5. **Pixel Classification and Processing:**
   - As shown in Figure 3, For each thread, conduct a detailed evaluation to categorize the pixel associated with it as either inside, outside, or directly on the boundary of the polygon.
     - **Outside Pixels**: Pixels that fall outside the polygon boundaries remain unaltered at their initialized value of zero, signifying their exclusion from the polygon-specific data.
     - **Inside Pixels**: If a pixel is determined to lie within the boundaries of the polygon, its value is updated to 1.
     - **Boundary Pixels**: For pixels positioned on the polygon's boundary, the polygon edges are adjusted to align precisely with the pixel grid. The polygon's edge that intersects with the pixel boundary is analyzed to calculate the trapezoidal area formed, providing a nuanced representation of the boundary interaction.
       Atomic Add is used only in boundary cases to handle multiple polygons that may pass through the same pixel.

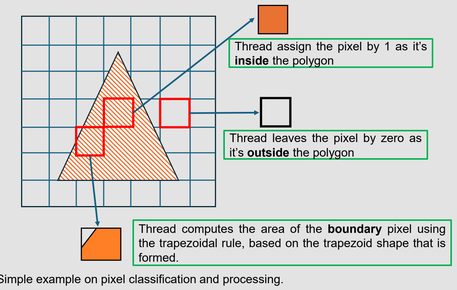

Fig. 3. Simple example on pixel classification and processing.

A fixed-size thread block will be assigned to all polygons. This block scans the polygon pixels in segments in a coalesced memory access until the entire polygon is processed. For example, as illustrated in Figure 4, a 1D thread block of 4 threads first processes the first 4 pixels region, then shifts to the next region, and continues in this manner until it scans the whole boundary box.

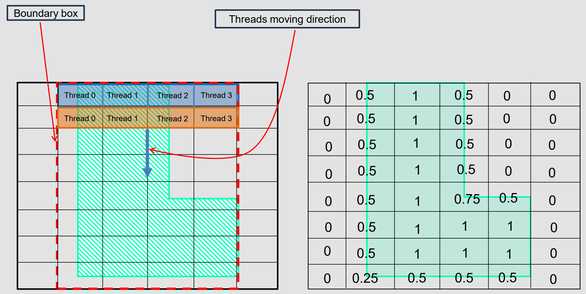

Fig. 4. Example on output grid for simple L-shape being rasterized using block of threads.

Results

Performance Analysis (Figure 5)

The execution times for rasterizing Manhattan and curvilinear polygon datasets across various CPU:GPU configurations reveal significant speed enhancements when utilizing GPUs. All testing was conducted on a **NVIDIA H100** GPU machine, providing a robust platform for benchmarking. The performance is compared against a highly optimized CPU algorithm that surpasses our algorithm when run on a CPU.

Manhattan Shapes:

The dataset consists of **172,585,897** polygons with an average of 8 vertices each. In every configuration tested, GPUs significantly outperformed CPUs. The original 1:1 CPU:GPU setup showed the GPU achieving a remarkable execution time of 2.2 seconds versus the CPU's 639 seconds. Notably, with increased resource allocation:

- In a 16:4 setup, the GPU completed its task in 0.56 seconds compared to the CPU's 48 seconds.
- In a 32:4 configuration, the GPU's execution time was reduced to 0.29 seconds, with the CPU taking 27.5 seconds.

In each increased setup for the CPU, the number of streams on the GPU was also increased, contributing to the improved GPU performance.

Curvilinear Shapes:

For this dataset, consisting of **47,071,509** multigons with an average of 114 vertices, the GPUs consistently displayed superior performance. Initial results had the GPUs running around 29 to 30 seconds, whereas the updated configurations reveal:

- In a 4:4 setup, the GPU completed in 7.5 seconds as opposed to the CPU's 340 seconds.
- For a 16:4 configuration, the GPU executed in 1.8 seconds, while the CPU took 64 seconds.
- In a 32:4 setup, GPU time was 0.9 seconds compared to the CPU's 30 seconds.

Again, the increase in GPU streams in each configuration contributed to the enhanced performance of the GPU.

Overall Speedups:

- GPU-based rasterization achieved speedups of up to approximately **290x** for Manhattan shapes and **45x** for curvilinear shapes.
- These results underscore the GPUs' effectiveness in handling both simple rectilinear and complex geometries, with particularly significant benefits for high-volume, simple-shape workloads.

Accuracy:

- Accuracy testing confirmed that pixel errors did not exceed 1% absolute error against CPU results, ensuring a high-fidelity representation of all features.

This analysis highlights the substantial efficiency benefits of GPU utilization in rasterization tasks, providing both rapid processing and accurate results across different polygon configurations.

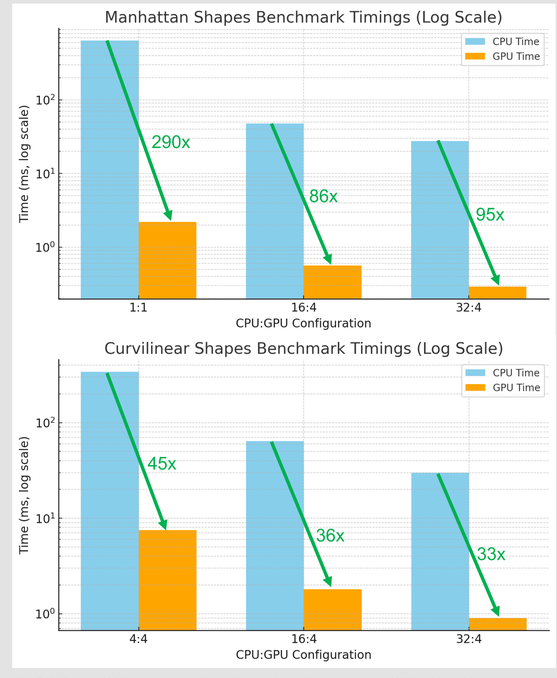

Fig. 5. CPU and GPU runtimes for Manhattan and curvilinear datasets. GPU achieved large speedups with pixel errors under 1% against CPU results.

Conclusion

The study demonstrates the impressive efficiency gains of GPU-based rasterization in handling large-scale, high-precision tasks across diverse polygon shapes. By leveraging parallel processing, the proposed algorithm not only achieves massive speedups—about 45x and 290x compared to traditional CPU methods—but also maintains high accuracy with errors under 1% against CPU results. These findings underscore the GPU's potential to enhance the rasterization processes in Electronic Design Automation (EDA) by significantly reducing computation time while ensuring precise geometric representation. This work paves the way for further exploration into GPU optimization techniques that could enhance the scalability and functionality of rasterization in increasingly complex semiconductor design scenarios.