Processing Integrated Circuit Layouts Using Python: A Case Study On Rapid Prototyping

Srinivasan Jagannathan
Exponent, Inc.
sjagannathan@exponent.com

Nareg Sinenian
Exponent, Inc.
nsinenian@exponent.com

Jared Starman[1]
Tesla Motors, Inc.
jstarman@teslamotors.com

Abstract

This paper narrates our experience developing, in a relatively short time, an application to "rasterize" layers of integrated circuit (IC) layout definitions specified in the Calma Graphic Data System (GDS) II file format. We developed software to parse GDS-II IC layouts to generate a bitmap of regions of the chip that are filled with material such as metal, poly-silicon, etc. Such bitmaps are useful for analysing the geometry of IC design and implementation. We developed the software in the Python programming language, reputed for use in rapid application development environments. Our experience developing and validating the application provides useful insights into the general methodology of iterative development, and the suitability of Python in non-traditional, rapid prototyping environments. Our experience shows that the choice of a versatile programming language can greatly improve productivity in a rapid application development environment where there is incomplete information at the outset.

Keywords: *Python, iterative development, rapid prototyping, IC layouts, GDS-II.*

1. Introduction and Problem Description

Integrated circuits are highly complex devices that incorporate millions of circuit components assembled through an intricate process on a semiconductor substrate. Modern integrated circuit (IC or "chip") design involves the use of electronic design automation tools that assist engineers in designing at the logic level ("hardware logic design"), all the way through specifying the placement of circuit components on the physical chip ("place and route"). Ultimately, placement of transistors, wires, capacitors, etc., on the chip effectively boils down to the manipulation of geometric shapes on the silicon wafer. Starting with a silicon wafer, successive processes operate to add layers of chemicals or metal, which are

[1] The work described in this paper was performed when Dr. Starman was employed at Exponent, Inc.

then etched away using specific patterns ("masks") to leave behind geometric shapes that eventually form the components of the IC. These specific geometric shapes may be specified in a variety of electronic file formats.

One commonly used format is the Calma Graphic Data System (GDS) II—a hierarchical description of the structures that comprise a semiconductor chip. [1] Imagine a semiconductor chip containing a circuit "C" created within a square region with sides 1 micron in length. Further, assume that the same circuit structure is repeated 1000 times on the chip (instances C1, C2, …., C1000) along with two wires running the length of the chip that connect each of these circuits, one wire for power, and the other for ground. This is illustrated in Figure 1. A GDS-II format representation of the chip could define a container structure for the entire chip, and another structure for the circuit C. Additionally, an array comprising 1 row of 1000 C structures could be defined. Finally, the power, ground, and interconnect wires could be defined as rectangles placed at specific locations within the chip structure. The GDS-II format allows for a compact representation of complex structures because, for example, the 1000 instances of C need not be separately defined, but instead defined as an array containing 1000 instances of C. Additionally, GDS-II allows defining geometric manipulations of a structure, such as rotation, magnification, and mirroring, before nested placement (at a specific location) within another structure.

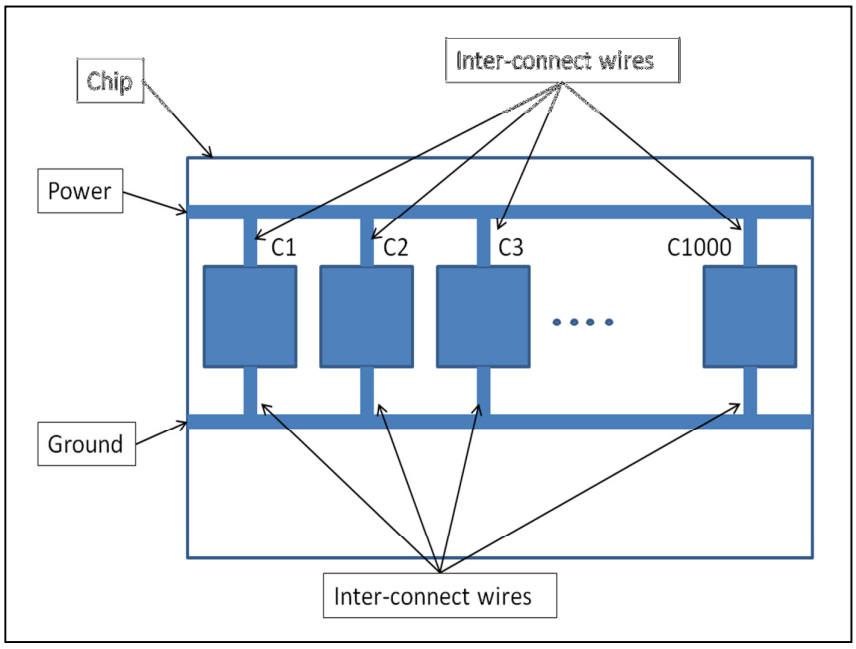


Figure 1. An illustrative representation of a circuit layout on a chip.

A number of computer design and automation tools exist in the electronic design automation (EDA) industry that allow for reading, processing, visualizing, and editing GDS-II files. KLayout [2] is free and open source. Cadence Virtuoso [3] and Synopsys IC Workbench [4] are proprietary software available for purchase. However, none of these tools could be readily customized to perform specific quantitative analysis on the geometric shapes defined

for the chip. For example, our analysis—the details of which are confidential—required performing statistical processing of the filled regions in an IC.

We investigated and eliminated a number of potential solutions to our problem. For example, we investigated building upon the open source KLayout tool, but did not pursue it because of the significant challenges in understanding and adapting a complex body of software source within the time constraints of our project. We also investigated performing a screen capture of a visual rendering of a chip using one of the tools available to us, and then analyzing the captured image. However, given the small scale features (e.g., interconnects) in a modern semiconductor chip and the relatively large size of the die, images of the entire die as viewed on a typical computer screen did not reveal the complex underlying details of the chip that were required for our analysis. For instance, to perform analysis of structures at 100 nanometer resolution in a chip having dimensions of 2 centimeters by 2 centimeters requires 40 billion pixels. For comparison, the average computer display has on the order of a few million pixels.

We therefore decided to develop our own GDS-II processing tool to generate the visualizations in a format we could use for our analysis. Validation of our software was very important. In addition, a significant technical constraint was the amount of memory required for our bitmaps. For instance, an image format that uses 8-bit color depth would require more than 37 gigabytes of storage for our 40 billion pixels. In a relatively short time period, we had to (1) develop a GDS-II parser, (2) efficiently manage storage requirements, and (3) validate the operation of the developed software.

This paper describes how we met these challenges, and the lessons we learned. We used an iterative development methodology whereby we developed the software as kernels of functionality that we could rapidly code and then validate. Each successive kernel built on an operational, validated underlying kernel of functionality. For instance, the first kernel was to successfully read in a GDS-II file. The second kernel was to create in-memory structures for the GDS-II structures and obtain the physical dimensions of the chip. The third kernel of functionality was to develop a rasterizer. The final kernel was to make processing more efficient. This approach presents a continuum of progressive challenges, where each challenge is solvable.

The remainder of this paper is organized as follows. Section 2 explains our choice of Python. Section 3 describes our development and validation of the kernels. Section 4 discusses our solutions for improving efficiency. We present a summary of our conclusions in Section 5.

2. Choosing Python

Python is an open source interpreted programming language that executes on a variety of platforms including Windows, Linux/Unix, and Mac OS X. [5] Python is used in a variety of application areas, and includes packages for processing of GDS-II formats [6] and for performing scientific computations on large arrays [7]. These features combined with our familiarity with Python made it an easy choice as our development platform. Moreover,

Python 2.7 is standard on Mac OS X and many Linux distributions, and easy to install on Windows.

However, the ease of using Python, combined with our iterative development methodology, concealed some unforeseen problems. Because our initial focus was only on being able to read in a GDS-II file, we did not appreciate that handling files larger than 2 gigabytes (e.g., bitmap files) with a 32-bit Python installation was not possible. Once we realized this problem, we had to switch over to a 64-bit implementation. At this stage, we chose Python 3 because it was more current. However, Python 3 (64-bit) was not fully compatible with the source we had developed for Python 2.7 (32-bit). For instance, values read from a GDS-II file as strings in Python 2.7 were read as binary objects in Python 3. Valuable time was spent porting our code to ensure it worked as intended with Python 3.

A significant problem we encountered with Python, though not unexpected, was the consumption of memory. Being an interpreted language, we expected that a Python script would consume more runtime memory than a comparable compiled program. In fact, the memory consumed to read in and process a GDS-II file was nearly 20-times the size of the file. Processing GDS-II files defining millions of geometric shapes therefore meant that our software consumed several gigabytes of RAM.

On the positive side, Python provided numerous benefits that allowed us to rapidly develop our software. Reading in GDS-II files into memory was as easy as importing the python-gdsii library and calling the "load" method. Likewise, Python includes a number of useful features available in many modern programming languages. For example, using associative arrays that allow objects to be indexed by a string, we could very easily access the details of a structure using its previously defined name in the GDS-II file. For instance, for the Figure 1 example, the details of circuit "C" could be accessed using the index "C." Another useful Python feature is support for sorting lists of complex structures by specifying a function to identify the sorting key. For example, in Python 3, to sort a list "L" of tuples (x,y,z) on the y-dimension, one could use the command "L.sorted(key=lambda tup: tup[1])." In this command, a lambda function is defined to return the y-value in a tuple as the sorting key. We found such features to be particularly useful in our development effort.

3. Developing Kernels of Functionality

As discussed above, we developed and validated progressive kernels of functionality, in increasing order of complexity. To begin with, our first kernel was to simply read in a GDS-II file into memory and traverse the structures as read. To validate our code, we relied on simple print statements to confirm we were reading in and traversing GDS-II files as intended. The second kernel of functionality was to create in-memory structures for each structure read from the GDS-II file. At the end of this stage, we wanted to have a set of polygons and a set of lines in memory, corresponding to every single polygon or line of the chip layout definition.

The hierarchical structure defined by GDS II dictates the use of a depth-first traversal algorithm to visit each structure. At each level, the following operators were calculated and passed to the children: (1) translation coordinates, *i.e.*, where the child is placed within the

parent, (2) the rotation with respect to origin of the parent, (3) magnification of the child, and (4) mirroring (whether the child should be flipped about the x-axis). Each of the operators was passed as a list, starting from operators for the root of the tree, and descending down to the current node. At a leaf, all the operations were performed in a stack-order to obtain the final position of the geometric shapes defined at that node. Development of this stage of functionality was particularly complicated because the GDS-II operators and the order of operations (*e.g.*, rotation and mirroring are not commutative) were not fully documented.

The Depth First Explode Algorithm

```
DepthFirstExplode(Node, ParentRotations, ParentScale, ParentMirror,
ParentOrigin)
   For each element in the Node:
      If element is a Polygon or Path:
          Get final vertex positions (apply operations in stack-order).
       If element is a structure S:
            Let R = rotation of S inside Parent
            Let Mag = scaling of S inside Parent
            Let Mir = mirroring of S inside Parent
            Let Org = position for placing S inside Parent
            NewRot = ParentRotations.Append(R)
            NewSc = ParentScale.Append(Mag)
            NewMir = ParentMirror.Append(Mir)
            NewOrg = ParentOrigin.Append(Org)
           DepthFirstExplode(S, NewRot, NewSc, NewMir, NewOrg)
       If element is an array A of structure S:
            Let R = rotation of A inside Parent
            Let Mag = scaling of A inside Parent
            Let Mir = mirroring of A inside Parent
            Let Org = position for placing A inside Parent
            NewRot = ParentRotations.Append(R)
            NewSc = ParentScale.Append(Mag)
            NewMir = ParentMirror.Append(Mir)
            NewOrg = ParentOrigin.Append(Org)
            For each row in A:
               For each column in A:
                  Let Org = location of current cell in A
                  NewRot = ParentRotations.Append(0)
                  NewSc = ParentScale.Append(1)
                  NewMir = ParentMirror.Append(0)
                  NewOrg = ParentOrigin.Append(Org)
                  DepthFirstExplode(S, NewRot, NewSc, NewMir, NewOrg)
```

Figure 2. The Depth First Explode algorithm to calculate polygons and line placements in an Integrated Circuit chip.

Our algorithm, called "DepthFirstExplode" is summarized as pseudocode in Figure 2. After DepthFirstExplode, we were able to programmatically calculate the dimensions of the chip to determine the size of the bitmap. Because the KLayout tool implements these operations, we used it to benchmark the coordinates generated by our software. For instance, we

validated the chip dimensions generated by our software by comparing them to the dimensions we could visually measure using the KLayout tool.

In the third kernel of functionality, we implemented a rasterization algorithm. The purpose of this algorithm was to generate a bitmap of regions in the current layer of the chip that are filled with material. Fundamentally, the algorithm took as input a list of polygons (or a line with end points and a width) represented as a set of successive vertices, and produced as output a filled region corresponding to the polygon (or line) in a matrix of pixels representing the chip. We implemented the well-known scan-line polygon-filling algorithm [8] as shown in Figure 3. Built-in Python sorting functions were particularly helpful for speeding up the development exercise, since the polygon-filling algorithm requires multiple sorting steps across different dimensions.

The Scan-line Polygon Fill Algorithm

```
PolyFill(Vertices)
      Construct a list of non-horizontal edges E from Vertices.
      Sort edges in ascending order of smaller y-value in each edge.

      For each edge e:
            Calculate 1/m (inverse of slope).

      Initialize Active Edge Table (AET) to empty.
      Calculate range (miny, maxy) of y across all vertices.
      For y=miny to maxy:
            For each edge in AET:
                  If largest y-value in edge == y:
                        Remove edge from AET.
                  Else:
                        Update x-value of edge for current y.
            For each edge not in AET:
                  If y >= smallest y-vale in edge:
                        Add edge to AET.

            Sort AET by x-values for current y.

            i=0.
            For each edge in sorted AET:
                  i=i+1.
                  Let x_i = x-value of i^th edge in sorted AET.
                  Let x_i+1 = x-value of (i+1)^th edge in sorted AET.
                  If i is odd:
                        Fill bitmap from (x_i,y) to (x_i+1,y).
```

Figure 3. Scan-line algorithm for filling polygons.

The scan-line algorithm for rasterization can be thought of as firing an imaginary ray in the x-direction and following the path of the tip of the ray. As the tip of the ray pierces a shape or exits a shape, we keep track of whether the ray is inside a polygon. Essentially, the scan-

line algorithm sorts all the edges according to the smaller y-coordinate in the edge and maintains an Active Edge Table of edges that have a point that could intersect the current ray. The ray is fired methodically starting with the smallest y-coordinate of any vertex and ending at the highest y-coordinate of any vertex. As the y-coordinate increases, new edges whose smaller y-coordinate vertex matches the current y-coordinate are added to the Active Edge Table. Conversely, edges whose higher y-coordinate vertex matches the current y-coordinate are removed from the Active Edge Table. Furthermore, for each edge in the Active Edge Table, the algorithm determines the x-coordinate of the point in the edge that has the current y-coordinate of the ray (i.e., these points are in the ray's path). The edges in the Active Edge Table are then sorted in ascending order of the x-coordinates of the points on the edges in the ray's path. The sorting establishes the order in which these edges are encountered by the tip of the ray. When the tip of the ray encounters the first edge in the path it is entering a shape; therefore all points thereafter until the ray exits should be a filled region. When the tip of the ray encounters the second edge in the path, the ray is exiting the shape; therefore all points thereafter until the ray enters another shape should be an unfilled region. In other words, odd edges indicate entry into a shape and even edges indicate exit from a shape.

Working on the order of 100-nanometer scales (and smaller) carries a huge storage penalty. A major decision we had to make was how to efficiently store the rasterized image for further processing. We only needed to discern between "filled" and "unfilled" regions of a chip. So we decided to use one bit per pixel, thereby reducing our storage by a factor of 8, but at the expense of bit arithmetic to fill up individual bits. However, even this improvement only reduced the storage to a few gigabytes from tens of gigabytes.

While our code was initially written to store matrices in memory, this was not possible for chips exceeding the RAM available. Fortunately, Python's NumPy package includes support for memory-mapped matrices that were stored on disk but manipulated in memory. We quickly converted our code to work with memory-mapped matrices. Thus, the versatility of Python, including its numerous support libraries, was a great help for our rapid prototyping effort.

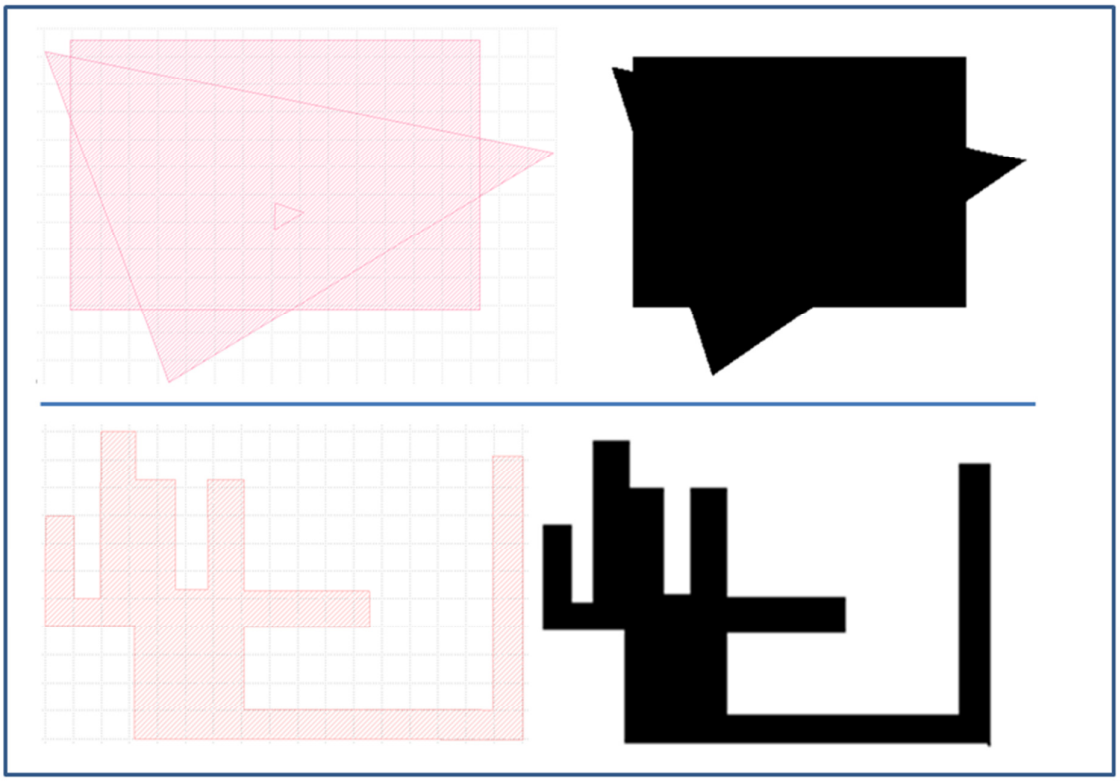


Figure 4. Examples of GDS-II layouts as rendered in the KLayout tool, and the corresponding PNG image generated by our rasterization software.

In order to validate the rasterization algorithm output, we developed additional code to help us visually confirm the algorithm operation. Specifically, we developed code to convert the bitmaps to a PNG image format, in order to visually inspect the bitmaps generated for test GDS-II files. We chose PNG because a Python package named PyPNG allows for easy conversion.[2] However, getting the PyPNG package, originally developed for a Python 2 implementation, to work with a Python 3 implementation was not straightforward and took some research and effort. But since validation of the software was a very important goal, we believed it was a worthwhile effort. In fact, we found and debugged implementation errors that manifested as minor defect patterns in PNG images. Of course, we only converted small test bitmaps to PNG images, and the PNG conversion was turned off for production use of our code. Example layouts, as displayed in KLayout, and the corresponding PNG images of our generated bitmaps are shown in Figure 4.

We now had a working program that could generate bitmaps of filled regions in specific layers of GDS-II files. As a working prototype, this software could get the job done, but it

[2] Another package that may be used instead of PyPNG is the PIL package. See https://pythonhosted.org/pypng/ca.html

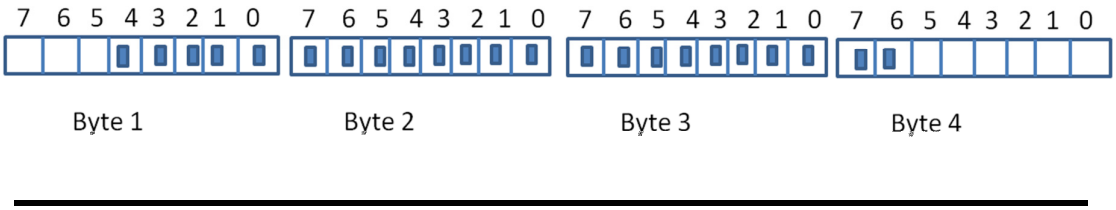
was quite slow. Our final kernel was to focus on improving the execution time of the software, as described in the next section.

4. Improving Efficiency of the Software

During the development of the early kernels, the focus was on obtaining a functional prototype that operated correctly. Once functional software was developed, the focus shifted to improving its efficiency. Given the gigabytes-size of the bitmap matrix, and the several millions of polygons, efficiency was very important. To put this in perspective, the bitmap generation process can take several hours to run on a professional desktop (2.3 GHz processor, 8GB RAM). Even a moderate improvement can result in significant time savings. From a storage perspective, any efficiency achieved in reducing in-memory data structures will have an impact on whether or not the computer will swap pages in and out of memory ("thrash") when processing a particularly complex chip.

One of the ways we reduced the memory footprint was to reduce the in-memory information. Initially, we kept track of placement hierarchy for each polygon, i.e., through which sequence of hierarchical placements a particular polygon came to be placed at a particular location in the chip. While this provided useful information, there was no room in memory for such information, so it was discarded. Other memory related optimizations we performed include performing in-place sorting of lists so as to avoid creating copies of the lists, and pre-allocating lists so as to avoid run-time reallocation and fragmentation of lists.

Next, we optimized execution time by modifying how the bitmap matrix was filled. Initially, we wrote the filling algorithm as a simple for-loop that filled a row of pixels, one pixel at a time. We had defined the bitmap as a matrix of unsigned bytes, and filling each pixel meant setting each individual bit in a byte. However, if the number of pixels to fill spanned multiple bytes, instead of setting individual bits in a byte at a time, a value for the entire byte can be specified, thereby setting all the bits in that byte in a single operation. Consider a series of 23 pixels to be filled, spanning 4 bytes. This is illustrated in Figure 5. By assigning a value (0x00 or 0xFF) to each of the intervening 2 bytes, there is a substantial reduction in the computational steps. Moreover, additional savings can be achieved by operating on words (32 bit numbers) instead of bytes. Additionally, the bits in the first and last words can also be assigned without use of any loops, as illustrated in Figure 5. The algorithm in Figure 5 assigns bit positions x1 to x2 in a word to 0. Position 0 corresponds to the most significant bit. It is assumed that word-size > x2 > x1>=0. Using this approach, we were able to reduce the processing time by several hours per chip.



Algorithm To Assign Pixels Within A Word To Zero Without A Loop

```
FillWord(row, column, x1, x2)
      bit1=wordsize-1 - x1 % wordsize #bit position of x1 in word
      run = (1<<bit1+1) - 1 #create 1s from bit1 to end of word
      bit2 = wordsize-1 - x2 % wordsize #position of x2 in word
      run2 = (1<<bit2)-1 #create 1s from bit2+1 to end of word
      run = run-run2    #create 1s from bit1 to bit2, inclusive
      mask = ~(run) #create 0s from bit1 to bit2, inclusive
      bitmap_matrix[row][column]= bitmap_matrix[row][column] & mask
```

Figure 5. An illustration to show how pixel filling can be optimized using block assignment of intervening bytes, and a loop-free algorithm to assign bits in the first and last bytes.

While further improvements can still be achieved (*e.g.*, using multiple threads and/or processors), our experience validates our approach to place emphasis on developing functional software first, and optimizations later.

However, the importance of the optimizations cannot be understated. Often, in a rapid application prototyping environment, the emphasis on functional software at the cost of efficiency results in low productivity and excessive computational overhead. Scalability is also a concern in many applications. A balanced approach that also focuses on efficiency can result in enormous savings in time, energy, and effort.

5. Summary of Conclusions

In this paper, we presented our experience developing complex software in a relatively short time using readily available tools. We iteratively developed kernels of functionality and validated each kernel before developing the next kernel. Our approach was useful because at any given time we had some portion of the software functional and validated. On the other hand, because we focused at first on a series of small achievable goals, we missed out on significant issues that we uncovered in the later stages of the development effort. Thus, our experience shows a trade-off in how the goals are designed. A purely short-term focus can negatively impact the overall objective.

Python's support for scientific calculations and readily available packages for various applications make it a very useful platform for development. Our experience confirms

Python is a versatile platform for rapid prototyping. However, Python is memory intensive and not readily portable across different versions. Overall, our recommendation is favorable, with a caveat to understand the overall goals of the project, and the limitations of Python, before embarking on development.

References

1. Steven M. Rubin, "Computer Aids for VLSI Design, Appendix C: GDS II Format," 1994. http://www.rulabinsky.com/cavd/text/chapc.html.
2. KLayout – High Performance Layout Viewer and Editor. http://www.klayout.de/index.html.
3. Cadence Virtuoso Layout Suite. http://www.cadence.com/products/cic/layout_suite/pages/default.aspx.
4. Synopsys IC Workbench. http://www.synopsys.com/Tools/Manufacturing/MaskSynthesis/Pages/ICWorkBench-EditViewPlus.aspx.
5. Python Programming Language. http://www.python.org/
6. python-gdsii. https://www.gitorious.org/python-gdsii.
7. NumPy. http://www.numpy.org/.
8. "Filling Polygons," CS 442/542 Computer Graphics Lecture Notes. http://ezekiel.vancouver.wsu.edu/~cs442/lectures/raster/polyfill/poly.pdf. Washingston State University, December 13, 2013.