GDSFactory: Build better hardware with better software.

JOAQUIN MATRES[1,*], SIMON BILODEAU[2], NIKO SAVOLA[1,3,†], TROY TAMAS[1], SEBASTIAN GOELDI[4], HELGE GEHRING[1], A N MCCAUGHAN[5], STANLEY CHEUNG[6]

[1]GDSFactory
[2]Google Development LLC, Mountain View, CA 94043, USA
[3]Department of Applied Physics, Aalto University, PO Box 13500, FIN-00076 Aalto, Finland
[4]PsiQuantum, 700 Hansen Way, Palo Alto, CA 94304, USA
[5] National Institute of Standards and Technology, Boulder, CO 80305, United States of America
[6]Department of Electrical and Computer Engineering, North Carolina State University, Raleigh, NC 27606, USA
[†]Part of the work done while the author was a student researcher at Google Quantum AI.
[*]*jmatres@gdsfactory.com*

Abstract

For efficient design, verification and validation of integrated circuits and components it is important to have an easy to customize and extend workflow. Python has become the industry standard programming language for machine learning, scientific computing and engineering.

GDSFactory is a Python library to build chips (Photonics, Analog, Quantum, MEMs, …) that provides you a common syntax for design, simulation (Ansys Lumerical, tidy3d, MEEP, MPB, DEVSIM, SAX, Elmer, Palace, …), verification (Klayout DRC, LVS, netlist extraction, connectivity checks, fabrication models) and validation.

The paper describes the capabilities of GDSFactory, showcasing its end-to-end workflow for layout, simulation, verification, and validation, enabling users to turn their chip designs into validated products.

Introduction

Hardware iterations typically require months of time and involve substantial financial investments, often in the millions of dollars. In contrast, software iterations can be completed within hours and at a significantly lower cost, typically in the hundreds of dollars. Recognizing this discrepancy, GDSFactory aims to bridge the gap by leveraging the advantages of software workflows in the context of hardware chip development.

To achieve this, GDSFactory offers a comprehensive solution through a unified Python API. This API enables users to drive various aspects of the chip development process, including layout

design, verification (such as optimization, simulation, and design rule checking), and validation (through the implementation of test protocols). By consolidating these functionalities into a single interface, GDSFactory streamlines the workflow and enhances the efficiency of hardware chip development.

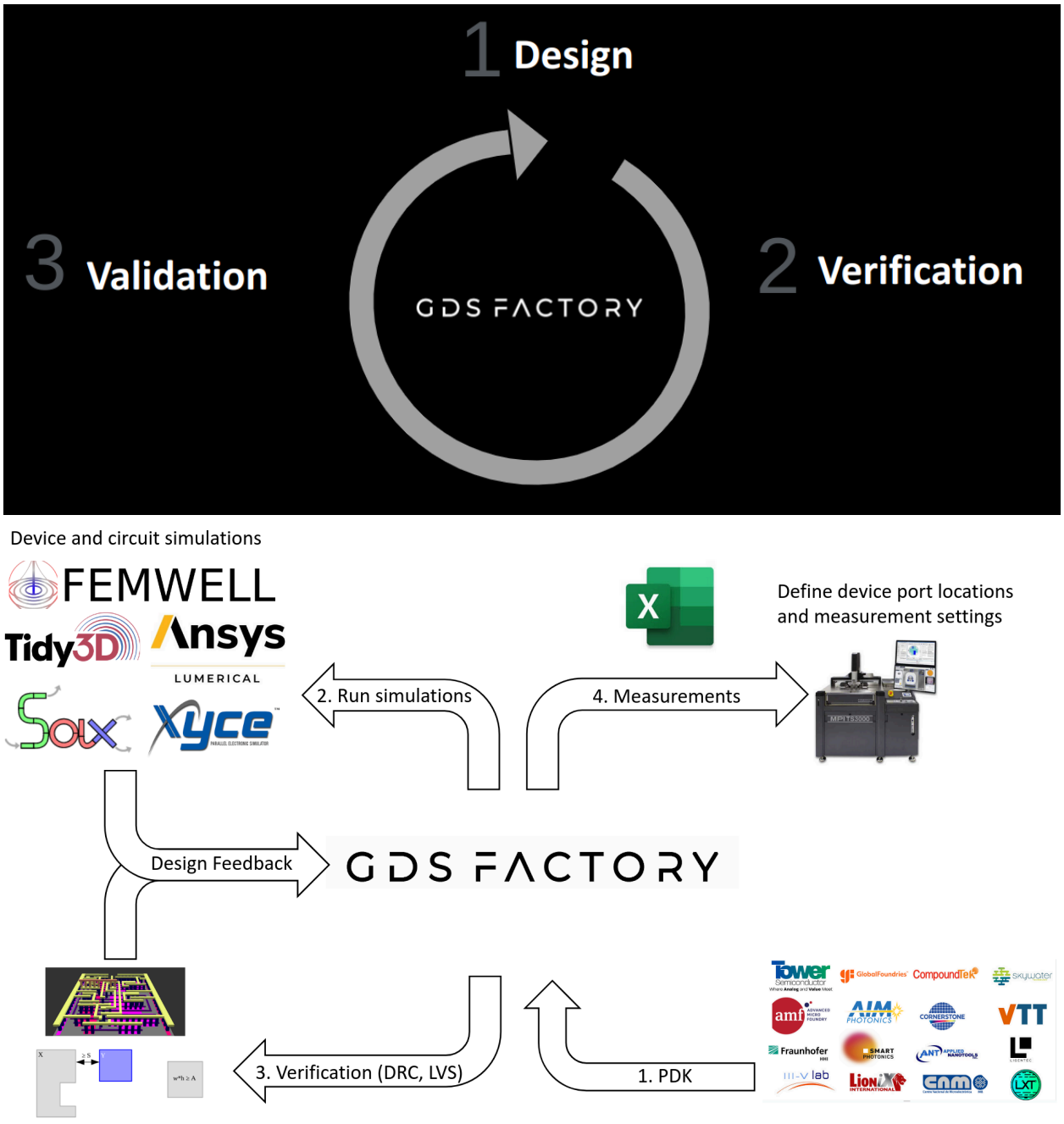

Figure 1: High-level overview of design, verification and validation process.

With GDSFactory you can:

- Design (Layout, Simulation, Optimization)
    - Capture design intent in a schematic and automatically generate a layout.
    - Define parametric cells (PCells) functions in python or YAML. Define routes between component ports.
    - Test component settings, ports and geometry to avoid unwanted regressions.
- Verify (DRC, DFM, LVS)
    - Run simulations directly from the layout thanks to the simulation interfaces. No need to draw the geometry more than once.
        - Run Component simulations (EM solver, FDTD, EME, TCAD, thermal …)
        - Run Circuit simulations from the Component netlist (Sparameters, Spice)
        - Build Component models and study Design For Manufacturing.
    - Create DRC rule decks.
    - Make sure complex layouts match their design intent (Layout Versus Schematic).
- Validate
    - Make sure that as you define the layout you define the test sequence, so when the chips come back you already know how to test them.
    - Model extraction: extract the important parameters for each component.
    - Build a data pipeline from raw data, to structured data and dashboards for monitoring your chip performance.

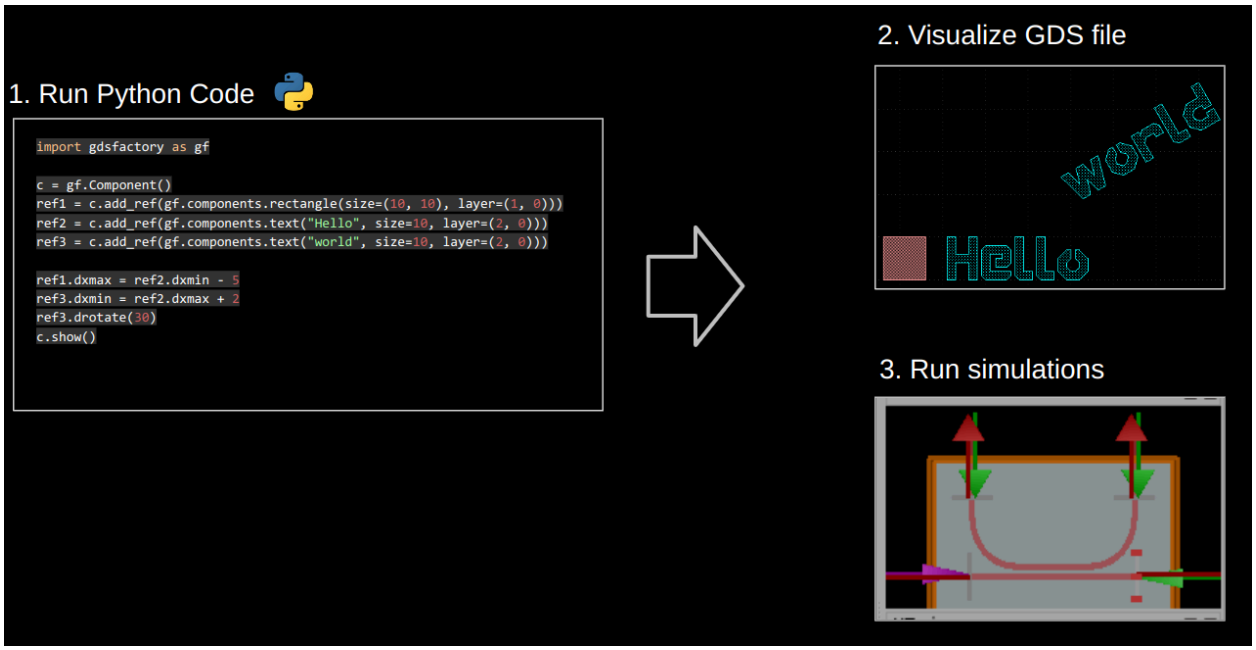

Figure 2: You can drive your layout, design and simulations from a python or YAML code.

Design flow

GDSFactory offers a range of design capabilities, including layout description, optimization, and simulation. It allows users to define parametric cells (PCells) in Python, facilitating the creation of complex components. The library supports the simulation of components using different solvers, such as finite element mesh, mode solver (finite element, finite difference time domain, finite different Bloch mode solver), TCAD and thermal simulators, and FDTD simulations. Optimization capabilities are also available through an integration with Ray Tune, enabling efficient parameter tuning for improved performance.

As input, GDSFactory supports 3 different workflows that can be also mixed and matched.

1. Write python code. Recommended for developing cell libraries.
2. Write YAML code to describe your circuit. Recommended for circuit design. Notice that the YAML includes schematic information (instances and routes) as well as Layout information (placements).
3. Define schematic, export SPICE netlist, convert SPICE to YAML and tweak YAML.

As output you write a GDSII or OASIS file that you can send to your foundry for fabrication. It also exports component settings (for measurement and data analysis) and netlists (for circuit simulations). The following examples concentrate on photonic integrated design, however they are readily adaptable for RF and analog circuit design.

Parametric PCells in Python or YAML

A PCell is a Parametric Cell describing the geometry of a particular device. PCells can accept other PCells as arguments in order to build arbitrarily complex Components.

```python
import gdsfactory as gf

@gf.cell
def mzi_with_bend(radius:float=10)->gf.Component:
      c = gf.Component()
      mzi = c.add_ref(gf.components.mzi())
      bend = c.add_ref(gf.components.bend_euler(radius=radius))
      bend.connect('o1', mzi['o2'])
      c.add_port('o1', port=mzi['o1'])
      c.add_port('o2', port=bend['o2'])
      return c


c = mzi_with_bend(radius=100)
c.show()
```

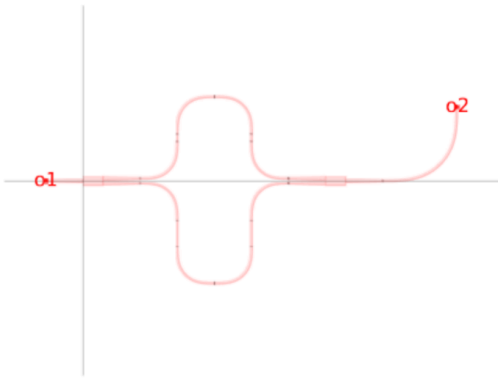

Figure 3: Example of PCell in python.

You can also describe automated single or bundles or routes between different components.

```python
@gf.cell
def nxn_to_nxn() -> gf.Component:
      c = gf.Component()
      c1 = c.add_ref(gf.components.nxn(east=3, ysize=20))
      c2 = c.add_ref(gf.components.nxn(west=3))
      c2.move((40, 10))
      routes = gf.routing.get_bundle(
        c1.get_ports_list(orientation=0),
        c2.get_ports_list(orientation=180),
        with_sbend=True,
        enforce_port_ordering=False,
      )
      for route in routes:
         c.add(route.references)
      return c


c = nxn_to_nxn()
c.show()
```

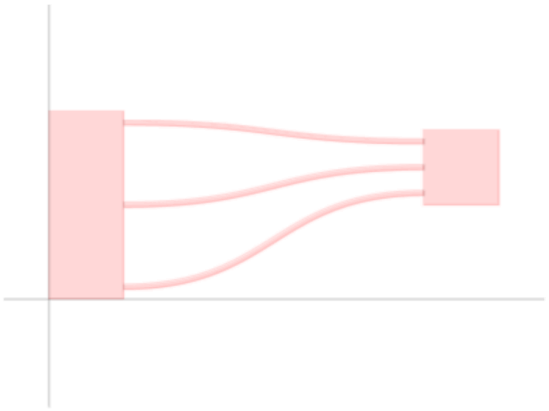

Figure 4: Example of PCell in python with automated routing.

The python and YAML syntax are equivalent. You can also write a PCell as below.

```yaml
name: nxn_to_nxn
instances:
  c1:
       component: nxn
       settings:
       east: 3
       ysize: 20
  c2:
       component: nxn
       settings:
       west: 3
placements:
  c2:
       x: 40
       y: 10
routes:
  optical:
    routing_strategy: get_bundle
    settings:
      with_sbend: True
    links:
      c1,o4: c2,o1
      c1,o3: c2,o2
      c1,o2: c2,o
```

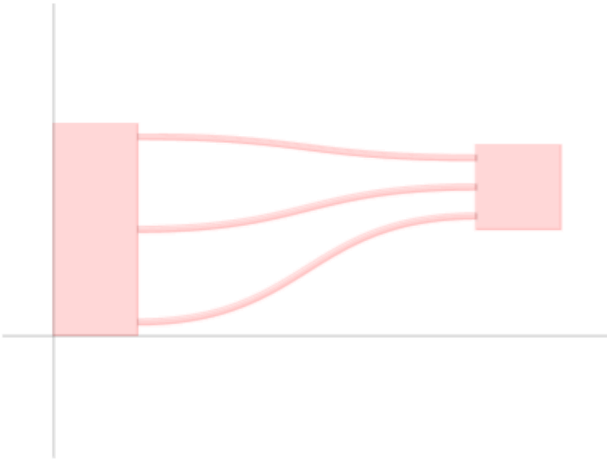

Figure 5: Example of PCell in YAML with automated routing. This YAML is equivalent to the figure below.

A PCell is a Parametric Cell describing the geometry of a particular device. PCells can accept other PCells as arguments in order to build arbitrarily complex Components. GDSFactory supports PCells in python and YAML

Define what is YAML.
One of the advantages of YAML is that is more concise and multiple people write it in the same way

Schematic Driven layout

For complex circuits you can start with a Schematic view that you can convert to YAML.

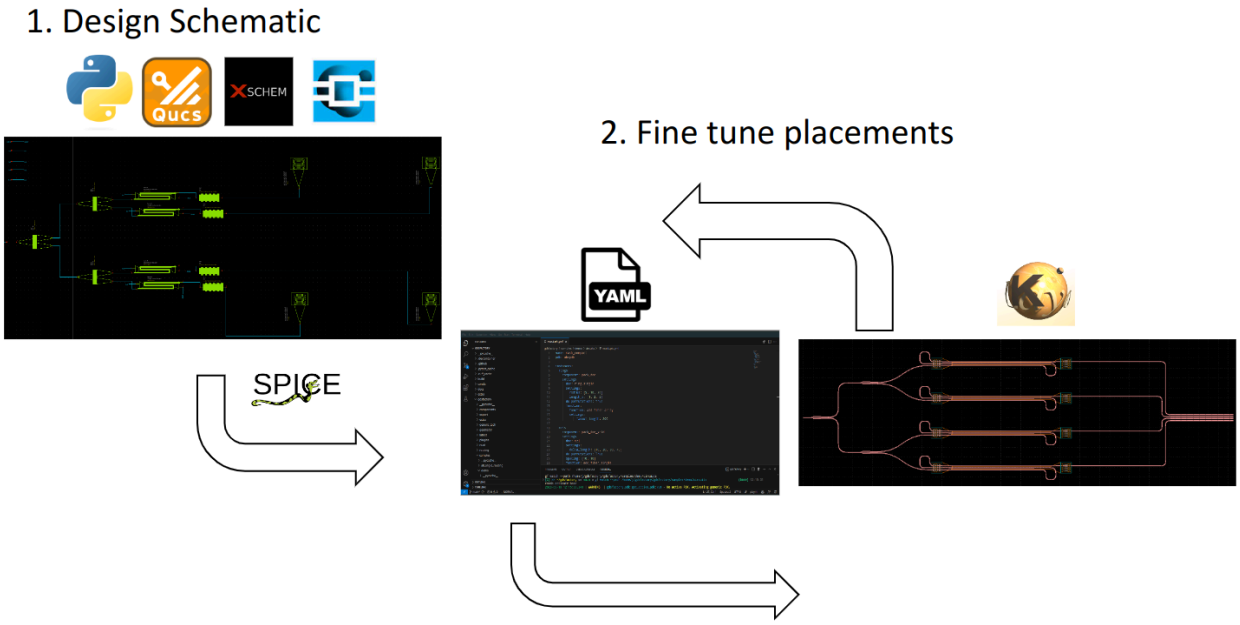

Figure 6: Schematic driven layout. You can export the SPICE netlist into YAML with approximate placements of each component. Then fine tune the placements until you are happy with the layout. The YAML format works as an intermediate step between layout and schematic.

Simulations directly from Layout

GDSFactory python API enables linking together different solvers so that you don't have to draw the geometry twice. Solvers work both at the device, circuit and system level.

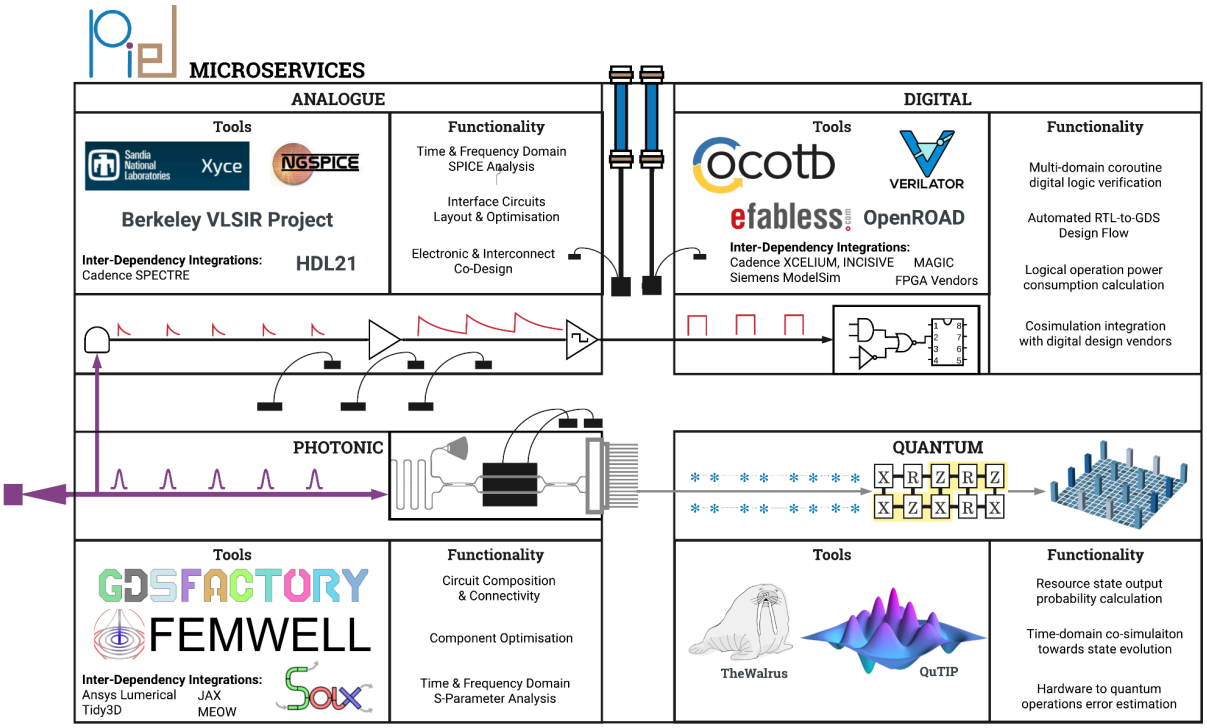

Figure 9: GDSFactory microservices architecture allows you to run different types of simulations with clearly defined inputs and outputs with the same python library.

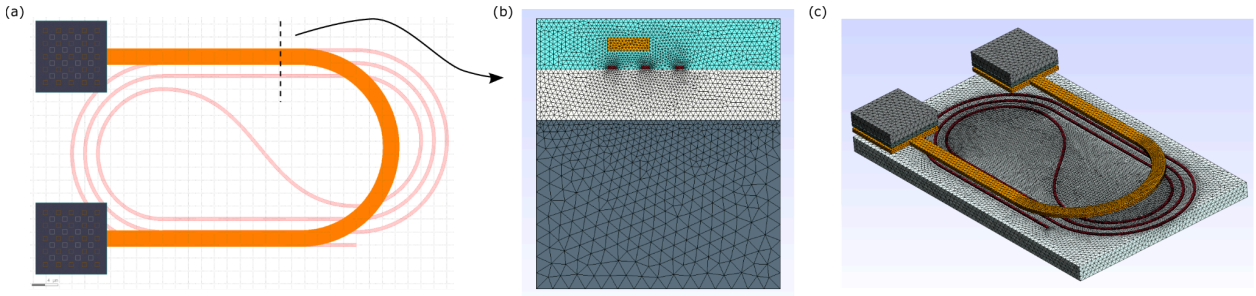

Figure 10: From layout to mesh to simulation without having to replicate your layout in your simulation. Here we show the GDSFactory meshing plugin. (a) Dummy heater layout, using the generic layer stack. (b) Cross-sectional mesh (including cladding, bottom oxide, and substrate), at the location indicated in the layout. (c) Full 3D mesh (cladding and substrate not shown)

Conclusion

The paper has highlighted the key features and functionalities of GDSFactory for hardware design. By leveraging the power of Python, GDSFactory empowers designers with a familiar and flexible programming language widely used in machine learning, scientific computing, and

engineering. This enables seamless integration with existing software ecosystems and promotes code reuse and collaboration.

The verification and validation capabilities of GDSFactory play a crucial role in ensuring the manufacturability and functionality of the designed chips. From functional verification to design for robustness and manufacturing, GDSFactory offers tools and methods to design chips, detect potential issues, and optimize performance.

Furthermore, GDSFactory provides an interactive schematic capture feature that enhances the design process and facilitates the alignment between design intent and the produced layout. With support for SPICE and YAML, designers have the flexibility to define and modify schematics in a user-friendly manner, either visually or programmatically.

The ability to define test sequences and measurement parameters within GDSFactory streamlines the post-fabrication testing process. By establishing a clear measurement and data analysis pipeline, designers can evaluate the fabricated components against the design specifications, ensuring the delivery of known good dies. In conclusion, GDSFactory is a comprehensive and extensible design automation tool that empowers designers to efficiently develop integrated circuits and components. Its Python-driven workflow, combined with its integration capabilities, verification tools, schematic capture feature, and test sequence definition, provides a powerful platform for turning chip designs into validated products.

Acknowledgements

We would like to acknowledge all of the contributors to the GDSFactory project who at the time of writing are: Joaquin Matres Abril (Google), Simon Bilodeau (Google), Niko Savola (Google, Aalto University), Marc de Cea Falco (Google), Helge Gehring (Google), Yannick Augenstein (FlexCompute), Troy Tamas (GDSFactory), Ryan Camacho (BYU), Sequoia Ploeg (BYU), Prof. Lukas Chrostowski (UBC), Erman Timurdogan (Lumentum), Sebastian Goeldi (PsiQuantum), Damien Bonneau (PsiQuantum), Floris Laporte (GDSFactory), Alec Hammond (Meta), Thomas Dorch (HyperLight), Jan-David Fischbach (KIT), Alex Sludds (Lightmatter), Igal Bayn (Quantum Transistors), Skandan Chandrasekar (UWaterloo), Tim Ansell, Ardavan Oskoii (Meta), Bradley Snyder (Superlight) and Amro Tork (Mabrains). Some of the GDSFactory authors have also contributed to other pictures in the paper, Erman Timurdogan contributed the GDSFactory design, verification and validation cycles figure and Dario Quintero contributed the GDSFactory microservices architecture figure.

We would also like to acknowledge the foundational work of other open-source developers: Adam McCaughan (PHIDL), Juan Sanchez (DEVSIM), Matthias Köfferlein (Klayout), Ardavan Oskooi, Alec Hammond, Stephen Johnson (MEEP), Jesse Lu (FDTDZ), Dan Fritchman (VLSIR), Dario Quintero (PIEL), Floris Laporte (SAX, MEOW).