Design and analysis of high-throughput defect inspection algorithms for the semiconductor industry

Name: Marija Trajanoska
Student ID: s2535483
Date: 14/7/2021
Specialisation: Data Science
1st supervisor: Dr. Hao Wang
2nd supervisor: Dr. Laura Dinu Gurtler (ASML)

Master's Thesis in Computer Science
Leiden Institute of Advanced Computer Science (LIACS)
Leiden University
Niels Bohrweg 1
2333 CA Leiden
The Netherlands
Acknowledgement

This project is part of a graduation internship at ASML Delft performed in the time period January to July 2021. It is the result of a fruitful collaboration with my supervisors and colleagues, to whom I would like to express my gratitude. Thanks to Hao for providing the needed academic foundation to an industry context. Thanks to Vincent for being my sounding board day in, and day out, and for challenging me to be deeply involved in this research project. Thanks to Laura for her enthusiastic support, especially towards the end. This work is, in many ways, also an outcome of inspirational discussions with experts within ASML, including Marco Wieland, Maxim Pisarenco, Thomas Huisman, Ton Kiers, Tim Houben and Patrick Tinnemans. I wish to also acknowledge Leiden University faculty and ASML staff for their support and knowledge-sharing. Last but not least, warm thanks to my friends and family for their love and encouragement.
Abstract

Wafer inspection is an integral part of the wafer fabrication process. As electron scanning throughput grows, more stress is placed on the downstream defect detection algorithms. This work focuses on reducing the monetary cost of these algorithms through targeting both computational cost and hardware implementation. We first develop a holistic methodology for quantifying the cost of defect inspection algorithms, and then apply this methodology to a simple defect inspection approach. We also investigate a learning-based alternative for defect inspection which uses simple convolutional neural networks (CNNs). We quantify its performance in terms of the previously proposed methodology, and locate a significant problem with deploying CNNs to the representative class balance with respect to the extreme infrequency of defects.

Keywords: defect, inspection, complexity, convolutional neural networks, lithography
<table>
<thead>
<tr>
<th></th>
<th>Table Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Variable overview</td>
<td>22</td>
</tr>
<tr>
<td>2</td>
<td>Example of complexity breakdown per step (filtering)</td>
<td>23</td>
</tr>
<tr>
<td>3</td>
<td>Comparison between bottom up variants: complexity and run time</td>
<td>25</td>
</tr>
<tr>
<td>4</td>
<td>Comparison between bottom up and existing systems</td>
<td>28</td>
</tr>
<tr>
<td>5</td>
<td>Dataset overview: pixel size, input size, dataset size</td>
<td>32</td>
</tr>
<tr>
<td>6</td>
<td>Neural network numerical complexity overview per layer</td>
<td>34</td>
</tr>
<tr>
<td>7</td>
<td>CNN results for 1d architecture</td>
<td>39</td>
</tr>
<tr>
<td>8</td>
<td>CNN results for 2d architecture</td>
<td>40</td>
</tr>
<tr>
<td>9</td>
<td>Complexity breakdown: offsetting without interpolation</td>
<td>52</td>
</tr>
<tr>
<td>10</td>
<td>Complexity breakdown: offsetting with interpolation</td>
<td>52</td>
</tr>
<tr>
<td>11</td>
<td>Complexity breakdown: difference computation</td>
<td>52</td>
</tr>
</tbody>
</table>
# List of Figures

<table>
<thead>
<tr>
<th>Figure</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Semiconductor device manufacturing process</td>
<td>7</td>
</tr>
<tr>
<td>2</td>
<td>Raster scan pattern</td>
<td>8</td>
</tr>
<tr>
<td>3</td>
<td>escan 1000</td>
<td>8</td>
</tr>
<tr>
<td>4</td>
<td>Overview of different types of defects</td>
<td>10</td>
</tr>
<tr>
<td>5</td>
<td>Comparing ROC curves</td>
<td>11</td>
</tr>
<tr>
<td>6</td>
<td>Dataset overview by size</td>
<td>14</td>
</tr>
<tr>
<td>7</td>
<td>Missing contact hole defect</td>
<td>14</td>
</tr>
<tr>
<td>8</td>
<td>Dataset overview</td>
<td>15</td>
</tr>
<tr>
<td>9</td>
<td>Defect inspection principle</td>
<td>16</td>
</tr>
<tr>
<td>10</td>
<td>Signal distribution of defect candidates</td>
<td>17</td>
</tr>
<tr>
<td>11</td>
<td>Inspection modes overview</td>
<td>17</td>
</tr>
<tr>
<td>12</td>
<td>Bottom up performance across pixel sizes</td>
<td>19</td>
</tr>
<tr>
<td>13</td>
<td>Bottom up performance across modes</td>
<td>19</td>
</tr>
<tr>
<td>14</td>
<td>Random mode without interpolation: complexity and run time</td>
<td>23</td>
</tr>
<tr>
<td>15</td>
<td>Random mode with interpolation: complexity and run time</td>
<td>24</td>
</tr>
<tr>
<td>16</td>
<td>Array mode variants: complexity and run time</td>
<td>24</td>
</tr>
<tr>
<td>17</td>
<td>Bit depth analysis</td>
<td>26</td>
</tr>
<tr>
<td>18</td>
<td>Flow chart for 8-bit implementation</td>
<td>26</td>
</tr>
<tr>
<td>19</td>
<td>Flowchart for the filtering stage</td>
<td>26</td>
</tr>
<tr>
<td>20</td>
<td>Defect classification process – overview</td>
<td>31</td>
</tr>
<tr>
<td>21</td>
<td>Architectures: 1d vs 2d</td>
<td>33</td>
</tr>
<tr>
<td>22</td>
<td>Different activation functions</td>
<td>35</td>
</tr>
<tr>
<td>23</td>
<td>ROC curves for different activation functions</td>
<td>36</td>
</tr>
<tr>
<td>24</td>
<td>ROC curves for different kernel sizes</td>
<td>36</td>
</tr>
<tr>
<td>25</td>
<td>ROC curve for different number of trainable parameters</td>
<td>37</td>
</tr>
<tr>
<td>26</td>
<td>ROC curves for different image sizes</td>
<td>37</td>
</tr>
<tr>
<td>27</td>
<td>Network architecture</td>
<td>38</td>
</tr>
<tr>
<td>28</td>
<td>Example learning curves</td>
<td>39</td>
</tr>
<tr>
<td>29</td>
<td>Output probabilities per pixel size (1d)</td>
<td>40</td>
</tr>
<tr>
<td>30</td>
<td>Output probabilities per pixel size (2d)</td>
<td>41</td>
</tr>
<tr>
<td>31</td>
<td>ROC across pixel sizes and architectures</td>
<td>42</td>
</tr>
<tr>
<td>32</td>
<td>Varying classification threshold across pixel sizes and architectures</td>
<td>42</td>
</tr>
<tr>
<td>33</td>
<td>ROC curves for all architectures (1d, 2d, 3d) across pixel sizes</td>
<td>43</td>
</tr>
<tr>
<td>34</td>
<td>CNN evaluation time overview</td>
<td>44</td>
</tr>
<tr>
<td>35</td>
<td>Representative class balance results</td>
<td>44</td>
</tr>
<tr>
<td>36</td>
<td>Align step – 8-bit flowchart</td>
<td>53</td>
</tr>
<tr>
<td>37</td>
<td>Difference step – 8-bit flowchart</td>
<td>53</td>
</tr>
<tr>
<td>38</td>
<td>Subtraction step – 8-bit flowchart</td>
<td>53</td>
</tr>
<tr>
<td>39</td>
<td>Train and validation graphs for 8 nm pixels (1d)</td>
<td>54</td>
</tr>
<tr>
<td>40</td>
<td>Train and validation graphs for 10 nm pixels (1d)</td>
<td>54</td>
</tr>
<tr>
<td>41</td>
<td>Train and validation graphs for 12 nm pixels (1d)</td>
<td>55</td>
</tr>
<tr>
<td>42</td>
<td>Train and validation graphs for 14 nm pixels (1d)</td>
<td>55</td>
</tr>
<tr>
<td>43</td>
<td>Train and validation graphs for 8 nm pixels (2d)</td>
<td>56</td>
</tr>
<tr>
<td>44</td>
<td>Train and validation graphs for 10 nm pixels (2d)</td>
<td>56</td>
</tr>
<tr>
<td>45</td>
<td>Train and validation graphs for 12 nm pixels (2d)</td>
<td>57</td>
</tr>
<tr>
<td>46</td>
<td>Train and validation graphs for 14 nm pixels (2d)</td>
<td>57</td>
</tr>
<tr>
<td>47</td>
<td>Train and validation graphs for 8 nm pixels (3d)</td>
<td>58</td>
</tr>
<tr>
<td>48</td>
<td>Train and validation graphs for 10 nm pixels (3d)</td>
<td>58</td>
</tr>
<tr>
<td>49</td>
<td>Train and validation graphs for 12 nm pixels (3d)</td>
<td>59</td>
</tr>
<tr>
<td>50</td>
<td>Train and validation graphs for 14 nm pixels (3d)</td>
<td>59</td>
</tr>
</tbody>
</table>
1 Introduction

Almost all of today’s electronic products contain integrated circuits (IC). An integrated circuit, commonly referred to as a chip, is a set of electronic circuits on a small flat piece of semiconductor material, usually silicon, which can perform a variety of crucial functions for various modern devices (Turley 2002). The central processing unit (CPU) is one example of an integrated circuit.

The patterns of integrated circuits are created on semiconductor wafers by a technology called lithography (Levinson 2019). In semiconductor manufacturing, a single wafer contains multiple instances of the same chip. The small block of semiconductor material on which one instance of the integrated circuit is fabricated is called a die. The wafer is then cut (diced) into many pieces, each containing one copy of the circuit. The relationship between the wafer and the die is shown in Figure 1.

![Wafer Diagram](image)

Figure 1: Semiconductor device manufacturing process. Multiple chip instances fabricated on one wafer, which is then diced and packaged individually.

The intricate details of the lithographic process are out of the scope of this work. We instead focus on one of its final stages: the inspection phase.

As a physical process, lithographic printing is not immune to errors and thus requires an elaborate inspection phase to ensure quality. Consequently, wafer inspection, the practice of finding defects on a wafer, is an integral part of the wafer fabrication process. Over the last years, continued improvement of lithography technology has enabled the semiconductor industry to print ever finer features, which in turn posits the necessity for identifying ever finer defects.

Electron beam (e-beam) and optical inspection are the two main technologies used to locate defects on a wafer. Optical beam technology has been the leading inspector of semiconductor wafer production since the birth of ICs. However, due to inherent physical limitations, the optical method cannot detect defects below 10 nanometers (nm). One nanometer is equal to $10^{-9}$, or one billionth, of a meter. For illustrative comparison, a human hair is between 50,000 and 100,000 nm thick. E-beam inspection has started to rule inspection at 10 nm and below.

E-beam inspection is conducted using a type of electron microscope called scanning electron microscope (SEM). The SEM produces images by scanning the surface of a sample with a focused beam of electrons. The electrons interact with the sample, producing low energy electrons, specific to surface topography and composition of the sample. The electron beam is scanned in a raster scan pattern, as shown in Figure 2. The position of the beam is combined with the intensity of the detected signal to produce an image.
The main attributes of e-beam technology are:

1. it is capable of very high resolutions;
2. it is flexible and applicable to a variety of materials and patterns;
3. it is slow, being orders of magnitude slower than optical lithography;¹ and
4. it is expensive and complicated, costing millions of dollars and requiring frequent service to maintain properly.

The challenge is thus to develop a tool which has the capability to inspect a wide area at high speed with high accuracy. There is a growing demand for more efficient, high-speed defect inspection (Kondo et al. 2021).

Since e-beam technology is slow, the industry is examining ways to simultaneously utilize more than one beam for scanning. In 2020, ASML released the eScan 1000 – the first-generation multibeam inspection system, shown in Figure 3. The tool features nine electron beams in a $3 \times 3$ array, and is reportedly 600% faster than previous e-beam wafer inspection tools.²

The throughput of the tool grows with the increasing number of beams. With single beam inspection, the speed bottleneck is the notoriously slow electron scanning procedure. Nowadays, with the increase in beams and subsequent increase in throughput, more stress is placed on the downstream data investigation pipeline. We refer to the data investigation pipeline as the tool’s *datapath*.

The objective of this work is to investigate SEM defect inspection given computational constraints. This is a feasibility study, centered on the scalability of the current state-of-the-art datapath. It is based on the understanding that the monetary cost and footprint (amount of space occupied by the hardware and software) of the datapath cannot scale linearly with the increase in the number of beams and subsequent

---

¹ A single beam can only scan a small area in the order of a few nanometers at a time, and has to sequentially move in a raster pattern to scan an entire wafer, which usually has a diameter of 300 millimeters (mm).

increase in tool throughput. The central question is: **How can we reduce the cost of defect inspection?** We consider the cost in euros to be a combination of the computational cost of the algorithm and the cost of its hardware implementation.

We first focus on developing a holistic methodology for quantifying the cost of defect inspection algorithms. We then apply this methodology to the *bottom up algorithms*, a simple approach based on the principle of defect inspection and simple denoising. Finally, we explore entirely novel learning-based alternatives which may be better suited for next-generation multibeam inspection tools.

The research questions can be summarized as follows:

(I) How can we *quantify* the computational cost (time complexity) of defect inspection algorithms?

(II) How can we *reduce* the computational cost of defect inspection algorithms?

(III) Are there more cost-effective hardware alternatives for defect inspection?

(IV) How can convolutional neural networks (CNNs) be leveraged for defect inspection?

The rest of the work is structured as follows. Section 2 outlines the nature and context of the problem. Section 3 answers research question (I). Section 4 outlines the available dataset. Section 5 details the bottom up approach and answers research questions (II) and (III). Section 6 describes our CNN methodology and answers research question (IV). Section 7 highlights some points worth considering and ideas for future work, and Section 8 states our conclusions.
2 Background

2.1 Scanning electron microscope inspection

Simply put, the SEM works by shooting electrons at a target and identifying how many low energy electrons are emitted. Two types of signal are usually detected: backscattered electrons and secondary electrons, which provide different types of information and make the SEM a versatile tool for different applications.

The resulting image is the convolution between the pattern being scanned and the electron-optical point spread function (PSF). The PSF is the impulse response of the imaging system (Li et al. 2018; Jia et al. 2020). The grayscale value of the image at a location represents the number of electrons detected at a location.

The size of the beam from which we shoot electrons is called the electron optical spot size. The spot size must be tuned based on the anticipated defect size – a smaller spot size is needed to identify smaller defects. In general, the diameter of the spot size should be smaller than the diameter of the defect. When the spot size is smaller than the defect size, neighboring pixels carry information about the defect, and this information has to be successfully combined.

The essence of this scanning methodology is that the pattern and the background light up differently and can therefore be differentiated. For instance, the background can appear bright, while the pattern appears dark. A defect is indicated by a discrepancy between the expected and actual brightness level in the SEM image. The detection is based on pixel comparison on adjacent dies. A more detailed explanation of the process follows in Section 5.

2.2 Defect types and causes

Defects can have many causes, including unexpected artifacts, shadowing effects, and particles on the mask (Yu et al. 2020). Every detected defect is handled as an indicator of process malfunction. Defects cannot be repaired on the wafer, and process engineers should correct the process itself to avoid such defects.

Integrated circuits come in a variety of different patterns: dot, line, logic, etc. Different patterns have different common defect types, which can additionally come in a variety of sizes and strengths. In dot patterns, defects can be classified as: missing dots, residue, extensions, bridges, critical distance and shifts. Example line pattern defects are shown in Figure 4.

![Figure 4: Overview of different types of defects on a ‘lines and spaces’ pattern. This is by no means an exhaustive list, but merely an illustration of common defect types. Photo via Lithography Defects.](image-url)
3 Methodology for comparing algorithms

In the following section, we will formalize four different aspects for analyzing the inspection methods. The goal is to provide a holistic framework for determining when an algorithm is good or better than another algorithm. This section answers research question (I): how do we quantify the cost of defect inspection algorithms?

3.1 Capture and nuisance rate

Capture and nuisance rate measure how good a defect inspection algorithm is at what we fundamentally want it to do: capture defects. Namely, it is not enough to ask how expensive it is to deploy the algorithm; we need to instead ask: how expensive is it to deploy an algorithm with a certain performance? Granted, an algorithm can be cheap yet ineffective, or expensive but effective.

For now, consider the defect inspection algorithm a black box. At its output, we get defect candidates. Correctly identified defects are called captured defects. Incorrectly identified defects are called nuisance defects. Defects which fail to be identified by the algorithm are missed defects.

The capture rate (CR) is the number of correctly identified defects as a percentage of the total number of defects. The nuisance rate (NR) is the number of incorrectly identified defects as a percentage of the total number of defects. We say that we achieve a certain capture rate at a certain nuisance rate, i.e., we achieve a certain performance at a certain cost. A satisfactory industry standard for most use cases is considered equal or above 90% capture rate at 10% nuisance rate.

![Comparing ROC Curves](image)

Figure 5: Comparing ROC curves (Ferraris 2018)

Each algorithm is characterized by a capture and nuisance rate pair. However, that does not always suffice to make a good comparison between algorithms. For instance, if algorithm A achieves 90% CR at 10% NR, and algorithm B achieves 90% CR at 5% NR, it is evident that algorithm B is better, since it achieves the same capture rate at lower nuisance. Likewise, if algorithm A achieves 90% CR at 10% NR, while algorithm B achieves 95% CR at 10% NR, it is again clear that algorithm B is better, since it achieves a better capture rate at the same nuisance. However, if algorithm A

---

Capture and nuisance rate seem to resemble precision and recall in machine learning, but it is worth highlighting the false parallel. The recall of the defect class is equivalent to its capture rate. However, precision is not the same as nuisance rate. Precision refers to the number of correctly identified defects as a percentage of all identified defects. Nuisance rate refers to the number of incorrectly identified defects as a percentage of the total number of defects. Hasty reading might falsely lead one to believe that the precision and nuisance rate add up to one, but that is not the case. While precision refers to the number of correctly identified defects and nuisance as the number of incorrectly identified defects, precision compares to all identified defects, whereas nuisance rate compares to all true defects.
achieves 92% CR at 12% CR, while algorithm B achieves 88% CR at 8% NR, neither satisfy the industry standard, and it is not immediately discernible which one is better – nor that they are the same.

For this reason, it is useful to have a capture and nuisance rate curve which describes the CR/NR pair at multiple points. This is similar to a receiver operating characteristic (ROC) curve. The ROC curve is a graphical plot that illustrates the trade-off of a binary classifier as its decision threshold is varied (Figure 5). The purpose of the capture-nuisance curve is the same. Again, while the ROC shows the true positive rate against the false positive rate, the capture-nuisance graph shows the capture against the nuisance rate.

3.2 Mathematical description

Computer algorithms are more often described in terms of pseudo code than mathematical notation. However, creating a mathematical description of the algorithms can be instrumental for a number of reasons.

First, mathematical notation serves as a communication mechanism with professionals in different disciplines, who may prefer a mathematical formalization of the problem, rather than an algorithmic one. Second, it enables a detailed comparison between algorithms. In fact, algorithms may on the surface have different steps, but detailed inspection can prove them to be fundamentally alike in nature. Although the concept might appear vague, this may be very useful, for example, in drawing from previous experiences in terms of suitable implementation. Finally, in the search for more efficient algorithms, it enables algorithm developers to identify steps which can mathematically be merged, broken down, replaced with more efficient alternatives, or disregarded altogether.

3.3 Numerical complexity and run time

Complexity theory of numerical analysis is the study of the number of arithmetic operations required to pass from the input to the output of a numerical problem (Smale 1997). We are interested in time complexity – the number of elementary operations on an input with a given size, where elementary operations are assumed to take a constant amount of time on a given machine and change only by a constant factor when run on a different machine. We consider addition, subtraction, multiplication and division as equivalent operations. It is worth noting that certain hardware, e.g. field-programmable logic arrays (FPGAs), provide only multiplications and additions (MACs – multiply-accumulates) as elementary operations.

Numerical complexity analysis is useful in two basic regards. First, it allows for a high-level overview of an algorithm’s steps in terms of their computational demand. Therefore, it is crucial for identifying bottlenecks. Second, numerical complexity allows for a first-order comparison between algorithms, not in terms of the nature of their operations (like the mathematical description), but in terms of the number of operations they require from input to output.

Although it is difficult to estimate execution time based on complexity due to intricacies in the hardware and implementation, it is possible to conduct a quantitative

---

3. Methodology for comparing algorithms

---

4 For instance, if an algorithm performs both filtering with a kernel and bicubic interpolation for the purpose of shifting, it may be useful to notice that both pass the image with a convolutional kernel and merging these steps is a feasible alternative. In another case, it may be useful to replace standard deviation with variance to avoid performing the square root operation, given that the same numerical stability applies.
comparison between algorithms or algorithm variants. To verify theoretical complexity estimates, we measure average run time across algorithms and compare that with theoretical expectations.

### 3.4 Bit depth

Bit depth refers to the number of bits used to store each sample. In our case, the sample is a grayscale image containing \( M \times N \) pixels. The higher the bit depth of the image, the more gray levels it can store. In the simplest case, a 1-bit image can store only two colors, black and white. Since we have 8-bit images, they store \( 2^8 \) gray levels per pixel.

Decreasing the bit depth can decrease computation time and memory requirements. In fact, the trend towards low-precision implementations for computationally expensive tasks has been prevalent for a few years, especially as it pertains to neural networks. In 2020, IBM published numerical representation techniques that enable the precision of training systems to be aggressively downscaled from 8 to 4 bits, enabling significant (seven-fold) hardware acceleration (Sun et al. 2020).

While some operations have fixed bit depth needs, others can be manually downscaled. For instance, multiplying two 8-bit samples results in a 16-bit sample by default, since that is the maximum number of pixels which can mathematically stem from the multiplication. However, if we divide two numbers, it is not as obvious how many float bits are necessary to represent the result to a satisfactory precision level. In such cases, it is critical to minimize the bit depth used.

While decreasing the needed bit depth implies a lower memory requirement by definition, implementation plays a large role in the resulting speedup. For example, decreasing the bit depth in a simple Python implementation will not necessarily result in a decrease in execution time. In fact, certain libraries have a specific default word size, for which they are optimized. The word size is simply the number of bits processed by the CPU as one unit. For instance, the numpy library is optimized for 32-bit float data, so converting to 16-bit might not result in significant speedup, if any. However, when implementing the algorithm at a lower level on a CPU (e.g., in C or C++) or different hardware altogether, decreasing bit depth can prove very significant in terms of performance.
4 Dataset

The available datasets consist of real SEM images capturing a contact holes pattern at 8, 10, 12 and 14 nm resolution and 8-bit pixels. All datasets contain images of the same dimensions – 4096 × 7754 pixels. Since all datasets cover the same area (in nanometers) at different resolutions (pixel sizes), they contain a different number of images. An overview of the number of images per dataset is given in Figure 6.

![Image of dataset overview]

Figure 6: Dataset overview by size for the different resolutions. When the image size is kept constant, more images are needed to cover the same area with 8 nm pixels, compared to 14 nm pixels.

The defects are programmed missing contact holes, which means contact holes were artificially removed by manipulating the original image. An example of a missing contact hole is shown in Figure 7. The diameter of the defect is 20 nm, occupying 2.5 pixels at 8 nm and 1.4 pixels at 14 nm. However, the alignment between the defect and the pixels can vary.

![Image of missing contact hole]

Figure 7: Missing contact hole – defect example (8 nm pixels).

Due to the different pixel sizes, the same number of pixels cover a different area in the different datasets. Thus, there are more repetitions of the pattern for the same image size (in pixels) at a larger pixel size compared to a smaller one. A sample of the dataset which illustrates this is shown in Figure 8.

There are 252 images with defects, which is around 25% of the images at 8 nm pixels. However, the defects span 20 nm in datasets with the pixel sizes 8, 10, 12,

---

5 We will round this image size to 4000 × 8000 pixels for simplicity in the continuation of this work.

6 Although the defect occupies 2.5 pixels at 8 nm, it is not necessarily limited to, in this case, 3 pixels. For instance, it is also possible for the defect to occupy 2 full pixels and .25 of two other pixels at each side, totaling 2.5 pixels. In this case, parts of the defect are spread over 4 pixels. In the 8 nm case, it is not a significant problem, since at least two full pixels contain defect information. At larger pixel sizes, however, poor alignment between the defect and the pixels can have a greater impact.
and 14 nm. If each pixel is considered an individual data point, only $7.8 \cdot 10^{-7}$ % of the data constitutes the defect class.

![Images of different pixel sizes](image.png)

Figure 8: Dataset overview: image of size $200 \times 200$ pixels at different pixel sizes.

During scanning, the pixel size contributes to the square of the imaging time (Kondo et al. 2021). Larger pixel sizes lead to faster scanning if the beam current is kept constant. In effect, if SEM images captured under conditions of large pixel size (low resolution) can be accurately inspected for defects, we can expect a significant improvement in throughput.
5  Bottom up approach for defect inspection

5.1  Introduction

The principle of defect inspection works by means of comparison of a target and reference image. The target and reference images are taken at the same place of different dies and, defects aside, should be identical. If that is the case, assuming the images are perfectly aligned, subtracting one image from the other should result in a zero matrix. If there is a defect in one of the images, the result will be a matrix with non-zero elements precisely at the defect location. The principle of defect inspection is illustrated in Figure 9.

In reality, it is not the case that images are identical. They differ because of several factors, including noise, distortions, and misalignment. For this reason, it is necessary to pre-process the images to reduce noise, align them to each other, and only then perform the defect inspection. Moreover, defect inspection cannot merely be a subtraction operation, since properties such as contrast or brightness can differ across images. It is therefore also necessary to normalize for these effects.

![Figure 9: The defect inspection principle is based on a comparison between images. ‘Target’ refers to the target image, ‘ref1’ and ‘ref2’ are the first and second reference images. ‘target – ref1’ is the pixel-wise subtraction of the target and the first reference image. If both reference images differ from the target at a particular location, the target image has a defect at that location. Since both the target and reference images are real SEM images, a detected difference between the two images does not conclusively determine which one of the images contains a defect. For this reason, it is imperative to use two reference images for each target image. Given the statistical infrequency of defect occurrence, it is virtually impossible for both reference images to contain a defect where the target does not. Thus, this simple method allows for reliable defect detection. This method involves labeling pixels in the target image based on their signal value, which represents how different the value at these coordinates is from the corresponding value in the reference images. An example of the signal distribution of an image is given in Figure 10. When sorting pixel locations in decreasing order based on their signal values, top defect candidates appear first. Since defects can span multiple pixels, neighboring defect candidates (neighbors touch in the horizontal, vertical or diagonal direction) should be grouped as a single defect. The resulting signal can either be the sum or maximum of the grouped values.](image-url)
Figure 10: Signal distribution of defect candidates for 8 nm pixel images, resulting from random mode analysis. The plot was created by binning the signal values in small intervals (.02) and plotting candidates per bin (occurrence). True defects are colored in red, while non-defects are green. In reality, the green part extends further left, but the graph is cut off to show only high signal pixels. There is an area where the signal value for defects and non-defects overlaps, which results in nuisance and missed defects, depending on how we set the threshold.

5.2 Inspection modes
The reference image can be a real SEM image or a golden-reference image from a database. If the reference image is a real SEM image, we call the inspection die-to-die (D2D) inspection. Otherwise, if the reference image is rendered from a database, the inspection is called die-to-database (D2DB) inspection. If the image is a real SEM image, i.e., we are performing D2D inspection, there are two options for sourcing the reference image. The reference image can either be a shifted version of the target image (we call this array mode), or it can come from an entirely different die (we call this random mode). The modes are illustrated in Figure 11.

Figure 11: Inspection modes overview. In array mode inspection, both reference images are shifted versions of the target image and come from the same die. Alignment is not needed since the shift is known in advance. In random mode, the reference images and the target image come from different dies. An alignment step is needed.
5.2.1 Array mode

Array mode involves comparing an image to a shifted version of itself. Obviously, this holds true only for repeating patterns, but such patterns are a large portion of all use cases. This mode has several benefits, including that an image is likely to be relatively uniform in different parts, alleviating issues related to varying brightness, contrast, or distortions.

Array mode is used for repeating patterns where the exact number of pixels per repetition is known. For this reason, the alignment step is unnecessary, as illustrated in Figure 11. Computation permitting, it is possible to improve the performance in this mode by using, instead of two reference images, multiple reference images by taking their average. This average can be closer to an ideal reference image as we average out more noise. Taking the difference can then merely be a subtraction between the target image and this reference image. We refer to the regular version of array mode as 'UF' (uniform filtering), whereas this version is called 'UFba' (uniform filtering with Bayes averaging). The 'UF' specification refers to the type of filtering performed in the filtering stage, which will be detailed in Section 5.4.

5.2.2 Random mode

If the wafer does not have a repeating pattern, the image should be compared to another die taken at the same location. This version is called random mode.

Random mode can be performed with or without interpolation. Interpolation pertains to the alignment step, and it refers to shifting the images a non-integer number of pixels to improve their alignment. In this work, we use the terms interpolation and sub-pixel shift interchangeably. Fundamentally, interpolation is the process of using known data to estimate values at unknown locations (Parsania and Virparia 2018). There exist different types of interpolation, including bilinear and bicubic interpolation (Acharya and Tsai 2007).

In bilinear interpolation, we use the four nearest neighbors to estimate the pixel value at a given location. Let \((x, y)\) denote the coordinates of the location to which we want to assign a pixel value and let \(v(x, y)\) denote that value. For bilinear interpolation, the assigned value is obtained using the equation \(v(x, y) = ax + by + cxy + d\) where the four coefficients \((a, b, c, d)\) are determined from the four equations in four unknowns that can be written using the four nearest neighbors of point \((x, y)\). Bicubic interpolation, for instance, involves the sixteen nearest neighbors of a point and generally is superior at preserving fine detail than bilinear interpolation.
5.3 Capture and nuisance rates

This section outlines the capture and nuisance rate performance across pixel sizes and bottom up algorithm variants. Figure 12 shows performance across pixel sizes, where it is visible that the highest performance is achieved using 8 nm pixels, and performance decreases for increasing pixel size. Larger pixel sizes are more difficult since there is less information per defect and a higher signal to noise ratio. The lines for the larger pixel sizes are cut off since the same area was scanned at all resolutions and less images are needed to encapsulate it at larger pixel sizes.

![Graph showing capture and nuisance rate performance across pixel sizes](image1)

Figure 12: Bottom up performance across pixel sizes for array mode (UF).

Figure 13 visualizes the performance for one pixel size across different algorithm variants. Overall, array mode outperforms random mode since the image is compared to a shifted version of itself, which is a more ideal reference image due to increased likelihood of constant properties (brightness, contrast, distortions) for the target and reference image, as well as knowledge of the exact shift. Array mode with multiple reference image averages (UFba) is better than using only two (UF), as expected. For random mode, we can increase the performance dramatically by interpolation, i.e., adjusting the offset between images to sub-pixel level precision.

![Graph showing capture and nuisance rate performance across modes for 10 nm pixels](image2)

Figure 13: Bottom up performance across modes for 10 nm pixels.
5.4 Reducing computational cost

Since capture and nuisance rates are important, it is tempting to simply choose the algorithm with the highest capture rate at the lowest nuisance rate. In practice, however, computational constraints come into play. We need to also consider the algorithm’s computational cost and suitability for different hardware implementations, as discussed in detail in Section 3.

5.4.1 Mathematical description of the algorithms

This section describes three central steps of the bottom up algorithm for D2D random mode without interpolation: filtering, alignment and difference. The high-level overview of the algorithm is as follows. First, one target image and two reference images are all independently passed through the filtering stage. Second, both reference images are aligned to the target image. Third, the target and reference images go through the difference calculation to obtain the final defect candidates.

5.4.1.1 Filtering

Let $f$ be an image of size $M \times N$. $f(x, y)$ refers to the pixel value of the image at the $x$th row and $y$th column. Further, $k \in N$ is the required kernel size; $0 \leq p \leq 1$ is the uniform parameter.

The kernel, also called the convolution filter, is denoted as $w$ and has size $k \times k$. It is a small matrix used in image processing for different ends, including noise filtering, sharpening, blurring, and edge detection (Gonzalez and Woods 2011). The particular function it serves is determined by its coefficients. The action is accomplished by performing a convolution between the kernel and the image, as detailed further. In this case, the kernel is initialized as

$$w_0 = \begin{bmatrix}
p^2 & p & \ldots & p & p^2 \\
p & 1 & \ldots & 1 & p \\
\vdots & \vdots & \ddots & \vdots & \vdots \\
p & 1 & \ldots & 1 & p \\
p^2 & p & \ldots & p & p^2
\end{bmatrix}$$

The filter $w_0$ is defined on $x, y \in [-a, a]$ for $k = 2a + 1$. This covers the case when $k$ is odd for notation convenience; the math is analogous when $k$ is even.

Upon initialization, the kernel is normalized with the sum of all elements as

$$w = \frac{1}{\sum_{s=-a}^{a} \sum_{t=-a}^{a} w_0(s, t)} w_0$$

The filtered image $g$ is obtained by convolving the image $f$ with the filter $w$, as follows

$$g(x, y) = w \ast f = \sum_{s=-a}^{a} \sum_{t=-a}^{a} w(s, t) t(x + s, y + t)$$

---

7 According to both practical experiments and a theoretical model of the scanning process, the ideal kernel size differs based on the pixel size. Such details are out of the scope of this work.

8 Convolution, strictly speaking, means computing the correlation but inverting one of the inputs (Gonzalez and Woods 2011), usually the convolutional filter, although it makes no difference. That is unnecessary here for multiple reasons, most obviously because the convolutional filter is symmetric and the inversion operation has no effect.

5. Bottom up approach for defect inspection
for $x = 1, 2, \ldots, M$ and $y = 1, 2, \ldots, N$.\(^9\) Alternatively, the convolution operation can be split into two separate steps along each of the axes, depending on which approach is more computationally efficient on the particular hardware.

### Alignment

Let $f_1$, $f_2$ be two images of equal size $M \times N$. We want to align the two images, i.e., find the offset of $f_2$ to $f_1$. We first take a patch of size $p \times p$ from the center of both images as follows

$$f' = \begin{bmatrix}
  f(\frac{M}{2} - \frac{p}{2}, \frac{N}{2} - \frac{p}{2}) & \cdots & f(\frac{M}{2} - \frac{p}{2}, \frac{N}{2} + \frac{p}{2}) \\
  \vdots & \ddots & \vdots \\
  f(\frac{M}{2} + \frac{p}{2}, \frac{N}{2} - \frac{p}{2}) & \cdots & f(\frac{M}{2} + \frac{p}{2}, \frac{N}{2} + \frac{p}{2})
\end{bmatrix}$$

The patches $f'_1, f'_2$ have respective average pixel values

$$m_1 = \frac{1}{p^2} \sum_{s=1}^{p} \sum_{t=1}^{p} f'_1(s, t)$$

$$m_2 = \frac{1}{p^2} \sum_{s=1}^{p} \sum_{t=1}^{p} f'_2(s, t)$$

We normalize the patches by computing

$$f''_1(x, y) = f'_1(x, y) - m_1$$

$$f''_2(x, y) = f'_2(x, y) - m_2$$

for every pixel in the patch, i.e. $x = 1, 2, \ldots, p$ and $y = 1, 2, \ldots, p$.

To find the optimal offset coordinates $(x_0, y_0)$, we compute

$$x_0, y_0 = \max_{x,y} \text{IFFT}(\text{FFT}(f''_1) \ast \text{FFT}(f''_2))$$

where FFT is the fast Fourier transform\(^{10}\), the complex conjugate of the FFT,\(^{11}\) and IFFT is the inverse Fourier transform which converts the signal back from the frequency domain. The coordinates of the maximal value of the expression denote the offset characterized by highest correlation between the two images (Gonzalez and Woods 2011).

---

\(^9\) $f$ needs to be zero-padded a pixels on each side (top, bottom, left, right) for the convolution with these boundaries. Alternatively, to avoid zero-padding, edge pixels can be ignored, and the convolution operation can be performed for $s = a, a + 1, \ldots, M - a$ and $t = a, a + 1, \ldots, N - a$.

\(^{10}\) Direct convolution may be more efficient than FFT for small kernel sizes based on our run time comparisons.

\(^{11}\) Without computing the complex conjugate (in practice, just rotating the matrix 180 degrees), the expression would result in the convolution instead of the correlation of the two images.

---

5. Bottom up approach for defect inspection
5.4.1.3 Difference

Given a target image \( f \) and two reference images \( f_1 \) and \( f_2 \), the goal is to find defective pixels in the target image. Assume the images have already been aligned to each other using the offset found in the alignment step (5.4.1.2), and all three images have the same size \( M \times N \).\(^{12}\)

We first compute the difference between the target image and each of the reference images as

\[
\begin{align*}
d_1 &= f - f_1 \\
d_2 &= f - f_2
\end{align*}
\]

by subtracting element-wise for each pixel.

The overall difference matrix is computed as

\[
d(x, y) = \sqrt{\max\left(d_1(x, y)d_2(x, y), 0\right)}
\]

for \( x = 1, 2, \ldots, M \) and \( y = 1, 2, \ldots, N \) where \( s_1, s_2 \) are the standard deviations of the difference matrices \( d_1, d_2 \) respectively.

To find the pixels containing defects, we check for which \((x, y)\) the value of \( d(x, y) \) exceeds a threshold \( t \). Alternatively, to find candidate defects, we sort the values in \( d \) in decreasing order and take the top \( n \).

5.4.2 Numerical complexity and run time

To estimate the numerical complexity of the algorithm variants, we first broke down the algorithm into steps (e.g., filtering, offsetting, difference – as detailed in Section 5.4.1), and then further broke down each step into necessary commands. We identified the variables which figure throughout the algorithm and summarized them in Table 1. Then, for each of the commands, we estimated complexity using their input arguments. Steps which do not scale with any of the input arguments are marked with \( c \) for constant (equivalent to \( O(1) \) in big-O notation). A summary of this process for the filtering step is given in Table 2 as illustration. The remaining steps are detailed in Appendix A, Tables 9 – 11.

<table>
<thead>
<tr>
<th>Variable</th>
<th>Typical value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>( n )</td>
<td>( 32 \cdot 10^6 )</td>
<td>image size in pixels ((M \times N))</td>
</tr>
<tr>
<td>( c )</td>
<td>( &lt;&lt; n )</td>
<td>constant</td>
</tr>
<tr>
<td>( k )</td>
<td>( 3 – 5 )</td>
<td>side of square kernel</td>
</tr>
<tr>
<td>( p )</td>
<td>( 100 )</td>
<td>side of square patch taken from image center</td>
</tr>
<tr>
<td>( f )</td>
<td>( 2 – 10 )</td>
<td>multiplication factor used for interpolation</td>
</tr>
</tbody>
</table>

By summing the complexities per command, we obtain an estimate for the number of operations per step. To validate the estimates, we plot the estimated complexity per algorithm step against the average run time per the respective step. The run time pertains to a Python implementation and is executed on a machine with the

\(^{12}\) In practice, \( M \times N \) is not actually the size of the initial images as the size changes slightly throughout the computation, but this notation is kept for simplicity.
Table 2: Complexity breakdown of filtering step

<table>
<thead>
<tr>
<th>Command</th>
<th>Complexity estimate</th>
</tr>
</thead>
<tbody>
<tr>
<td>Calculate kernel size based on input argument</td>
<td>( c )</td>
</tr>
<tr>
<td>Initialize kernel with zeros</td>
<td>( k^2 )</td>
</tr>
<tr>
<td>Initialize kernel with values</td>
<td>( ck^2 )</td>
</tr>
<tr>
<td>Compute multiplication factor</td>
<td>( 2k^2 )</td>
</tr>
<tr>
<td>Multiply kernel by factor</td>
<td>( k^2 )</td>
</tr>
<tr>
<td>Convolve</td>
<td>( nk^2 )</td>
</tr>
</tbody>
</table>

following specification: Intel(R) Core(TM) i7-8665U CPU @ 1.90GHz – 2.11 GHz; 16 GB RAM, no GPU. The implementation analyzes 8 images in parallel on 8 processes. The absolute run time is less relevant; we instead focus on the relative ratio between the different steps.

For random mode without interpolation, the resulting graph is given in Figure 15. The graph confirms that the overall trend matches between theory and practice, with the most significant deviation being in the 'difference' step, which is overestimated in theory but faster in practice. A possible explanation for this discrepancy is the usage of \textit{vectorized arithmetic operations} in numpy. In high-level languages like Python, vectorization describes the use of optimized, pre-compiled code written in a low-level language (e.g. C) to perform mathematical operations over a sequence of data, as a substitute for explicit iteration in the native language code.\textsuperscript{13} Aside from that, the filtering step is most expensive, as could be anticipated due to the convolution between the entire image and a kernel. This remark is not trivial and it is the essence of research question (II). Since based on the mathematical description, the filtering operation is separable for symmetrical kernels, it is worth considering this option further to reduce its computational cost.

Figure 14: Random mode without interpolation: theoretical complexity and run time. The figure presents a breakdown per algorithm step. The run time is given as the mean over the entire dataset with the error bars representing the standard deviation. The theoretical complexity scale is in \(10^8\) operations (ops).

Figure 15 shows the comparison between run time and theoretical complexity estimate for random mode with interpolation. Here, the most important thing to note

\textsuperscript{13} https://www.pythonlikeyoumeanit.com/Module3IntroducingNumpy/VectorizedOperations.html

5. Bottom up approach for defect inspection
is that in practice, the interpolation factor seems to greatly affect the computation time. In theory, while run time should be proportional to the interpolation factor, the interpolation step ('find offset') should not be more expensive than filtering. The reasoning behind this is that filtering scales with the image size $n$, while interpolation scales with $f_p \ll n$. Therefore, it is worth looking further into the implementation of the interpolation step. Since both filtering and interpolation are most expensive, and both involve convolution with a kernel, it is worth exploring a way of merging these steps as one, as well.

![Figure 15: Random mode with interpolation: theoretical complexity and run time. The figure presents a breakdown per algorithm step. The run time is given as the mean over the entire dataset with the error bars representing the standard deviation. The theoretical complexity scale is in $10^8$ operations (ops). We assume no re-use of information, so all three images are filtered, for instance.](image)

(a) Run time (s). The three colors represent values for the interpolation factor.  
(b) Theoretical complexity ($10^8$ ops). The interpolation factor is 5.

![Figure 16: Array mode variants: theoretical complexity and run time. The figure presents a breakdown per algorithm step, comparing the two variants of array mode: UF and UFba. The run time is given as the mean over the entire dataset with the error bars representing the standard deviation. The theoretical complexity scale is in $10^8$ operations (ops).](image)

(a) Run time (s).  
(b) Theoretical complexity ($10^8$ ops).

Figure 16 shows the comparison between the run time and theoretical complexity estimate for array mode in its two variants, UF and UFba. The theoretical estimate is consistent with the practical results, aside from the difference between the 'get

---

14 We perform the sub-pixel shift by first determining the correct sub-pixel shift, and then shifting the image. To determine the correct sub-pixel shift, we take a patch from the center of the image, upsize it by the interpolation factor, and find the shift as usual on the upsized path. We can determine the sub-pixel shift accordingly.
reference image’ step between the two modes. In reality, getting the reference image is less different for UF and UFba compared to what is established in theory. We suspect this also stems from the nature of vectorized operations in our Python implementation.

Table 3: Comparison between bottom up variants, array mode and random mode, in terms of numerical complexity and run time. Numerical complexity is given in Mega ($10^6$) operations per second (MOPS).

<table>
<thead>
<tr>
<th>Algorithm mode</th>
<th>Run time (s)</th>
<th>MOPS per image</th>
<th>Ops per pixel</th>
</tr>
</thead>
<tbody>
<tr>
<td>Random, no interpolation</td>
<td>9</td>
<td>886</td>
<td>28</td>
</tr>
<tr>
<td>Random, interpolation</td>
<td>26</td>
<td>1757</td>
<td>56</td>
</tr>
<tr>
<td>Array, UFba</td>
<td>13.5</td>
<td>1767</td>
<td>56</td>
</tr>
<tr>
<td>Array, UF</td>
<td>14.5</td>
<td>852</td>
<td>27</td>
</tr>
</tbody>
</table>

An overview of the required number of operations overall and per pixel, along with the run times, for the different variants of array and random mode is given in Table 3. Overall, the short answer to research question (II) is that we can reduce the cost of defect inspection algorithms by reducing the number of operations per pixel. Array mode UF is the algorithm with the lowest number of operations per pixel, and should be used whenever there is a repeating pattern. Random mode without interpolation is not significantly more expensive, but performs poorly for larger pixel sizes (already for 10 nm). Thus, for non-repeating patterns, a simple implementation equivalent to random mode with interpolation amounts to around 56 operations per pixel.

Although we include the average run time over 72 images for transparency, the algorithms are benchmarked on a personal laptop and Python implementation and the run times may not be reliable for absolute comparisons between the algorithms, as laptop CPU activity can vary unpredictably. When taken at face value, the run times suggest that random mode without interpolation is almost three times cheaper than random mode with interpolation. For array mode, the UFba implementation does not rank as more expensive than UF, contrary to what is established in theory.

### 5.5 Cost-effective hardware alternatives

#### 5.5.1 Bit depth

The results for the bit depth analysis are demonstrated in Figure 17. The analysis is performed by running the same algorithm but manually restricting its bit depth. For 8 nm pixels, we observe a difference in performance between 32 and 16-bit precision. For 8-bit precision, we obtain slightly lower capture at the same nuisance rates, although the distinction is small and the 8-bit variant also performs above the 90% CR at 10% NR threshold. For 14 nm pixels, the 16 and 32-bit exhibit identical performance, while 8-bit precision is consistently marginally lower. Overall, we can conclude that 16-bit precision suffices. 8-bit precision should not be excluded either, especially for use cases with less rigorous performance requirements.

A high-level overview of the 8-bit solution and the required bit-depth per step is given in Figure 18. We use fixed-point notation of the form <int.$\frac{u}{s}$> to represent the bit depth. In this notation, int refers to the required number of ‘integer bits’ (bits in the integer portion of the result) and $\frac{u}{s}$ refers to the required number of bits in the fractional part of the result (i.e., bits behind the comma). The $u/s$
Figure 17: Bit depth analysis performed by running the same algorithm at varying bit depth levels at analyzing the CR/NR graphs. In the graph for 8 nm (left), the 16 and 32 bit line overlap completely. The graph for 14 nm (right) is zoomed at lower nuisance rates to make the gap in performance apparent.

notation refers to whether the result needs to be signed or unsigned. Note that the sign bit is part of the integer portion of the result. The total number of bits at any stage can be computed as $\text{int} + \text{frac}$. For instance, $<8.4u>$ refers to 8 integer bits, 4 bits after the comma, unsigned.

Each step is then broken down into intermediary steps; an example for the filtering step is shown in Figure 19. The flow charts for the rest of the steps are presented in Appendix B, Figures 36 – 38. The 16-bit precision flow chart is not added for brevity, since it is analogous.

Figure 18: Overview of the flow chart for an 8-bit implementation.

Figure 19: Flow chart for the filtering phase in an 8-bit implementation.

5. Bottom up approach for defect inspection
The purpose of the flow chart is to detail the intermediate steps, noting where the bit depth can be decreased. For instance, multiplying a \(<8.0u>\) image with a \(<4.4u>\) kernel results in a \(<8.4u>\) output, which can be rounded to \(<8.0u>\) if the added precision is not needed. The multiplication operation does not change the integer portion of the result because \(0 \leq k \leq 1\) (but the kernel is specified as \(4.4u\) because the 8-bit precision diagram grants 8-bit precision by default, unless more is needed). It is also worth noting that the subtracting step requires \(<9.0s>\) output, since its result may be negative – this is an example of where the algorithm designer has no power to intervene.

Even if a step has certain bit depth as input and output, it can still require a larger intermediary bit depth. The greatest rise in terms of bit depth in the intermediary steps occurs in the alignment step as part of the FFT, where an FFT on \(<8.0u>\) input results in \(<16.0u>\) output, since 8-bits are needed for both the real and imaginary parts. Multiplying two such values results in \(<32.0u>\) since multiplying two 16-bit values can at most require 32 bits for storing the result (Figure 36. Such steps should be identified in advance and evaluated in terms of feasibility, for instance in terms of memory requirements.

### 5.5.2 Comparison to existing systems

This section answers research question (III) on whether there are more cost-efficient hardware alternatives. To reduce the overall cost, aside from reducing the total number of needed operations, it is also possible to make an operation cheaper. The two strategies are complementary and can be used in combination.

Field-programmable gate arrays (FPGAs) are versatile systems which are fully programmable and can theoretically implement any algorithm. However, some algorithms are more suited for a cost-efficient FPGA implementation. Based on the mathematical description and required bit depth of the bottom up algorithms, they are similar in nature to those of other systems (Systems A and B) which already implement FPGAs efficiently. These systems are inserted for comparison purposes: they have an entirely different application but deploy similar algorithms, which makes a solid basis for comparison.

In order to evaluate how the bottom up approach implemented on System C compares to Systems A and B in terms of cost, we consider how these systems compare to each other in relation to the necessary number of operations per pixel, data rate (in Giga pixels per second), and required number of operations per second (in Tera operations per second) to keep up with the data rate (Table 4). Unknown values are marked ‘NA’ for ‘not applicable’. To allow for a margin of error, the estimated operations per pixel for System C with the bottom up approach is taken to be 100 (which is around 3 times the estimate for random mode without interpolation and array mode UF, and roughly 2 times the estimate for random mode with interpolation and array mode UFba).

The absolute cost of goods is confidential; the numbers have been relativized by encoding the smallest value as \(x\) and expressing the rest in relative terms.\(^{15}\) In conclusion, the FPGA implementation, costing only \(2x\), is decidedly more cost-effective than scaling up the existing datapath linearly with increase in the number of beams.\(^{16}\) This shows the potential of reducing datapath cost through targeting the number of operations per pixel and the cost of one operation. Further, it highlights the importance of the mathematical description and detailed flowchart in determining

---

15. Note that this is a redefinition of the variable \(x\) used in Section 5.4.1.

16. The exact cost of the linear scaling is confidential.
algorithm similarity and communicating with experts, in order make ultimately make valid and informative comparisons between systems.

Table 4: Comparison between the bottom up and existing systems A and B. The cost of goods (monetary cost) of System C (bottom up) is estimated as a simple function of the operations per pixel, data rate, and operations per second of the systems which perform similar operations.

<table>
<thead>
<tr>
<th>System</th>
<th>A</th>
<th>B</th>
<th>C (bottom up)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Operations per pixel</td>
<td>NA</td>
<td>55</td>
<td>100</td>
</tr>
<tr>
<td>Data rate (Gpix/s)</td>
<td>NA</td>
<td>700</td>
<td>600</td>
</tr>
<tr>
<td>Operations per second (TOPS)</td>
<td>1320</td>
<td>45</td>
<td>60</td>
</tr>
<tr>
<td>Cost of goods (CoG)</td>
<td>10x</td>
<td>x</td>
<td>2x</td>
</tr>
</tbody>
</table>
6 CNNs for defect inspection

6.1 Introduction

In assessing feasibility of the datapath, it is reasonable to consider novel alternatives. While the bottom up approach is hand-crafted, it is also possible to perform defect inspection in a learning-based manner. In recent years, deep learning algorithms have successfully been employed for computer vision, most commonly through the use of convolutional neural networks (CNNs).

CNNs are a category of neural network with a track record of impressive results in a variety of computer vision tasks (Russakovsky et al. 2015), and are applicable for defect inspection. In fact, CNNs are widely used in industrial applications for surface defect detection (including steel, magnetic tile, rail, screen, solar and cell surface defect inspection). According to some sources (e.g., Qi, Yang and Zhong 2020), deep learning (DL) is likely to become one of the mainstream technologies for industrial vision in the future.

Thus far, deep learning has been investigated only for isolated steps of the datapath, such as denoising (Yu et al. 2020; Midoh and Nakamae 2019) and defect simulation (Wang, Yu and Pu 2021). Depending on desired output format, these tasks require their own specialized architectures from the deep learning family, such as generative adversarial networks (GANs).

On one hand, there exist potential limitations to using an end-to-end DL model. Primarily, it is largely a black box setting, which might make it difficult to debug if it does not perform as intended. In contrast, hand-crafted algorithms allow for tuning more interpretable parameters with a presumably observable and explainable effect on performance.

On the other hand, there exist potential benefits which might outweigh the drawbacks. CNNs can potentially result in improved speed, improved capture/nuisance rates, or both. Improved speed can stem from a decrease in either computation time, imaging (scanning) time, or both. Simple patterns can potentially be learned by simple neural networks, which can make deep learning more computationally efficient as compared to traditional algorithms. Additionally, networks may be capable of defect detection under rougher imaging conditions (e.g. larger pixel sizes), resulting in improved throughput due to decreased imaging time (Wang et al. 2020). Finally, the homogeneous nature of the convolution operations makes CNNs well-suited for GPUs without the need for custom implementation, which may be costly and time-consuming.

6.2 Context

While deep learning seems worth investigating, the question is how can it be incorporated. In principle, deep learning can fit into the existing datapath in three basic ways.

1. Deep learning can figure as a first step in a multi-stage approach. The goal is to implement an initial step which removes a majority of the uninteresting data, leaving only a small subset for further inspection. Ideally, the first step should be very computationally cheap, while the second can be more complex.

2. Deep learning can replace the entire pipeline with an end-to-end alternative. In the context of this approach, the network can output predictions per image

Although explainability is a benefit, it can still be argued that the necessity for hand-tuning and direct involvement of a human in the loop is a disadvantage of these algorithms.
or per pixel in the image. In the former, the network predicts per image whether the image contains a defect, as a classic binary classification task. In the latter, the network’s input and output have the same dimension, and the network predicts, per pixel, whether it is a defect or not, as a form of image segmentation.

(3) Lastly, deep learning can figure as a post-processing step to the existing output from the bottom up approach. This is the reverse multi-stage scenario from (1); the bottom up figures as a first step, while deep learning can be used on a subset of the data to fine-tune the results and increase the capture rate.

6.3 Related work

Deep learning methods have demonstrated significant performance improvement against traditional algorithms in many different areas of the semiconductor industry (Yu et al. 2020). Still, deep learning for defect inspection is a recent area of interest in research and, to our knowledge, DL-based end-to-end defect inspection does not yet exist in production. This section overviews two relevant studies which examine different approaches to this task.

Recently, Ouchi et al. (2020) proposed a DL-based encoder-decoder D2DB inspection that can distinguish a defect deformation from a normal deformation by learning the luminosity distribution in normal images. The authors pose the problem as anomaly detection (which falls into the category of unsupervised learning), and train the network exclusively using images without defects. This is beneficial since it is difficult to obtain defect images with sufficient variety. Due to the problem formulation, the authors claim the model can detect unseen defects. However, while reportedly the central upside of this architecture is powerful generalizability with respect to different types of defects, this method was tested only on a single pattern (lines and spaces) with three types of defects and one image per defect type. The entire dataset consists of 13 images per resolution (pixel size), 10 of which are non-defect images. Thus, the evaluation is undertaken on 23% defect and 77% non-defect data. The authors made an important initial investigation into anomaly detection for defect inspection and their evaluation resulted in no misdetections. However, the small size of the test set as well as the unrealistic class balance (in reality, defects occur in less than $10^{-6}$% of images) necessitates significant replication prior to making any solid conclusion.

Deep learning has also recently been adopted to avoid manual feature extraction and classification in optics, where the goal is to detect and classify nanoparticles deposited on wafers. Kolenov et al. (2020) performed CNN-based particle detection using four different particle diameters (40, 50, 60 and 80 nm) and a background class, achieving up to 95% accuracy – a two-fold increase from traditional algorithms. Interestingly, they only used 250 images per class and a 60-20-20 split for training, validation, and testing. The network architecture is simple; an input of $150 \times 150$ grayscale images, two convolutional layers operating with $5 \times 5$ kernel size (5 and 8 kernels per layer, combined with $2 \times 2$ maxpooling after each layer), and three fully-connected layers with 120, 84, and 5 neurons, respectively. Due to the similar nature of the problem, this study is a solid reference for our work, providing evidence that simple architectures and a small-sized training set can suffice to train a neural network for defect inspection. However, the class imbalance problem is not addressed in this work, either.
6.4 Experimental setup

6.4.1 Classification task description

Our initial investigation is formulated as a binary classification problem which aims to distinguish between images containing defects and images without defects. Since the task is rather straightforward and well-defined, it is reasonable to start with a simple feed-forward convolutional neural network, and progress to a more complex architecture if needed.

A simple overview of the defect classification pipeline is given in Figure 20. Since the raw images are very large ($4000 \times 8000$ pixels) and a network cannot be trained with so many input parameters, they have to be analyzed in parts. We call one part an image crop.

In context of the multi-stage approach (Section 6.2), binary classification implies labelling images as candidates for further inspection (potentially containing defects) and images which do not need to be inspected further (with no defects whatsoever). Unlike the bottom up approach, this problem setup does not result in an exact defect location, as we only predict a binary label per image. However, we do in fact know the location of the defect to precision equal to the size of the image crop. Depending on the size of the crop, this may be already sufficient for certain applications and qualify as an end-to-end solution.

Separating images into two classes of images with and without defects does not leave room for identifying the quantity or nature of defects. In context of the multi-stage approach, these questions can be answered by the second analysis step, if necessary. The statistical infrequency of defects makes it very unlikely for more than one defect to occur in a crop of a relatively small size.

![Defect classification process - overview](image)

Figure 20: Defect classification process – overview. A large SEM image is cropped into smaller parts. We perform binary classification for each image crop individually.

6.4.2 Input format

Clean ground truth SEM images for supervised training purposes can be difficult and expensive to achieve. Obtaining such data can require a large number of averages – averaging multiple images taken at the same region – for which the wafer has to be scanned repeatedly, which is not only slow but can also cause damage to the wafer (Yu et al. 2020). More trivially, SEM machines are working around the clock, so creating large and varied datasets may not be easily prioritized.

We overview the dataset at our disposal in Section 7, focusing among other aspects on the scarcity of defect data. As a reminder, if each pixel is considered an individual data point, only $7.8 \cdot 10^{-7}$ % of the data constitutes the defect class. More realistically, for $252 \times 252$ pixel crops (as often appears in literature, see Section 6.3), the dataset contains 0.0004% defect data. This is an extreme class imbalance, which is unlikely to be remedied by adjusting class weights while training, as confirmed by
our experiments. Note that decreasing the size of the sections may result in an easier task for the network due to the decrease in input parameters, but then we have less defect data (and the class imbalance worsens). For this reason, we expect that there is an optimal balance between the size of the image crop and the resulting dataset size.

To remedy this problem, we use strategic cropping to take advantage of the same defect multiple times. We crop the image near the defect such that the defect appears with all possible offsets within the crop, which is made possible by the repeating nature of the pattern. For instance, the same defect appears on the top-right, then top-middle, then top-left of the crop, and so on. Depending on the size of the crop, we can greatly increase the number of defect class examples in this way (the smaller the crop, the smaller the factor by which we can increase the amount of defect data). As for the non-defect class, we randomly sample from images in order to gain an equal class balance (50% defect data, 50% non-defect data). Some of the non-defect data is thus left unused.

Based on our experiments, it is also important to tune the size of the crop. For instance, the networks were struggling to train on crops with sizes $512 \times 512$ pixels. For $252 \times 252$ sized input, the dataset with $8$ nm pixels was performing well, but performance decreased for increasing pixel sizes. This is due to the fact that smaller pixel sizes contain a smaller surface area within the same number of pixels. Smaller surface area implies less features, i.e., less repetitions of the pattern (see Section 4), which results in a simpler task for the network to learn. Hence, we use smaller image crops as inputs for larger pixel sizes. Table 5 overviews the final input size per pixel size.

The optimal input size was tuned on the validation set (see Section 6.6.1). We only experimented with dimensions which resulted in a neat division of the whole image into sections. This strategy ensures that no areas of the image have to be inspected multiple times, saving needless computation. It is also possible to disregard this fact, if there is reason to believe that certain other dimensions are more optimal (e.g., dimensions which are multiples of two are often beneficial for neural networks, as well as square inputs).

Table 5: Relationship between pixel size, optimal input size, and the dataset size used. The optimal input size per resolution was tuned on the validation set. The dataset size is determined by the size of the input, since it determines how much defect data can be created from the images at our disposal.

<table>
<thead>
<tr>
<th>Pixel size (nm)</th>
<th>Input size</th>
<th>Dataset size</th>
</tr>
</thead>
<tbody>
<tr>
<td>8</td>
<td>$256 \times 242$</td>
<td>38,808</td>
</tr>
<tr>
<td>10</td>
<td>$195 \times 180$</td>
<td>35,280</td>
</tr>
<tr>
<td>12</td>
<td>$157 \times 149$</td>
<td>30,240</td>
</tr>
<tr>
<td>14</td>
<td>$128 \times 121$</td>
<td>30,240</td>
</tr>
</tbody>
</table>

6.4.3 **Network architectures**

The final architecture and hyperparameters were decided upon following experimentation detailed in Section 6.6. Following Kolenov et al. (2020), we keep the architecture simple, as that is the primary constraint for solving this problem.
Since the goal is to have an inexpensive architecture, we experimented with two main types of architectures: one using one-dimensional $1 \times 3$ kernels (1d) and another using more traditional, two-dimensional $3 \times 3$ kernels (2d). The difference between performing the convolution with each of these kernel types is given in Figure 21. The operation which involves convolving a kernel and an image is detailed in Section 5.4.1. However, while the bottom up algorithm requires manual defining of the kernel coefficients, these parameters are learned by the network in a learning-based approach.

![1x3 kernel](image1.png) ![3x3 kernel](image2.png)

(a) $1 \times 3$ kernel. (b) $3 \times 3$ kernel.

Figure 21: Comparison between architecture with one-dimensional (1d) and two-dimensional (2d) kernel. One square represents one pixel in the image.

6.4.3.1 Architecture with reference images

To help strengthen network learning, it may be useful to provide reference images along with the target image, similar to the defect inspection principle in the bottom up approach. In this way, multiple images can be given as channels in the input. One of these images will be the test image, always positioned in the same place in the stack. The idea is for the network to learn to detect defects by comparing the images to each other. The benefit of this approach is that learning is based on image comparison, and not memorizing the particular pattern on the die or the specific defect type, thereby potentially improving generalizability. Naturally, the increased size of the input contributes to added complexity and will likely necessitate the existence of deeper and wider network architecture. For now, we experiment with the same architecture from Section 6.4.3, with input containing the target and two reference images stacked together as different channels. We call this architecture ‘3d’ for notational convenience.

6.5 Mathematical description and computational complexity

Implementing a neural network broadly consists of training and evaluation phases. Training consists of several epochs (iterative passes through the training set) that involve both a forward propagation and backward propagation phase. Testing only consists of a forward propagation phase. After the network is trained once, it is only necessary to perform testing in production. We therefore focus on the computational complexity of forward propagation only.

Broadly speaking, every neural network has a basic computation unit called a neuron (Orponen 1994). The neuron receives input signals $x_1, \ldots, x_n$ from either other neurons or the outside environment (e.g. pixel values). It computes its output signal $y$ by adding together the $x_i$’s multiplied by some weights $w_j$, possibly subtracting a

---

18 All variables from previous sections are redefined for consistency with deep learning literature.

6. CNNs for defect inspection
bias term \( w_0 \), and finally applying some activation function \( \sigma \) to the result.

\[
y = \sigma \left( \sum_{j=1}^{n} w_j x_j - w_0 \right)
\]

(11)

Let \( f \) be an image of size \( M \times N \) as before (Section 5.4). In the case of a convolutional layer, the network iteratively learns coefficients for \( K \) kernels. Assume the kernel is two-dimensional and square, with size \( k \times k \).

For each pixel in the image, given a kernel of size \( k \times k \), per Equation 11, we need for perform \( y = \sigma(\sum_{j=1}^{K} k_j x_j - k_0) \), which amounts to \( 2k^2 + k + 1 \) operations per pixel per kernel.¹⁹ For fully-connected layers, each neuron has to perform \( 2n_{l-1} + n_{l-1} + 1 \) operations, where \( l \) refers to the current layer and \( n_{l-1} \) is the number of neurons in the previous layer.²⁰

An overview of the number of operations needed per layer for the 2d architecture for 10 nm pixels (Figure 27) is given in Table 6. We focus on 10 nm pixels for illustration purposes since they are likely to be used in practice. The total number of operations is \( 5.04 \cdot 10^6 \), totalling approximately 143 operations per pixel. Based on this, the simple 2d architecture should be in the same order of magnitude as the 100 operations per pixel bottom up estimate outlined in Section 5.4. The 1d architecture should be about half, while the 3d is double the cost.

Table 6: Operations (ops) per layer assuming 195 \times 180 \text{input} (optimal for 10 \text{nm pixels}) and 2d kernel

<table>
<thead>
<tr>
<th>Layer type</th>
<th>Parameters</th>
<th>Ops per neuron</th>
<th>Ops per layer</th>
</tr>
</thead>
<tbody>
<tr>
<td>Convolutional</td>
<td>( k = 3; \ K = 4 )</td>
<td>88</td>
<td>3.08 \cdot 10^6</td>
</tr>
<tr>
<td>Convolutional</td>
<td>( k = 3; \ K = 6 )</td>
<td>132</td>
<td>1.02 \cdot 10^6</td>
</tr>
<tr>
<td>Convolutional</td>
<td>( k = 3; \ K = 12 )</td>
<td>264</td>
<td>465 \cdot 10^3</td>
</tr>
<tr>
<td>Fully-connected</td>
<td>5 \cdot 10^3 \text{ in}; 32 out</td>
<td>15 \cdot 10^3</td>
<td>480 \cdot 10^3</td>
</tr>
<tr>
<td>Fully-connected</td>
<td>32 \text{ in}; 16 \text{ out}</td>
<td>97</td>
<td>1552</td>
</tr>
<tr>
<td>Fully-connected</td>
<td>16 \text{ in}; 8 \text{ out}</td>
<td>49</td>
<td>392</td>
</tr>
<tr>
<td>Fully-connected</td>
<td>8 \text{ in}; 1 \text{ out}</td>
<td>25</td>
<td>25</td>
</tr>
</tbody>
</table>

¹⁹ The total number of operations amounts to \( K(2k^2 + k + 1) \) if \( K \) is the number of kernels.

²⁰ The total number of operations per layer is \( n_l(2n_{l-1} + n_{l-1} + 1) \), where \( n_l \) is the number of pixels in the current layer.

6. CNNs for defect inspection
6.6 Results

6.6.1 Hyperparameter search

This section outlines the hyperparameter search performed in order to obtain a suitable activation function, kernel size, and network size. The weight initialization strategy is also important, and it is chosen according to its suitability for the chosen activation function based on mainstream recommendations (e.g., Géron 2017). For the hyperparameter search, the network was trained on the training set (60% of all available data) and the best performing parameters were chosen based on validation set performance (20% of all data). The final evaluation is performed on the remaining data which is the test set (20% of all data). The split is kept constant across experiments by always using the same random seed. Since the dataset is available at four different resolutions and a few network architectures, we chose one pixel size – 10 nm (since it is most likely to be used in practice) and one architecture type (2d, since it is most traditional and of moderate cost) for the hyperparameter search.

6.6.1.1 Activation functions

Figure 22 shows the activation types which were considered in this work, namely: Rectified Linear Unit (ReLU), Leaky ReLU, Exponential Linear Unit (ELU) and Scaled Exponential Linear Unit (SELU). The activation functions are given in Figure 22. The performance of the network per activation function is given in the ROC curves in Figure 23. In this case, SELU outperforms the rest of the activation functions, so it was chosen for the final networks. The idea behind SELU is that the exponential linear unit is scaled in order to inherently incorporate normalization to the network outputs, i.e., the network activations are kept at zero mean and unit variance (Klambauer et al. 2017). This property is meant to ensure stable training and remove the need for batch normalization. The outcome from our experiments is in alignment with popular recommendations (Géron 2017).

![Activation Functions](image_url)

Figure 22: Overview of the different activation functions tested in this work, where x is the activation function input and σ(x) is the activation function. Aside from ReLU, all other variants have tunable parameters. We consider their default values, as follows: Leaky ReLU is plotted with α = 0.01; ELU with α = 1; SELU with α = 1.6732 and scale = 1.0507.
Figure 23: ROC curves for different activation functions. The experiment is performed on the validation set (20% of all data) for 10 nm pixels while keeping the architecture (2d) and all other hyperparameters, aside from the weight initialization strategy, constant. Weights are initialized according to popular recommendations for each activation function. SELU outperforms other activation functions.

6.6.1.2 Kernel sizes

Figure 24 shows network performance across kernel sizes: 3, 5, and 7 used in all of the convolutional layers. The kernel sizes were chosen per convention as odd numbers. In principle, one defect is 20 nm, which can occupy at most 4 pixels (at 8 nm pixels) and at least 2 pixels (at 14 nm pixels). Although it is possible for defect information to be spread over 4 pixels, the majority of the data still is encapsulated in two pixels, which means that a 3 × 3 kernel should suffice to identify the defect, and 5 × 5 definitively contains all defect information. Based on the results, kernel size 5 performs best, followed by 3 and then 7. This is to be expected, since the 7 × 7 kernel implies that the outer pixels do not carry any information on the defect, so they only add noise, making the task more difficult for the network. Although kernel size 5 slightly outperforms 3, the difference in performance is not significant enough to invest the added computation. The rest of the work thus uses kernel size 3.

Figure 24: ROC curves for different kernel sizes. Kernel sizes are kept odd following convention. The experiment is performed on the validation set (20%) for 10 nm pixels, keeping the architecture (2d) and all other hyperparameters constant. The 5 × 5 outperforms the other kernels.
6.6.1.3 Trainable parameters

Figure 25 shows network performance as a function of the complexity of the network, as represented by the number of trainable parameters. When increasing the size of the architectures, we keep their structure and only double the kernels/units of each layer. Based on the results, it is evident that a bigger network is not necessarily more useful in solving the problem. A bigger network is more likely to overfit on the training data, even with early stopping and regularization set in place. Conversely, if the network is too simple, it is also not sufficient to learn the task at hand (underfitting).

![ROC curve for different number of trainable parameters](image)

Figure 25: ROC curve for different number of trainable parameters. The experiment is performed on the validation set (20%) for 10 nm pixels, keeping the architecture (2d) and all other hyperparameters constant. It shows that neither a simple model (red curve; underfitting) nor a complex model (purple line; overfitting) would be suitable for the task, while moderate complexity (green line) is optimal.

6.6.1.4 Image size

Figure 26 shows ROC curves for different input image sizes. We note that there is an optimal size for the input image size that is neither too big nor too small. As mentioned before, decreasing the size of the sections may result in an easier task for the network due to the decrease in input parameters, but then we have less defect data (and consequently less training data in general, when maintaining a 50/50 class balance). For this reason, there is an optimal value for the image crop.

![ROC curves for different image sizes](image)

Figure 26: ROC curves for different image sizes. The experiment is performed on the validation set (20% of all data) for 10 nm pixels, keeping the architecture (2d) and other hyperparameters constant. The graph shows that neither an image small that is too small (light blue line) nor too big (dark blue line) is suitable, while the moderate size (pink line) performs best.
### 6.6.1.5 Final parameters summary

The network with 2d architecture is presented in Figure 27. The rest of the network architectures are analogous. The networks consist of 3 convolutional layers with 4, 6, and 12 kernels (stride of 1, no padding) and 3 fully-connected layers with 32, 16, and 8 units, respectively. Each convolutional layer is followed by a maxpooling layer with stride 2. Recall that, since the goal is to have an inexpensive architecture, we experimented with two main types of architectures: one using one-dimensional $1 \times 3$ kernels (1d) and another using more traditional, two-dimensional $3 \times 3$ kernels (2d). The networks are trained with Scaled Exponential Linear Unit (SELU) activation in the convolutional and hidden layers, and Binary Cross Entropy (BCE) loss. The learning rate was set to $10^{-4}$ after manual tuning. Per common recommendations (Géron 2017), LeCun initialization (LeCun et al. 2012) is used with SELU. The input is standardized between 0 and 1 from its original 8-bit precision (0-256) for easier and more stable learning. The batch size is 32. To prevent overfitting, the early stopping criterion is for the lowest validation loss to be within the last five epochs. The split is 60-20-20 for training, validation, and testing, respectively.

![Figure 27: Network architecture diagram for 2d network.](image)

### 6.6.2 Test results

Figure 28 exemplifies the learning curves for neural networks. A complete overview of these graphs for all pixel sizes and network variants is given in Appendix C, Figures 39 – 50. The metrics we monitor are accuracy and loss (Binary Cross Entropy) after each epoch, as customary during network training and validation, but also the capture and nuisance rates, which are particularly informative for our use case.

Note that the networks were trained using an early stopping criterion which halts training when validation loss has not decreased in the last five epochs. However, the graphs exemplified are chosen such that the last visualized epoch represents the final model used during testing, i.e., the one showing best validation performance. All 1d and 2d networks hit the early stopping criterion within less than 10 epochs.

6. CNNs for defect inspection
The capture and nuisance rate per pixel size are shown in Tables 7 and 8 for 1d and 2d architecture respectively. In all cases, the metrics meet the necessary threshold of 90% CR at 10% NR. Since 1d architectures are more efficient, we can conclude that it suffices to use a 1d architecture for solving the problem at 50/50 class balance.

In general, the smaller pixels sizes exhibit better performance compared to the larger pixel sizes. This is to be expected from both theory and practice perspectives (bottom up results given in Section 5.4). In theory, larger pixel sizes mean coarser images, i.e., less information per defect. Additionally, in our datasets, the dose (the number of electrons deposited on the wafer) was kept constant for all resolutions, which results in a lower signal to noise ratio in the larger pixel sizes. These two factors contribute to a more difficult task for the larger pixel sizes. Moreover, examining the validation loss per pixel size shows a clear trend – the larger the pixel size, the larger the loss. This means that there is more certainty and better class separation for smaller pixel sizes compared to larger ones.

<table>
<thead>
<tr>
<th>Pixel size</th>
<th>Capture rate (%)</th>
<th>Nuisance rate (%)</th>
</tr>
</thead>
<tbody>
<tr>
<td>8</td>
<td>99.705</td>
<td>0.836</td>
</tr>
<tr>
<td>10</td>
<td>99.241</td>
<td>4.837</td>
</tr>
<tr>
<td>12</td>
<td>97.784</td>
<td>10.207</td>
</tr>
<tr>
<td>14</td>
<td>94.539</td>
<td>7.16</td>
</tr>
</tbody>
</table>

Figure 28: Learning curves: train and validation graphs for 8 nm pixels (2d)
The increasing difficulty of the task for increasing pixel sizes is also exemplified in Figures 29 and 30, which show the test probability values per pixel size for 1d and 2d architecture, respectively. It is apparent that although the performance of the network does not drastically decrease for increasing pixel sizes (and is well-within the 90% CR at 10% NR requirement), its certainty definitely decreases for increasing pixel sizes. In fact, while the network can almost perfectly separate between the classes at 8 nm, there is clear trend of increasing number of probabilities close to 0.5 as pixel size increases. Note that the predicted probabilities directly affect the loss, so this is merely another visual representation of the increase in loss. Note that this trend seems slightly less prominent in 2d architectures compared to 1d, as can be expected, since they perform the convolution with a square kernel and consolidate more spatial information in the process.

Figure 29: Output probabilities per pixel size for 50/50 balance between classes for 1d architecture. Values clustered around 0 are likely to contain no defects, while values close to 1 are likely to contain a defect. The 0.5 classification threshold is marked in red. More values clustered around the threshold means more uncertainty for the network.
Figure 30: Output probabilities per pixel size for 50/50 balance between classes for 2d architecture. Values clustered around 0 are likely to contain no defects, while values close to 1 are likely to contain a defect. The 0.5 classification threshold is marked in red. More values clustered around the threshold means more uncertainty for the network.

### 6.6.3 ROC curves

Figure 31 shows the ROC curves across pixel sizes and architectures. It is clearly visible that small pixel sizes perform better than larger ones in all cases and regardless of architecture (their graphs are most pinned to the upper left corner). This matches the discussion in the previous section about the increasing difficulty of the problem at lower resolutions. Additionally, 2d architectures always perform better than 1d architectures, which is to be expected as well, since they perform the convolution with a square kernel and consolidate more spatial information in the process. The discrepancy is smaller for 8 nm pixels, and larger for some of the bigger pixel sizes. This highlights the fact that the objective function is much easier to learn at higher resolutions, and that adding two-dimensional spatial information does not add a lot of value, or can more easily be compensated in the fully-connected part of the network.

The ROC curves can also be used to project the performance of the network when the class balance is varied. For instance, assume the network achieves 90% capture rate at 1% nuisance rate for 50/50 class balance. Then, when the class balance is 99.9% to 0.01%, we can expect to get 90% capture rate at 100% nuisance rate. This calculation foreshadows the drop in performance of our CNNs in the case of extreme class imbalance.
Figure 31: ROC across pixel sizes and architectures. The legend is sorted from best (most pinned to the upper left corner), to worst (least pinned to the upper left corner).

### 6.6.4 Capture and nuisance curves

In the bottom up approach, it is possible to vary the classification threshold in order to manipulate the capture and nuisance rate manually. This is an important property that adds interpretability and allows adaptation of the algorithm properties based on the use case.

Figure 32: Varying classification threshold across pixel sizes and architectures. The graph was created by manually moving the classification threshold for the test set in the range 0 to 1 with 0.05 step. Legend is sorted from best to worst.

In the deep learning approach, it is possible to do the same by varying the classification threshold. Namely, the network has one output node which gives a probability value between 0 and 1. Values below 0.5 are rounded to 0 and labelled as zeros (no defect), while values above 0.5 are rounded to 1 and labelled as 1 (defect).

Figure 32 was created by varying the classification threshold in the range from 0 to 1 with 0.05 step. The outcome shows a desirable property of neural networks in terms of their suitability for manual tuning. It also shows that, for example, if 12 nm
pixels perform at 97% CR at 10% NR (1d) and we want to decrease the NR, we can simply increase the classification threshold and obtain the desired result.

### 6.6.5 Results from architecture with reference images

Figure 33 compares the performance of the 3d architecture to the other network architectures. The learning curves for the 3d architectures are given in Appendix C.3 (Figures 47 – 50), where it is visible that it takes more epochs to train these networks (based on the same stopping criterion) compared to architectures without reference images, as is to be expected by the increase in input complexity and trainable parameters. The 3d architecture exhibits comparable performance to the 2d architecture with not much added benefit, yet added detriment in terms of evaluation complexity. This fact likely stems from the fact that the network is able to implicitly incorporate any reference image information during training without references too, simply from other training samples. Future work should consider using a larger network when training with reference images, to account for the increase in input complexity.

![ROC curves for all architectures (1d, 2d, 3d) across pixel sizes. Legend is sorted from best to worst. We plot performance on the test set.](image)

### 6.6.6 Evaluation time

As we are interested in computational efficiency, we overview the CNN evaluation times in Figure 34. Evaluation for both is performed on a machine with the following specification: Intel(R) Core(TM) i7-8665U CPU @ 1.90GHz -2.11 GHz; 16 GB RAM, no GPU. As it is performed on a personal laptop, moving to a server rack is expected to increase performance at least 10-fold, depending on the configuration.

On the left, we plot the average evaluation time per test set image over 10 test set runs. Note that images differ in size for the different CNNs. We sorted the box plots by increasing evaluation time, removing outliers which were more than two standard deviations away from the mean. These evaluation times are not representative since they stem from unrelated and unpredictable spikes in CPU usage. We chose box plots (as opposed to, for instance, mean and standard deviation error plots) due to the skewed distribution of the evaluation times, which makes standard deviation a misleading metric.

On the right, we normalize the evaluation times by extrapolating to evaluation for the same area in pixels. Note that this is still not the same area in nanometers. Now,
we can see a clear pattern in the evaluation time. First, there is a clear separation between the 1d and 2d architectures, and 1d architectures are faster than the 2d architectures. Second, using smaller pixel sizes, and consequently larger input crops, results in faster evaluation of the whole image. In other words, the more sections an image is cropped into for sequential analysis, the longer evaluation takes. This highlights the importance of tuning the size of the input crop such that the network is able to learn, while still keeping it as large as possible under that constraint. Furthermore, while it may take longer to evaluate at 14 nm pixels for the same area in pixels, the difference is unlikely to be relevant when considering the area in nanometers. Therefore, we can conclude that it is not significantly more expensive to evaluate test images at lower resolutions.

![Graphs showing evaluation time per test set image and normalized evaluation time per whole image](image)

(a) Average evaluation time per test set image. Images differ in size for different pixel sizes.

(b) Normalized evaluation time per whole image (4000 × 8000 pixels).

Figure 34: Evaluation time box plots displaying average run time (ms) over 10 test set runs. Sorted in increasing order; colors are not made to match on the left (not normalized) and right (normalized) subgraphs.

### 6.6.7 Representative class balance

Although the results given in this section present high capture rates at low nuisance rates, these evaluations were performed on a 50/50 class balance. In reality, this class balance is far from representative. In fact, defects are very rare, occurring at a frequency of $10^{-6}$ or even rarer – a fact rarely addressed in literature.

![Graph showing capture and nuisance rate](image)

Figure 35: Capture and nuisance graph on the representative class balance for 8 nm pixels and 2d architecture. The horizontal axis is scaled to $10^5$.

In order to determine the performance of these CNN architectures on the repres-
entative class balance, we used a network trained with the 50/50 class balance and evaluated it on a representative class balance. Figure 35 shows the results for 2d architecture and 8 nm pixels, which is the easiest problem to solve, as overviewed in previous results. The graph was created by moving the classification threshold manually in the range $10^{-5}$ to $10^{-4}$. It shows that in order to get 50% capture rate, we already need a 30,000% nuisance rate, which means that network performance degrades completely.

These results highlight the difficulty of the task at hand. Namely, although well-defined, the task is far from simple. Its subtlety lies in the infrequency and minuscule size of the defect, resulting in extreme class imbalance – a classic needle in a haystack problem. Since network performance degrades completely for 8 nm pixels, this implies major changes are needed, and thus further results for the more difficult pixel sizes were not explored in detail. Some promising ideas for future work in terms of alleviating this problem are presented in Section 7.
7 Discussion and future work

7.1 Class imbalance problem

As outlined in Section 6.6.7, while the simple architectures meet the industry standard at a 50/50 class balance, this is merely an initial experimental setup and not representative of the actual class balance, where this setup is no longer up to par. This section discusses several ideas for future work in this direction.

The dataset is said to be imbalanced if one class is underrepresented; in such cases, typically the minority class is the class of interest (Parsania and Virparia 2018). Class imbalance is detrimental since having few instances of the minority class means that the learning algorithm will be unable to generalize well and is likely to perform poorly in terms of relevant evaluation metrics on this class. Although the problem is common, limited systematic research is available in the context of deep learning. Some common strategies to deal with such cases are outlined in this section, largely following categorization by Buda, Maki and Mazurowski (2018).

7.1.1 Image augmentation

The original missing contact holes dataset consists of a singular type of defect – missing contact hole, where it is always the middle contact hole that is missing. While the network can easily learn this pattern, it is also important that it does not merely ‘memorize’ it, but is able to generalize to different scenarios. For instance, one of the other contact holes can be missing in a real scenario, or the defect can be of an entirely different type, e.g. merging contact holes. Additionally, the brightness and contrast of the image may vary, and we should ensure that network performance is not dependent on these inconsequential factors. For this reason, adding image augmentation in the training set should be a necessary next step in leveraging neural networks for defect inspection.

7.1.2 Architecture with difference image

Training neural networks is easier with simpler images. Following this idea, another way to strengthen learning is to provide the network with the difference image (see Section 5.4), rather than the raw image. The difference image will contain white noise in all positions except where there might be a defect. Like the previous approach, this aids generalization by removing the need to specialize to a particular pattern or type of defect. To execute this, we need to invest the added computation in aligning the images and computing the difference, necessitating that the bottom up approach is employed almost in its entirety. Still, due to the simplicity of the difference image, a very simple network may suffice to learn the task at hand. Moreover, this approach may be better suited for the reverse multi-stage approach discussed in Section 6.2, where CNNs are essentially used for bottom up post-processing.

7.1.3 Data-level methods

Data-level methods are under- and oversampling, both of which were implemented in this work (Section 6). Oversampling usually involves simply replicating randomly selected samples from the minority class. We oversampled by using different crops of the same defect instead of duplicating identical crops in order to create more variety. Future work should consider oversampling examples near the boundary between classes. Undersampling was also used in order to obtain the same number of examples per class, by randomly removing instances from the majority class. The disadvantage of this method is that it discards a portion of the available data. Future work would consider carefully selecting examples to be removed by identifying
redundant examples at the boundary between classes. It may be useful to visually hand-examine boundary cases to identify reasons why they might be problematic.

7.1.4 Classifier-level methods
Two major classifier-level methods are: thresholding and cost-sensitive learning. Thresholding is applied in the test phase and involves adjusting the decision threshold of a classifier. This method was employed to get the capture and nuisance rate graphs in Section 6.6. Cost-sensitive learning involves assigning a different cost for misclassification of examples from different classes. In this case, the minority class should be assigned a higher cost in the loss function such that these examples contribute more to the update of weights. Based on our experiments, neither of these methods were sufficient when used individually, but perhaps there exists a suitable combination.

7.1.5 Hybrid methods
The hybrid method would involve combining strategies from the data-level and classifier-level methods in different ways. One example is two-phase training (Havaei et al. 2017), where the network is pre-trained on a balanced dataset and fine-tuned on the original, imbalanced data. We did not attempt two-phase learning yet, but it is indeed a plausible next step.

7.1.6 Anomaly detection
A common approach for handling class imbalance, especially for extremely high imbalance like defect occurrence, is to pose the problem as an anomaly detection (novelty detection) problem (Buda, Maki and Mazurowski 2018; Kong et al. 2020). In this case, learning is centered around recognizing positive instances, rather than discriminating between classes. This task is usually performed with autoencoders trained on the identity function. The ultimate classification task is based on a reconstruction error between the input and output patterns. This method is beneficial for two reasons. First, it does not require a large and varied set of defect data, which is difficult to obtain. Second, it has the potential to detect any type of defect, not only ones that appear in the training set. In theory, anomaly detection is most suited as a major next step. However, it is likely that it should be used in combination with at least image augmentation (Section 7.1.1) and possibly the difference image input (Section 7.1.2). A possible downside is that the reconstruction error could behave similarly to the classification threshold and fail to clearly discriminate the few defective pixels.

7.2 Efficient processing of CNNs
This work explores computationally cheap CNNs for defect inspection. However, depending on the final implementation, it may be that high capture rates at low nuisance rates at the representative class balance require high computational complexity. This aspect is worth consideration as per the holistic algorithm evaluation framework proposed in Section 3.

The high computational complexity of neural networks is a well-documented problem. Sze et al. (2017) highlight that efficient processing of neural networks through techniques which improve energy efficiency and throughput is critical to their wide deployment. They overview a number of hardware optimizations which improve efficiency without sacrificing performance that can be relevant in a later stage of this project. Examples of such techniques are weight pruning (Molchanov et al. 2017) and binarized neural networks (Courbariaux et al. 2016). Another low-precision approach was outlined in Section 3 on bit depth, whereby IBM experienced a seven-fold increase in speed by aggressively downscaling on network precision.
8 Conclusion

Defect inspection is an integral part of the wafer fabrication process. As electron scanning throughput grows, more stress is placed on the downstream defect inspection algorithms. This work focuses on reducing the monetary cost of these algorithms through targeting both computational cost and hardware implementation.

We first develop a holistic methodology for evaluating defect inspection algorithms. Aside from considering algorithm performance in terms of capture and nuisance rate, we also quantify algorithm cost through an analysis of its mathematical description, numerical complexity and run time, and required bit depth. The contribution of this work is providing a quantifiable way of comparing algorithms.

We proceed with applying this framework to simple defect inspection algorithms based on the principle of defect inspection and simple filtering. We conclude that the monetary cost of the algorithm can be reduced by either reducing its number of operations per pixel, or reducing the cost of one operation (by changing the hardware). We outline a few variants of the simple algorithm and compare them to each other, listing both bottlenecks and opportunities for reducing the cost. When considering alternative hardware implementations, we highlight the importance of quantifying the required bit depth of the algorithm.

We also investigate a learning-based alternative for defect inspection which uses simple convolutional neural networks (CNNs) to perform binary classification. We consider three different types of architectures, differing in the dimension of the kernel (one or two-dimensional) and the type of input (with or without reference images). We perform a conventional hyperparameter search to find a suitable architecture and hyperparameters. We train the networks on a missing contact holes dataset at four different resolutions (8 – 14 nm), noting that tuning the optimal input size per resolution is crucial. We quantify CNN performance in terms of conventional machine learning metrics, capture and nuisance rate, as well as numerical complexity and run time, as per the previously proposed methodology.

We finally locate a significant problem with deploying CNNs to the representative class balance with respect to the extreme infrequency of defects, proposing ideas for improvement. Future work should strengthen learning by performing image augmentation and considering a different input format. Instances on the boundary between classes should be explicitly considered, both visually for insight, and given to the network for reinforcement. The class imbalance can also be addressed by hybrid methods, especially two-level learning. Anomaly detection seems to be a promising major next step due to its suitable format for this task, providing benefits such as lack of need for vast and diverse training data yet added generalizability in terms of the type of defect which can be identified.

Overall, while there seems to be potential for learning-based defect inspection at a low computational cost, a lot of work remains to be done. The contribution of this work lies in deploying the first proof of concept, and identifying crucial issues worth addressing on the road to learning-based defect inspection in production. Another important contribution is applying the holistic approach for algorithm comparison, which includes computational load, on CNNs, and highlighting the importance of focusing on validation with representative datasets – aspects which, based on our investigation, are often overlooked in deep learning studies for defect inspection.
References


References


8. References
Index

array mode, 17
backscattered electrons, 10
bicubic, 18
bilinear, 18
bottom up algorithms, 9
capture rate, 11
captured defects, 11
chip, 7
contact holes pattern, 14
convolution filter, 20
cost-sensitive learning, 47
datapath, 8
defect candidates, 11
die, 7
die-to-database, 17
die-to-die, 17
dose, 39

Electron beam, 7
field-programmable logic arrays, 12
integrated circuit, 7
interpolation, 18
interpolation factor, 24
kernel, 20
lithography, 7
missed, 11
nanometer, 7
neuron, 33
nuisance defects, 11
nuisance rate, 11
optical, 7
point spread function, 10
random mode, 17, 18
scanning electron microscope, 7
secondary electrons, 10
sub-pixel shift, 18
thresholding, 47
UF, 18

UFba, 18
vectorized arithmetic operations, 23
wafer inspection, 7
wafers, 7
Appendix

A Numerical complexity per algorithm step

This section shows the complete numerical complexity breakdown per algorithm step. All theoretical complexity graphs and operations per pixel calculations are based on these breakdowns.

Table 9: Complexity breakdown: offsetting without interpolation

<table>
<thead>
<tr>
<th>Command</th>
<th>Complexity estimate</th>
</tr>
</thead>
<tbody>
<tr>
<td>Get image center</td>
<td>c</td>
</tr>
<tr>
<td>Get patches around image centers</td>
<td>$p^2$</td>
</tr>
<tr>
<td>Get patch means</td>
<td>$2p^2$</td>
</tr>
<tr>
<td>Flip one patch</td>
<td>$p$</td>
</tr>
<tr>
<td>Fast Fourier transform (FFT)</td>
<td>$p \log p$</td>
</tr>
<tr>
<td>Find max of result</td>
<td>$p^2$</td>
</tr>
</tbody>
</table>

Table 10: Complexity breakdown: offsetting with interpolation

<table>
<thead>
<tr>
<th>Command</th>
<th>Complexity estimate</th>
</tr>
</thead>
<tbody>
<tr>
<td>Get image center</td>
<td>c</td>
</tr>
<tr>
<td>Get patches around image centers</td>
<td>$p^2$</td>
</tr>
<tr>
<td>Resize patch by factor (bicubic interpolation)</td>
<td>$4p(f - 1)$</td>
</tr>
<tr>
<td>Get patch means</td>
<td>$2(pf)^2$</td>
</tr>
<tr>
<td>Flip one patch</td>
<td>$pf$</td>
</tr>
<tr>
<td>Fast Fourier transform (FFT)</td>
<td>$pf \log pf$</td>
</tr>
<tr>
<td>Find max of result</td>
<td>$(pf)^2$</td>
</tr>
<tr>
<td>Make integer shift</td>
<td>c</td>
</tr>
<tr>
<td>Make the subpixel shift (bicubic interpolation)</td>
<td>$8n$</td>
</tr>
</tbody>
</table>

Table 11: Complexity breakdown: difference computation

<table>
<thead>
<tr>
<th>Command</th>
<th>Complexity estimate</th>
</tr>
</thead>
<tbody>
<tr>
<td>Subtract images</td>
<td>$n$</td>
</tr>
<tr>
<td>Crop based on offset</td>
<td>c</td>
</tr>
<tr>
<td>Compute standard deviation (x2)</td>
<td>$2 \cdot (2 \cdot n)$</td>
</tr>
<tr>
<td>Multiply standard deviations</td>
<td>c</td>
</tr>
<tr>
<td>Multiply difference matrices (x2)</td>
<td>$2n$</td>
</tr>
<tr>
<td>Find max between difference product and zero</td>
<td>$n$</td>
</tr>
<tr>
<td>Square root difference product</td>
<td>$n$</td>
</tr>
<tr>
<td>Divide numerator and denominator</td>
<td>$n$</td>
</tr>
</tbody>
</table>
B  Flow charts

This section overviews the flowcharts for the 8-bit implementation for the remaining steps – alignment, difference, subtraction. Note that this is not an exhaustive list of algorithm steps and does not capture the algorithm in its entirety, but rather it highlights the most expensive steps according to the numerical complexity estimates. The purpose is to illustrate the procedure by which the intermediary bit depth needs to increase and how it can be kept at a lower precision after each step.

![Align step – 8-bit flowchart](image)

Figure 36: Align step – 8-bit flowchart

![Difference step – 8-bit flowchart](image)

Figure 37: Difference step – 8-bit flowchart

![Subtraction step – 8-bit flowchart](image)

Figure 38: Subtraction step – 8-bit flowchart

C  Learning curves

This section provides an exhaustive overview of all learning curves for all CNNs mentioned in this work. The learning curves are organized in three sections according to the three different architectures: 1d, 2d, 3d and the pixel sizes are presented in increasing order. The purpose is to provide more detailed information on how fast the networks converge and how stable the training is in terms of validation accuracy, loss and CR/NR. As expected, the 3d networks take longest to train on average, due to the increased number of parameters. There does not seem to be a significant difference in terms of epochs needed to hit the stopping criterion for 1d and 2d architectures.
C.1 1d architectures

(a) Train and validation accuracy.  
(b) Train and validation loss.

(c) Train and validation CR and NR.

Figure 39: Train and validation graphs for 8 nm pixels (1d)

(a) Train and validation accuracy.  
(b) Train and validation loss.

(c) Train and validation CR and NR.

Figure 40: Train and validation graphs for 10 nm pixels (1d)
Figure 41: Train and validation graphs for 12 nm pixels (1d)

Figure 42: Train and validation graphs for 14 nm pixels (1d)
C.2 2d architectures

(a) Train and validation accuracy.  
(b) Train and validation loss.  
(c) Train and validation CR and NR.  

Figure 43: Train and validation graphs for 8 nm pixels (2d)

(a) Train and validation accuracy.  
(b) Train and validation loss.  
(c) Train and validation CR and NR.  

Figure 44: Train and validation graphs for 10 nm pixels (2d)
Figure 45: Train and validation graphs for 12 nm pixels (2d)

Figure 46: Train and validation graphs for 14 nm pixels (2d)
C.3 3d architectures

Figure 47: Train and validation graphs for 8 nm pixels (3d)

Figure 48: Train and validation graphs for 10 nm pixels (3d)
(a) Train and validation accuracy.
(b) Train and validation loss.
(c) Train and validation CR and NR.

Figure 49: Train and validation graphs for 12 nm pixels (3d)

(a) Train and validation accuracy.
(b) Train and validation loss.
(c) Train and validation CR and NR.

Figure 50: Train and validation graphs for 14 nm pixels (3d)