RRAM Characterization Analysis Notebook

This notebook contains Python analyses of data collected on the 256x256 (64K) TCAM array developed at Stanford. The TCAM has 4 WLs that are addressed by a single WL addr, and 2 BLs that are addressed by a single SL addr.

Notebook setup

In [ ]:

```
# Import libraries
import matplotlib.pyplot as plt
import numpy as np
import scipy
import scipy.stats as stats
import pandas as pd
from matplotlib import scale as mscale
from matplotlib import transforms as mtransforms
from matplotlib.ticker import Formatter, FixedLocator
from matplotlib.patches import Rectangle
```

In [ ]:

```
# Set up PPF scale for CDF plots
class PPFScale(mscale.ScaleBase):
    name = "ppf"

    def __init__(self, axis, **kwargs):
        mscale.ScaleBase.__init__(self, axis)

    def get_transform(self):
        return self.PPFTransform()

    def set_default_locators_and_formatters(self, axis):
        class VarFormatter(Formatter):
            def __call__(self, x, pos=None):
                return f"{x}"[1:]

        axis.set_major_locator(FixedLocator(np.array([0,.0001,.001,.01,.1,.2,.3,
        axis.set_major_formatter(VarFormatter())


    def limit_range_for_scale(self, vmin, vmax, minpos):
        return max(vmin, 1e-6), min(vmax, 1-1e-6)

    class PPFTransform(mtransforms.Transform):
        input_dims = output_dims = 1

        def ___init__(self, thresh):
            mtransforms.Transform.__init__(self)

        def transform_non_affine(self, a):
            return stats.norm.ppf(a)

        def inverted(self):
            return PPFScale.IPPFTransform()

    class IPPFTransform(mtransforms.Transform):
```

```
        input_dims = output_dims = 1

        def transform_non_affine(self, a):
            return stats.norm.cdf(a)

        def inverted(self):
            return PPFScale.PPFTransform()

# Register PPF scale
mscale.register_scale(PPFScale)
```

Calibration

Read voltage sweep

Here, we sweep the READ voltage across different programmed conductances and measure average current to see how "linear" the RRAM is. At all conductances tested, the I-V curves appear to be very linear. This ohmic behavior may simplify sense amplifier design and improve accuracy.

We will operate at 0.2V READ voltage during the remainder of these experiments, but this data confirms that the resistances/conductances measured will look similar at other READ voltages.

In [ ]:
```
# Load data and get bin values
names = ["addr", "v", "i"]
data = pd.read_csv("../data/read_voltage_sweep.tsv", sep="\t", names=names)
data["targbin"] = ((data["addr"] - data["addr"][0]) / 2) % 32

# Plot bin values
plt.title("Read Voltage Sweep (1024 cells w/ 32 levels)")
plt.xlabel("Voltage (V)")
plt.ylabel("Current (A)")
for targbin in range(32):
    gdata = data[data["targbin"] == targbin].groupby(["v"])
    plt.plot(gdata["v"].mean(), gdata["i"].mean())
plt.show()
```

FORMing

Load log file

The log file contains all the data from the dynamic FORM process. We need to parse it and look at some statistical properties of the FORMing process.

In this dataset, FORMing is done by dynamically incrementing VWL by 50mV starting from 1.3V until the cell resistance falls below 10kOhm. The VBL is fixed at 3.3V and the pulse width is 1ms.

In [ ]:
```
# Load log file
names = ["chip", "time", "addr", "operation", "data0", "data1", "data2", "data3"
data = pd.read_csv("../data/form2_log.csv", names=names)

# Get data from last FORM pulse and final resistance measurement only
read_data = data[data["operation"] == "READ"].groupby("addr").nth(-1)
```

```
form_data = data[data["operation"] == "SET"].groupby("addr").nth(-1)

# For SET operations: (data0, data1, data2, data3) = (VWL, VBL, VSL, PW)
# For READ operations: (data0, data1, data2, data3) = (res, g, meas_i, meas_v)
# res = resistance, g = conductance

data
```

FORMing VWL Distribution

As shown by the distribution, most cells FORM with a single pulse as indicated by the large bar at 1.3V (first pulse). The rest of the cells appear to FORM in a manner that looks like the right half of a Gaussian distribution.

In [ ]:
```
# VWL Distribution
form_data["data0"].hist(bins=np.arange(1.3, 2.51, 0.05001))
plt.title("FORMing VWL Distribution (65536 cells)")
plt.xlabel("VWL (V)")
plt.ylabel("Frequency")
plt.show()
```

FORMing Conductance Distribution

In [ ]:
```
# Conductance Distribution
(read_data["data1"] * 1e6).hist(bins=np.arange(100, 200, 5))
plt.title("FORMing Conductance Distribution (65536 cells)")
plt.xlabel("Conductance (uS)")
plt.ylabel("Frequency")
plt.show()
```

FORMing VWL Spatial Distribution

There seems to be a pattern in the FORMing voltages, repeating every 8 columns. This is probably a layout-dependent effect, as there is segmentation in the wordlines. The FORMing voltage appears to be lower at the edges.

In [ ]:
```
# FORMing VWL
mat = plt.imshow(form_data["data0"].values.reshape(256,256), interpolation=None)
plt.colorbar(mat)
plt.title("FORMing VWL Spatial Plot (65536 cells, V)")
plt.xlabel("BL/SL #")
plt.ylabel("WL #")
plt.show()
```

FORMing Conductance Distribution

From the FORMing conductance distribution, we can infer that it is more difficult to bring the darker stripes of cells below 10kOhm, so the VWL is higher for those cells accordingly.

In [ ]:
```
# FORMing Conductance
```

```
mat = plt.imshow(read_data["data1"].values.reshape(256,256) * 1e6, interpolation
plt.colorbar(mat)
plt.title("FORMing Conductance Spatial Plot (65536 cells, uS)")
plt.xlabel("BL/SL #")
plt.ylabel("WL #")
plt.show()
```

Multi-Bit Capability Evaluation

The first multi-bit test examines the post-programmed readout of 32 levels induced across the first 1024 cells of the array (32 cells/level * 32 levels). The programming method is DD-ISPP (dual-direction incremental step pulse programming), where the word line voltage is incremented until the cells fall into the target range. The main finding was that only the first bit line of the TCAM supports good multi-bit programming while the second bit line has huge relaxation problems. This may have something to do with the TCAM peripherals causing write disturbance. The data that led to this conclusion is presented below. All measurements are taken 15-30 minutes after programming. More information on the time-dependence of the distributions is given in the Retention section.

In [ ]:
```
# Load data
names = ["addr", "r"]
data = pd.read_csv("../data/read_mlc_bad.tsv", sep="\t", names=names)
data["g"] = 1/data["r"]

# Show multi-bit result
plt.rcParams["figure.figsize"] = (8,8)
data["bin"] = np.floor(data["g"] / 4e-6)
data["targbin"] = (data["addr"] - data["addr"][0]) % 32
matdata = data["bin"].values.reshape(32,32)
plt.matshow(matdata, vmin=0, vmax=32)
plt.title("MLC Capability (1024 cells w/32 levels)")
plt.xlabel("Target level #")
plt.ylabel("Cell #")
for i in range(len(matdata)):
    for j in range(len(matdata[0])):
        c = int(matdata[j,i])
        plt.text(i, j, str(c), va="center", ha="center")
plt.show()
```

In [ ]:
```
# Show distribution
plt.rcParams["figure.figsize"] = [6, 4]
plt.title("MLC Distributions (1024 cells w/32 levels)")
for i in range(32):
    rdata = data[data["targbin"] == i]
    plt.plot(sorted(rdata["g"].values * 1e6), np.linspace(0, 1, len(rdata)))
plt.xlabel("Conductance (uS)")
plt.ylabel("CDF")
plt.gca().set_yscale("ppf")
plt.xlim(0, 130)
plt.ylim(0.01, 0.99)
plt.tight_layout()
plt.show()
```

Note that every second cell (corresponding to the second bitline) above looks like it has deviated significantly from its programmed value. Next, we examine only cells on the first bitline to remove these outliers.

In [ ]:
```
# Load data
names = ["addr", "r"]
data = pd.read_csv("../data/read_mlc_good.tsv", sep="\t", names=names)
data["g"] = 1/data["r"]

# Show multi-bit result
plt.rcParams["figure.figsize"] = (8,8)
data["bin"] = np.floor(data["g"] / 4e-6)
data["targbin"] = ((data["addr"] - data["addr"][0]) / 2) % 32
matdata = data["bin"].values.reshape(32,32)
plt.matshow(matdata, vmin=0, vmax=32)
plt.title("MLC Capability (1024 cells w/32 levels)")
plt.xlabel("Target level #")
plt.ylabel("Cell #")
for i in range(len(matdata)):
    for j in range(len(matdata[0])):
        c = int(matdata[j,i])
        plt.text(i, j, str(c), va="center", ha="center")
plt.show()
```

In [ ]:
```
# Show distribution
plt.rcParams["figure.figsize"] = [6, 4]
plt.title("MLC Distributions (1024 cells w/32 levels)")
for i in range(32):
    rdata = data[data["targbin"] == i]
    plt.plot(sorted(rdata["g"].values * 1e6), np.linspace(0, 1, len(rdata)))
plt.xlabel("Conductance (uS)")
plt.ylabel("CDF")
plt.gca().set_yscale("ppf")
plt.xlim(0, 130)
plt.ylim(0.01, 0.99)
plt.tight_layout()
plt.show()
```

Now, we can try to pick out non-overlapping levels and examine them more carefully. We can get 5 non-overlapping levels here.

In [ ]:
```
# Define the non-overlapping levels
levels = [31, 26, 21, 12, 1]
```

In [ ]:
```
# Show multi-bit result
plt.rcParams["figure.figsize"] = (8,8)
data["bin"] = np.floor(data["g"] / 4e-6)
data["targbin"] = ((data["addr"] - data["addr"][0]) / 2) % 32
matdata = data["bin"].values.reshape(32,32)
plt.matshow(matdata, vmin=0, vmax=32)
plt.title("MLC Capability (1024 cells w/32 levels)")
plt.xlabel("Target level #")
plt.ylabel("Cell #")
for i in range(len(matdata)):
    for j in range(len(matdata[0])):
```

```
        c = int(matdata[j,i])
        plt.text(i, j, str(c), va="center", ha="center")
for level in levels:
    rect = Rectangle((level - 0.5, -0.5), 1, 32, linewidth=2, edgecolor="r", fac
    plt.gca().add_patch(rect)
plt.show()
```

In [ ]:

```
# Show distribution
plt.rcParams["figure.figsize"] = [6, 4]
plt.title("MLC Distributions (1024 cells w/5 levels)")
for i in levels:
    rdata = data[data["targbin"] == i]
    plt.plot(sorted(rdata["g"].values * 1e6), np.linspace(0, 1, len(rdata)))
plt.xlabel("Conductance (uS)")
plt.ylabel("CDF")
plt.gca().set_yscale("ppf")
plt.xlim(0, 130)
plt.ylim(0.01, 0.99)
plt.tight_layout()
plt.show()
```

Let's examine the distributions with more data now (16378 cells):

In [ ]:

```
# Load data
names = ["addr", "time", "r", "g"]
data = pd.read_csv("../data/retention-many.tsv.gz", names=names, sep="\t")
data["time"] = data["time"] - data.groupby("addr")["time"].transform("first")
data["targbin"] = ((data["addr"] - data["addr"][0]) / 2) % 32

data
```

In [ ]:

```
# Show distribution
plt.rcParams["figure.figsize"] = [6, 4]
plt.title("MLC Distributions (16384 cells w/32 levels, ~4hrs)")
tdata = data.groupby("addr").nth(-1)
for i in range(32):
    rdata = tdata[tdata["targbin"] == i]
    plt.plot(sorted(rdata["g"].values * 1e6), np.linspace(0, 1, len(rdata)))
plt.xlabel("Conductance (uS)")
plt.ylabel("CDF")
plt.gca().set_yscale("ppf")
plt.xlim(0, 140)
plt.ylim(0.001, 0.999)
plt.tight_layout()
plt.show()
```

If we filter down to 5 levels:

In [ ]:

```
# Define the non-overlapping levels
levels = [31, 27, 22, 14, 1]

# Show distribution
plt.rcParams["figure.figsize"] = [6, 4]
plt.title("MLC Distributions (16384 cells w/5 levels, ~4hrs)")
tdata = data.groupby("addr").nth(-1)
for i in levels:
```

```
    rdata = tdata[tdata["targbin"] == i]
    plt.plot(sorted(rdata["g"].values * 1e6), np.linspace(0, 1, len(rdata)))
plt.xlabel("Conductance (uS)")
plt.ylabel("CDF")
plt.gca().set_yscale("ppf")
plt.xlim(0, 140)
plt.ylim(0.001, 0.999)
plt.tight_layout()
plt.show()
```

If we filter down to 4 levels:

In [ ]:
```
# Define the non-overlapping levels
levels = [31, 24, 15, 2]

# Show distribution
plt.rcParams["figure.figsize"] = [6, 4]
plt.title("MLC Distributions (16384 cells w/4 levels, ~4hrs)")
tdata = data.groupby("addr").nth(-1)
for i in levels:
    rdata = tdata[tdata["targbin"] == i]
    plt.plot(sorted(rdata["g"].values * 1e6), np.linspace(0, 1, len(rdata)))
plt.xlabel("Conductance (uS)")
plt.ylabel("CDF")
plt.gca().set_yscale("ppf")
plt.xlim(0, 140)
plt.ylim(0.001, 0.999)
plt.tight_layout()
plt.show()
```

Retention and Read Noise

In this section, we evaluate how well the cells can maintain their conductance values after programming. We first program 1024 cells (using the first bitline). After each cell is programmed, we immediately measure the short-term relaxation by performing 1000 READs. After all cells are programmed, we measure the long-term relaxation by continuously reading all cells.

Room-Temperature Retention/Read Noise

In [ ]:
```
# Open read file
names = ["addr", "time", "r", "g"]
data = pd.read_csv("../data/retention.tsv.gz", names=names, sep="\t")
data["time"] = data["time"] - data.groupby("addr")["time"].transform("first")
data["dg"] = data["g"] - data.groupby("addr")["g"].transform("first")
data["absdg"] = np.abs(data["dg"])
data["targbin"] = ((data["addr"] - data["addr"][0]) / 2) % 32
data
```

In [ ]:
```
# Plot measured conductance vs. time
plt.title("RT Retention/Noise (64 cells w/32 levels)")
plt.xlabel("Time (s)")
plt.ylabel("Conductance (uS)")
for addr in range(0, 128, 2):
```

```
    adata = data[data["addr"] == addr]
    plt.plot(adata["time"], adata["g"]*1e6)
plt.xscale("log")
rect = Rectangle((3.8, 0), 3600, 128, edgecolor=None, facecolor="white", zorder=
plt.gca().add_patch(rect)
plt.text(116.96, 64, "Gap in measurement\nwhile programming cells", zorder=20, h
plt.show()
```

Let's look at the 5 levels we were considering before.

Example 1:

In [ ]:

```
# Plot measured conductance vs. time
plt.title("RT Retention/Noise (5 cells w/5 levels chosen)")
plt.xlabel("Time (s)")
plt.ylabel("Conductance (uS)")
for addr in np.array(levels)*2:
    adata = data[data["addr"] == addr]
    plt.plot(adata["time"], adata["g"]*1e6)
plt.xscale("log")
rect = Rectangle((3.8, 0), 3600, 128, edgecolor=None, facecolor="white", zorder=
plt.gca().add_patch(rect)
plt.text(116.96, 64, "Gap in measurement\nwhile programming cells", zorder=20, h
plt.show()
```

Example 2:

In [ ]:

```
# Plot measured conductance vs. time
plt.title("RT Retention/Noise (5 cells w/5 levels chosen)")
plt.xlabel("Time (s)")
plt.ylabel("Conductance (uS)")
for addr in np.array(levels)*2 + 128:
    adata = data[data["addr"] == addr]
    plt.plot(adata["time"], adata["g"]*1e6)
plt.xscale("log")
rect = Rectangle((3.8, 0), 3600, 128, edgecolor=None, facecolor="white", zorder=
plt.gca().add_patch(rect)
plt.text(116.96, 64, "Gap in measurement\nwhile programming cells", zorder=20, h
plt.show()
```

Example 3:

In [ ]:

```
# Plot measured conductance vs. time
plt.title("RT Retention/Noise (5 cells w/5 levels chosen)")
plt.xlabel("Time (s)")
plt.ylabel("Conductance (uS)")
for addr in np.array(levels)*2 + 256:
    adata = data[data["addr"] == addr]
    plt.plot(adata["time"], adata["g"]*1e6)
plt.xscale("log")
rect = Rectangle((3.8, 0), 3600, 128, edgecolor=None, facecolor="white", zorder=
plt.gca().add_patch(rect)
plt.text(116.96, 64, "Gap in measurement\nwhile programming cells", zorder=20, h
plt.show()
```

The conclusion is that READ noise and drift may become an issue when you place levels too close

together. If we remove the second (orange) level, we have 2 bits per cell and the READ noise does not appear to pose an issue after 7 hours.

Room-Temperature READ Distribution after ~7hrs

In [ ]:
```
# Show distribution
plt.rcParams["figure.figsize"] = [6, 4]
plt.title("MLC Distributions (1024 cells w/32 levels, ~7hrs)")
tdata = data.groupby("addr").nth(-1)
for i in levels:
    rdata = tdata[tdata["targbin"] == i]
    plt.plot(sorted(rdata["g"].values * 1e6), np.linspace(0, 1, len(rdata)))
plt.xlabel("Conductance (uS)")
plt.ylabel("CDF")
plt.gca().set_yscale("ppf")
plt.xlim(0, 130)
plt.ylim(0.01, 0.99)
plt.tight_layout()
plt.show()
```

Conductance Deviation vs. Initial Conductance

A good way to understand the stability of each conductance level is to look at the *conductance deviation plot* shown below. Intermediate conductances are less stable.

In [ ]:
```
# Get first and last conductance values
firstdata = data.groupby("addr").nth(0)
lastdata = data.groupby("addr").nth(-1)
plt.title("Conductance Deviation after ~7hrs")
plt.xlabel("Initial Conductance (uS)")
plt.ylabel("Change in Conductance (uS)")
for i in range(32):
    fdata = firstdata[firstdata["targbin"] == i]["g"]*1e6
    ldata = lastdata[lastdata["targbin"] == i]["dg"]*1e6
    plt.plot(fdata, ldata, 'o')
    plt.ylim(-40, 40)
plt.show()
```

Average Absolute Conductance Deviation vs. Time

Average conductance deviation vs. time seems to increase in a power-law fashion. There seems to be a difference in the exponent based on the conductance level. Intermediate conductances deviate the fastest.

In [ ]:
```
# Plot setup
plt.title("Average Conductance Deviation vs. Time")
plt.xlabel("Time (s)")
plt.ylabel("Mean Absolute Change in Conductance (uS)")

# Define time intervals to group together
time_intervals = np.concatenate([np.logspace(-2, 0.5, 50), np.logspace(3.7, 4.2,

# Define fit function
```

```
fit_fn = lambda x, a, k: a*x**k

# For each level
for i in levels:
    ldata = data[data["targbin"] == i]
    mean_abs_dg = ldata.groupby(pd.cut(ldata["time"], time_intervals))["absdg"].
    xdata = [c.left for c in mean_abs_dg.index.values]
    ydata = mean_abs_dg.values
    popt, pcov = scipy.optimize.curve_fit(fit_fn, xdata, ydata)
    print(f"Level {i}: a = {popt[0]}, k = {popt[1]}, CoV(a) = {pcov[0]}, CoV(k)
    p = plt.plot([c.left for c in mean_abs_dg.index.values], mean_abs_dg, "o")
    fitx = np.logspace(-2, 4.2)
    fity = fit_fn(fitx, *popt)
    plt.plot(fitx, fity, color=p[-1].get_color())
plt.gca().set_xscale("log")
plt.gca().set_yscale("log")
plt.show()
```

High-Temperature Retention

To perform high-temperature retention measurements, the chips were programmed and measured, then placed on a hot plate for 1 hour. After 1 hour, the chip was remeasured. The results are presented at three temperatures below.

One interesting observation is that the pre-bake distributions keep getting worse and worse. This means that the post-programmed relaxation is somehow affected by the baking. **TODO: do retention/relaxation/noise measurements after baking**

85C for 1hr

At 85C, there appears to be little effect of the baking on the resistance distributions.

In [ ]:

```
# Load data
names = ["addr", "r"]
data = pd.read_csv("../data/prebake85.tsv", sep="\t", names=names)
data = data[(data["addr"] // 2048) % 4 != 1]
data = data[(data["addr"] // 2048) % 4 != 2]
data["g"] = 1/data["r"]
data["targbin"] = ((data["addr"] - data["addr"][0]) / 2) % 32

# Show distribution
plt.rcParams["figure.figsize"] = [6, 4]
plt.title("MLC Distributions (8192 cells w/32 levels, prebake)")
for i in range(32):
    rdata = data[data["targbin"] == i]
    plt.plot(sorted(rdata["g"].values * 1e6), np.linspace(0, 1, len(rdata)))
plt.xlabel("Conductance (uS)")
plt.ylabel("CDF")
plt.gca().set_yscale("ppf")
plt.xlim(0, 140)
plt.ylim(0.001, 0.999)
plt.tight_layout()
plt.show()

# Load data
names = ["addr", "r"]
data = pd.read_csv("../data/postbake85.tsv", sep="\t", names=names)
```

```
data = data[(data["addr"] // 2048) % 4 != 1]
data = data[(data["addr"] // 2048) % 4 != 2]
data["g"] = 1/data["r"]
data["targbin"] = ((data["addr"] - data["addr"][0]) / 2) % 32

# Show distribution
plt.rcParams["figure.figsize"] = [6, 4]
plt.title("MLC Distributions (8192 cells w/32 levels, 1hr @ 85C)")
for i in range(32):
    rdata = data[data["targbin"] == i]
    plt.plot(sorted(rdata["g"].values * 1e6), np.linspace(0, 1, len(rdata)))
plt.xlabel("Conductance (uS)")
plt.ylabel("CDF")
plt.gca().set_yscale("ppf")
plt.xlim(0, 140)
plt.ylim(0.001, 0.999)
plt.tight_layout()
plt.show()
```

110C for 1hr

At 110C, there appears to be little effect of the baking on the resistance distributions.

In [ ]:

```
# Load data
names = ["addr", "r"]
data = pd.read_csv("../data/prebake110.tsv", sep="\t", names=names)
data["g"] = 1/data["r"]
data["targbin"] = ((data["addr"] - data["addr"][0]) / 2) % 32

# Show distribution
plt.rcParams["figure.figsize"] = [6, 4]
plt.title("MLC Distributions (16384 cells w/32 levels, prebake)")
for i in range(32):
    rdata = data[data["targbin"] == i]
    plt.plot(sorted(rdata["g"].values * 1e6), np.linspace(0, 1, len(rdata)))
plt.xlabel("Conductance (uS)")
plt.ylabel("CDF")
plt.gca().set_yscale("ppf")
plt.xlim(0, 140)
plt.ylim(0.001, 0.999)
plt.tight_layout()
plt.show()

# Load data
names = ["addr", "r"]
data = pd.read_csv("../data/postbake110.tsv", sep="\t", names=names)
data["g"] = 1/data["r"]
data["targbin"] = ((data["addr"] - data["addr"][0]) / 2) % 32

# Show distribution
plt.rcParams["figure.figsize"] = [6, 4]
plt.title("MLC Distributions (16384 cells w/32 levels, 1hr @ 110C)")
for i in range(32):
    rdata = data[data["targbin"] == i]
    plt.plot(sorted(rdata["g"].values * 1e6), np.linspace(0, 1, len(rdata)))
plt.xlabel("Conductance (uS)")
plt.ylabel("CDF")
plt.gca().set_yscale("ppf")
plt.xlim(0, 140)
```

```
plt.ylim(0.001, 0.999)
plt.tight_layout()
plt.show()
```

130C for 1hr

At 130C, there appears to be little new effect of the baking on the resistance distributions.

In [ ]:

```
# Load data
names = ["addr", "r"]
data = pd.read_csv("../data/prebake130.tsv", sep="\t", names=names)
data["g"] = 1/data["r"]
data["targbin"] = ((data["addr"] - data["addr"][0]) / 2) % 32

# Show distribution
plt.rcParams["figure.figsize"] = [6, 4]
plt.title("MLC Distributions (16384 cells w/32 levels, prebake)")
for i in range(32):
    rdata = data[data["targbin"] == i]
    plt.plot(sorted(rdata["g"].values * 1e6), np.linspace(0, 1, len(rdata)))
plt.xlabel("Conductance (uS)")
plt.ylabel("CDF")
plt.gca().set_yscale("ppf")
plt.xlim(0, 140)
plt.ylim(0.001, 0.999)
plt.tight_layout()
plt.show()

# Load data
names = ["addr", "r"]
data = pd.read_csv("../data/postbake130.tsv", sep="\t", names=names)
data["g"] = 1/data["r"]
data["targbin"] = ((data["addr"] - data["addr"][0]) / 2) % 32

# Show distribution
plt.rcParams["figure.figsize"] = [6, 4]
plt.title("MLC Distributions (16384 cells w/32 levels, 1hr @ 130C)")
for i in range(32):
    rdata = data[data["targbin"] == i]
    plt.plot(sorted(rdata["g"].values * 1e6), np.linspace(0, 1, len(rdata)))
plt.xlabel("Conductance (uS)")
plt.ylabel("CDF")
plt.gca().set_yscale("ppf")
plt.xlim(0, 140)
plt.ylim(0.001, 0.999)
plt.tight_layout()
plt.show()
```

Average deviation from programmed value vs. temperature after 1hr

This indicates that there is little impact of baking on the chip. Possibly, electrical pulses are required during the heating for there to be an effect. Or the hot plate baking method may not be the right approach.

In [ ]:

```
names = ["addr", "r"]
temps = [85, 110, 130]
```

```
absdg_levels = []
for temp in temps:
    # Load prebake data
    predata = pd.read_csv(f"../data/prebake{temp}.tsv", sep="\t", names=names)
    predata["g"] = 1e6/predata["r"]
    predata = predata[(predata["addr"] // 2048) % 4 != 1]
    predata = predata[(predata["addr"] // 2048) % 4 != 2]
    predata["targbin"] = ((predata["addr"] - predata["addr"][0]) / 2) % 32

    # Load postbake data
    postdata = pd.read_csv(f"../data/postbake{temp}.tsv", sep="\t", names=names)
    postdata["g"] = 1e6/postdata["r"]
    postdata = postdata[(postdata["addr"] // 2048) % 4 != 1]
    postdata = postdata[(postdata["addr"] // 2048) % 4 != 2]
    postdata["targbin"] = ((postdata["addr"] - postdata["addr"][0]) / 2) % 32
    postdata["dg"] = postdata["g"] - predata["g"]
    postdata["absdg"] = np.abs(postdata["dg"])

    absdg_levels.append([postdata[postdata["targbin"] == i]["absdg"].values.mean

plt.title("Temperature Dependence of Conductance Deviation (1hr)")
plt.xlabel("Temperature (K)")
plt.ylabel("Conductance Deviation (uS)")
for d, level in zip(np.array(absdg_levels).T, levels):
    plt.plot(np.array(temps) + 273, d, label=str(level))
```

Endurance

To measure endurance, we perform SET/RESET cycling using a hardware-timed pulse train without READing (fast mode). The pulse train waveform is verified with the oscilloscope. We also validate with a software-timed pulse train (slow mode), which generally exhibits good agreement with the hardware-timed pulse train. We test different SET conditions at 1us and 20ns. We are basically unable to get the cells to fail in a reasonable amount of testing time.

In [ ]:

```
# Show cycling up to 1e8
data = pd.read_csv("../data/endurance_log.csv", header=None)
data = data[data[3] == "READ"]
data["g"] = 1e6/np.float64(data[4])
plt.title("Endurance Cycling >1e8 Single Cell\nREAD every 5e4 cycles, VWL_SET=1.
plt.xlabel("Cycle #")
plt.ylabel("Conductance (uS)")
plt.plot(np.arange(len(data["g"][::2]))*5e4, data["g"][::2], "b.")
plt.plot(np.arange(len(data["g"][1::2]))*5e4, data["g"][1::2], "r.")
plt.show()
```

In [ ]:

```
# Show cycling up to 1e7 for different VWLs
data = pd.read_csv("../data/endurance_log_many.csv", header=None)
data = data[data[3] == "READ"]
data["g"] = 1e6/np.float64(data[4])
plt.title("Endurance Cycling >1e7 Single Cell\nREAD every 5e4 cycles, VWL_SET=1.
plt.xlabel("Cycle #")
plt.ylabel("Conductance (uS)")
for addr in [768, 770, 772, 774, 776, 778]:
    adata = data[data[2] == addr]
    plt.plot(np.arange(len(adata["g"][::2]))*5e4, adata["g"][::2], "b.")
```

```
    plt.plot(np.arange(len(adata["g"][1::2]))*5e4, adata["g"][1::2], ".")
plt.show()
```

Sweeps

TODO

Programming Methods and Tuning

Here, we evaluate various write-verify programming methods. In particular, we test out FPPV, ISPP, DD-ISPP,Smart-ISPP, Smart-DD-ISPP, GSR, and RADAR. Each algorithm uses fixed pulse width and assumes block RESET operation is available (GSR and (Smart)-DD-ISPP methods do not need block RESET). The algorithms are described below and each has tradeoffs in terms of:

1. *Number of tunable parameters:* fewer means that the parameter tuning process is more straightforward and less time-consuming to configure, but greater number of parameters typically leads to lower pulse count
2. *Dynamic range of WL/BL/SL voltages:* having fewer voltage levels to apply simplifies peripheral design in the array
3. *Number of pulses required:* having fewer is always better, but usually requires more parameters to tune
4. *Difficulty of implementation:* simpler algorithms require less control logic

	Tunable Parameters per Level	Difficulty of Implementation	Dynamic Range Required	Pulse Count (Estimate)
FPPV	1	1	1	5
ISPP	1	1	2	5
DD-ISPP	2	2	2	4
Smart-ISPP	2	2	2	3
Smart-DD-ISPP	3	3	2	2
RADAR	5	4	3	1
GSR	8	3	3	2

The tradeoffs for each algorithm will be described further below in the corresponding section.

TODO

FPPV

FPPV stands for Fixed Pulse Programming Voltage method, and as its name suggests, it involves the use of fixed amplitude pulses to program a cell to a given range. In terms of dynamic range: there is one BL level (for SET) and one SL level (for RESET). For WL voltage, there is one level for each targeted programming range. Hence, for targeting of `N` programming ranges, only `N` DAC levels

are necessary for the WL, which is advantageous in terms of area (and potentially power) when designing peripheral memory circuits.

The idea of the algorithm is that when targeting a range `i`, you apply a pulse with the WL voltage that gets you there the fastest. As long as your conductance is lower than the target, you keep applying SET pulses with the corresponding amplitude, and when the conductance goes above the target, you use a strong RESET pulse to get back to HRS and try again.

FPPV is relatively easy to tune; first, you perform a sweep to determine how the WL voltage affects the final conductance, then you pick the WL voltages that result in the desired conductance range being achieved in minimum number of pulses.

TODO: make `fppv_tune.py`

ISPP

ISPP stands for Incremental Step Pulse Programming method, and as its name suggests, it involves the use of increasing (WL voltage) amplitude pulses to program a cell to a given range. In terms of dynamic range: there is one BL level (for SET) and one SL level (for RESET). For WL voltage, there is typically a larger degree of tunability required, and hence a higher resolution DAC will need to be used than for FPPV.

The idea of the algorithm is that when targeting a range `i`, you start from `VWL = 0V` and keep stepping up the WL voltage until you get to the target. As long as your conductance is lower than the target, you keep applying SET pulses with the corresponding amplitude, and when the conductance goes above the target, you use a strong RESET pulse to get back to HRS and try again.

ISPP is very easy to tune; there is only one parameter, the step size. To tune it, you can thus simply evaluate the algorithm with different step sizes.

TODO: make `ispp_tune.py`