# **Challenge #17: Sorting on a Systolic Array**

## **Objective**

To implement Bubble Sort using a **systolic array architecture** in Python and evaluate its computational performance across varying problem sizes. This project aims to bridge the conceptual gap between sorting algorithms and hardware-parallel computation models.

## **Prompts Used**

1. "What is a systolic array and how can Bubble Sort be implemented on one?"
2. "Design a systolic array for sorting and explain its structure."
3. "Implement Bubble Sort using a systolic model in Python with object-oriented components."
4. "Benchmark this systolic sorter for different input sizes."
5. "Visualize and compare its performance to a baseline such as Python’s built-in sort."

## **Methodology**

### **1. Understanding the Architecture**

* A **systolic array** is a network of simple processors (PEs) that rhythmically compute and pass data through the array in a pipelined fashion.
* In Bubble Sort, we exploit its nature of comparing and swapping **adjacent** elements — perfect for a **systolic structure**.
* Each **PE** holds one value and communicates with its immediate neighbor, making a decision to swap based on comparisons.

### **2. Implementation in Python**

* A PE class was defined to simulate each processing element’s local behavior.
* The systolic\_bubble\_sort() function creates an array of these PEs and applies alternating **odd** and **even** comparison passes to propagate smaller values to the front — akin to how the “bubbles” float in traditional Bubble Sort.
* We used random.sample() to generate diverse input datasets for each simulation run.

### **3. Performance Measurement**

* We selected increasing input sizes: **10, 100, 1000, 2000, 4000, and 8000** elements.
* For each size, the execution time of the systolic Bubble Sort was measured using time.time().
* We also benchmarked Python’s built-in sorted() on the same data for comparison.

### **4. Visualization**

* matplotlib was used to plot execution time vs. input size for both algorithms.
* Each point on the graph was annotated with the precise runtime.
* A color-coded legend and clear gridlines improved interpretability.

## **Code Overview**

### **Processing Element Class**

class PE:

def \_\_init\_\_(self, value):

self.value = value

def compare\_and\_swap(self, neighbor):

if self.value > neighbor.value:

self.value, neighbor.value = neighbor.value, self.value

### **Systolic Sort Driver**

def systolic\_bubble\_sort(data):

n = len(data)

pes = [PE(val) for val in data]

for t in range(n - 1):

for i in range(0, n - 1, 2): # Even phase

pes[i].compare\_and\_swap(pes[i + 1])

for i in range(1, n - 1, 2): # Odd phase

pes[i].compare\_and\_swap(pes[i + 1])

return [pe.value for pe in pes]

### **Benchmarking Loop**

sizes = [10, 100, 1000, 2000, 4000, 8000]

systolic\_times = []

builtin\_times = []

for size in sizes:

data = random.sample(range(size \* 10), size)

# Systolic sort timing

start = time.time()

systolic\_bubble\_sort(data.copy())

end = time.time()

systolic\_times.append(end - start)

# Built-in sort timing

start = time.time()

sorted(data.copy())

end = time.time()

builtin\_times.append(end - start)

### **Performance Plot**

plt.plot(sizes, systolic\_times, label="Systolic Bubble Sort", marker='o', color='orange')

plt.plot(sizes, builtin\_times, label="Python Built-in Sort", marker='s', color='green')

## 

## **Results**

### **Execution Time Table**

| **Input Size** | **Systolic Sort (s)** | **Built-in Sort (s)** |
| --- | --- | --- |
| 10 | 0.00005 | 0.00001 |
| 100 | 0.003 | 0.00002 |
| 1000 | 0.14 | 0.0001 |
| 2000 | 0.51 | 0.0003 |
| 4000 | 2.07 | 0.0007 |
| 8000 | 9.19 | 0.0015 |

* Python’s sorted() is based on Timsort (O(n log n)) and remains efficient even at scale, while systolic Bubble Sort suffers from O(n²) complexity.

## **Conclusions**

* The systolic model for Bubble Sort provides valuable insight into parallel, hardware-style computation.
* It clearly illustrates how a distributed system of simple PEs can collaboratively achieve sorting.
* However, in practical terms, Bubble Sort remains inefficient for large data, regardless of the hardware model.
* This project demonstrates the tradeoff between conceptual simplicity and real-world performance — especially critical in embedded and VLSI system design.