Chipyard Configurator

A RISC-V PARAMETERIZATION TOOL

Konstantinos Pikoulas
Georgios Klempetsanis

What is RISC-V ?

RISC-V is an ISA whose development began in 2010 at the University of California, Berkeley

Why do we care?

Because it is **open standard**!! RISC-V is offered under royalty-free open-source licenses.

What is Chipyard ?

Chipyard is a framework composed of a collection of tools and libraries designed to provide an integration between open-source and commercial tools for the development of systems-on-chip.

Why are we using it?

We are using Generators. A Generator can be thought of as a generalized RTL design, written using a mix of meta-programming and standard RTL. This type of meta-programming is enabled by the Chisel hardware description language.

RISC-V®

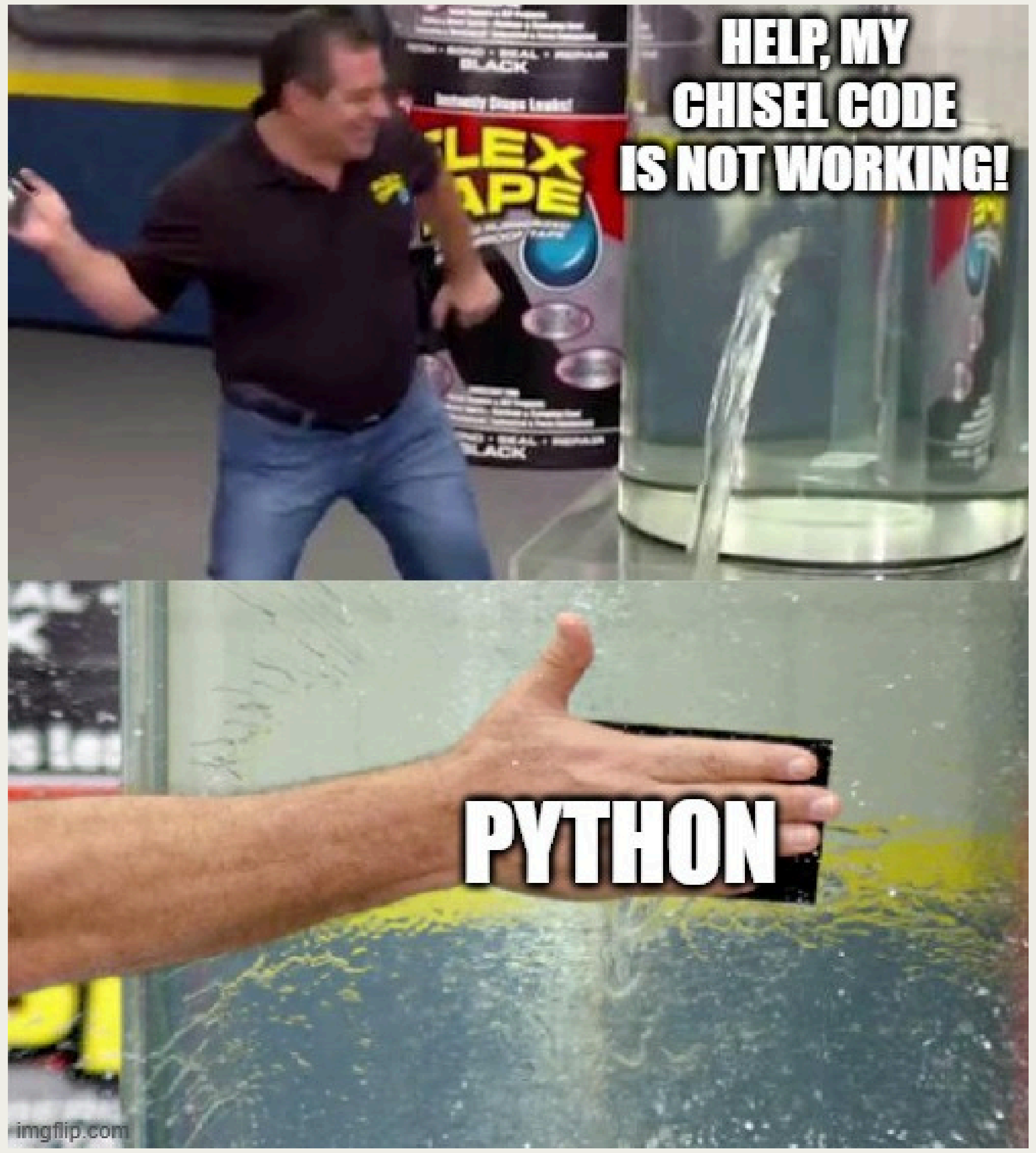
CHIPYARD

But chisel code is so hard to maintain!

That’s why we made a python wrapper that generates chisel code for you!

How it works...

Read config.json	*How many cores?* *L1/L2 cache configuration?* *Virtual Addressing?*
Validate configuration	*Is it a power of 2 (e.g. cache ways)?* *Is the configuration valid?*
Generate Scala files.	*Generate the chisel code based on the validated configuration.*
Build	*Build the RTL.*



CHIPYARD

Project Flow: From Config to Benchmark

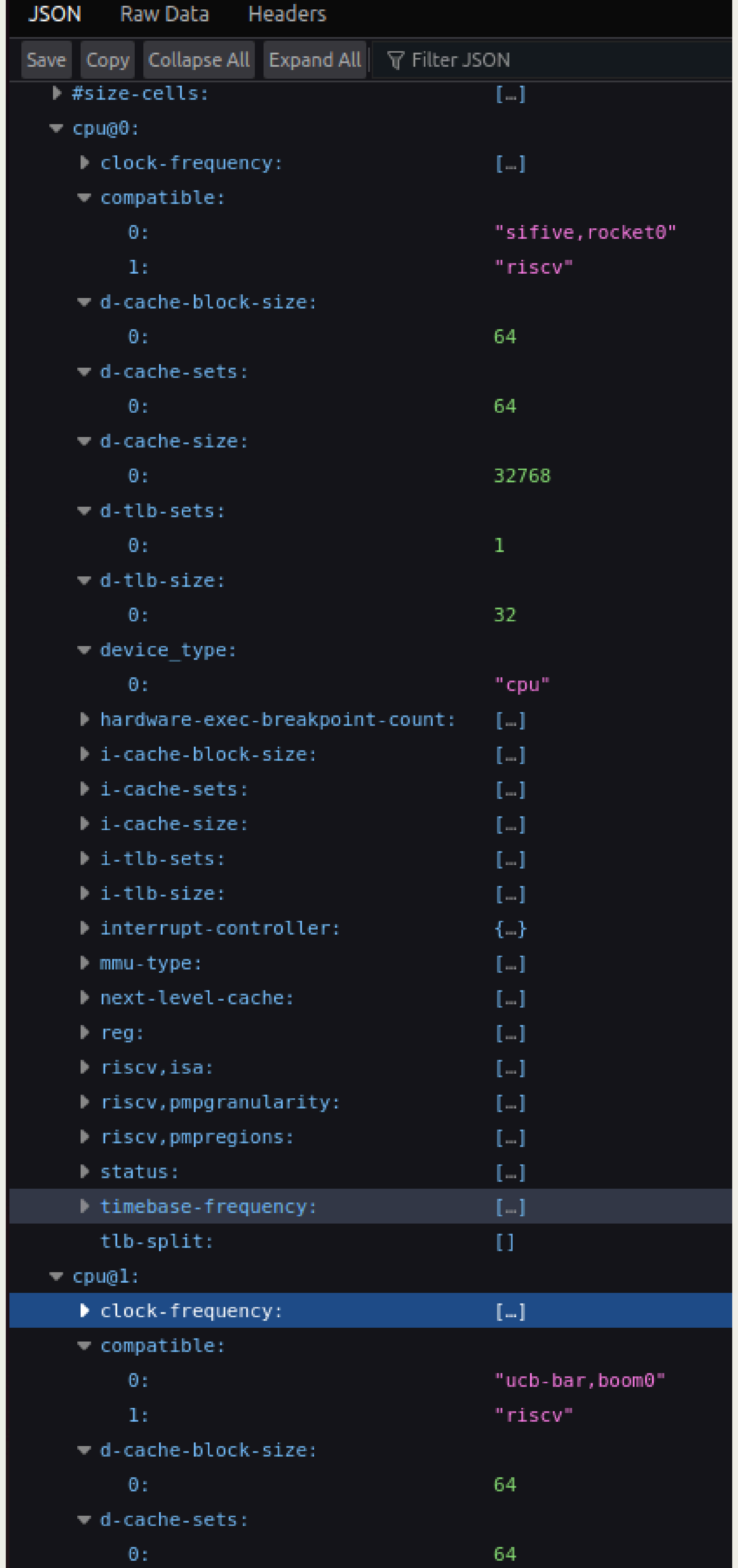
How it works...

Custom Config	*Define MyConfig.scala (BOOM + Rocket, bus width, caches)*
Generate JSON	*Chipyard build -> JSON capturing H/W params*
Parse in Python Script	*Produces Header file (.h) with the parameters*
Compile benchmarks	Compile our matric_mult.c and conv.c with RISC-V toolchain
Run	*Run on Spike/Verilator to measure cycle counts*

Extracting Config Data from JSON

- *We search the JSON for “cpus”, “cache sizes”, “bus widths”, etc*
- *Our script can adapt any fields we define in the Chipyard config*
- *Output macros, e.g:*

1. *TOTAL_CORES = sum of all CPU nodes*
2. *L1_DCACHE_SIZE from “cache-sets * cache-block-size * ways”*
3. *Additional data: system bus width, pipeline features etc*



```
  ▶ #size-cells:                                […]
  ▼ cpu@0:
    ▶ clock-frequency:                          […]
    ▼ compatible:
        0:                                      "sifive,rocket0"
        1:                                      "riscv"
    ▼ d-cache-block-size:
        0:                                      64
    ▼ d-cache-sets:
        0:                                      64
    ▼ d-cache-size:
        0:                                      32768
    ▼ d-tlb-sets:
        0:                                      1
    ▼ d-tlb-size:
        0:                                      32
    ▼ device_type:
        0:                                      "cpu"
    ▶ hardware-exec-breakpoint-count:           […]
    ▶ i-cache-block-size:                       […]
    ▶ i-cache-sets:                             […]
    ▶ i-cache-size:                             […]
    ▶ i-tlb-sets:                               […]
    ▶ i-tlb-size:                               […]
    ▶ interrupt-controller:                     {…}
    ▶ mmu-type:                                 […]
    ▶ next-level-cache:                         […]
    ▶ reg:                                      […]
    ▶ riscv,isa:                                […]
    ▶ riscv,pmpgranularity:                     […]
    ▶ riscv,pmpregions:                         […]
    ▶ status:                                   […]
    ▶ timebase-frequency:                       […]
      tlb-split:                                []
  ▼ cpu@1:
    ▶ clock-frequency:                          […]
    ▼ compatible:
        0:                                      "ucb-bar,boom0"
        1:                                      "riscv"
    ▼ d-cache-block-size:
        0:                                      64
    ▼ d-cache-sets:
        0:                                      64
```

Config-Driven Loop Transformations

- *We define macros in code:*

1. *#define TILE_SIZE (...)*
2. *#define UNROLL_FACTOR (...)*

- *These macros compute from TOTAL_CORES and L1_DCACHE_SIZE (or other parameters)*
- *Loop tiling, unrolling, merging, etc. become "plug-and-play"*

Applying Tiling + Unrolling

- *Tiling:*

1. *Outer loops break the matrix dimension M into sub-blocks*

- *Unrolling:*

2. *Inner loop factor chosen from TOTAL_CORES or other performance heuristics*

- *If SoC has bigger caches → bigger tile blocks*
- *If SoC has more cores → higher unroll factor*

2D Tiling + Partial Kernel Unroll

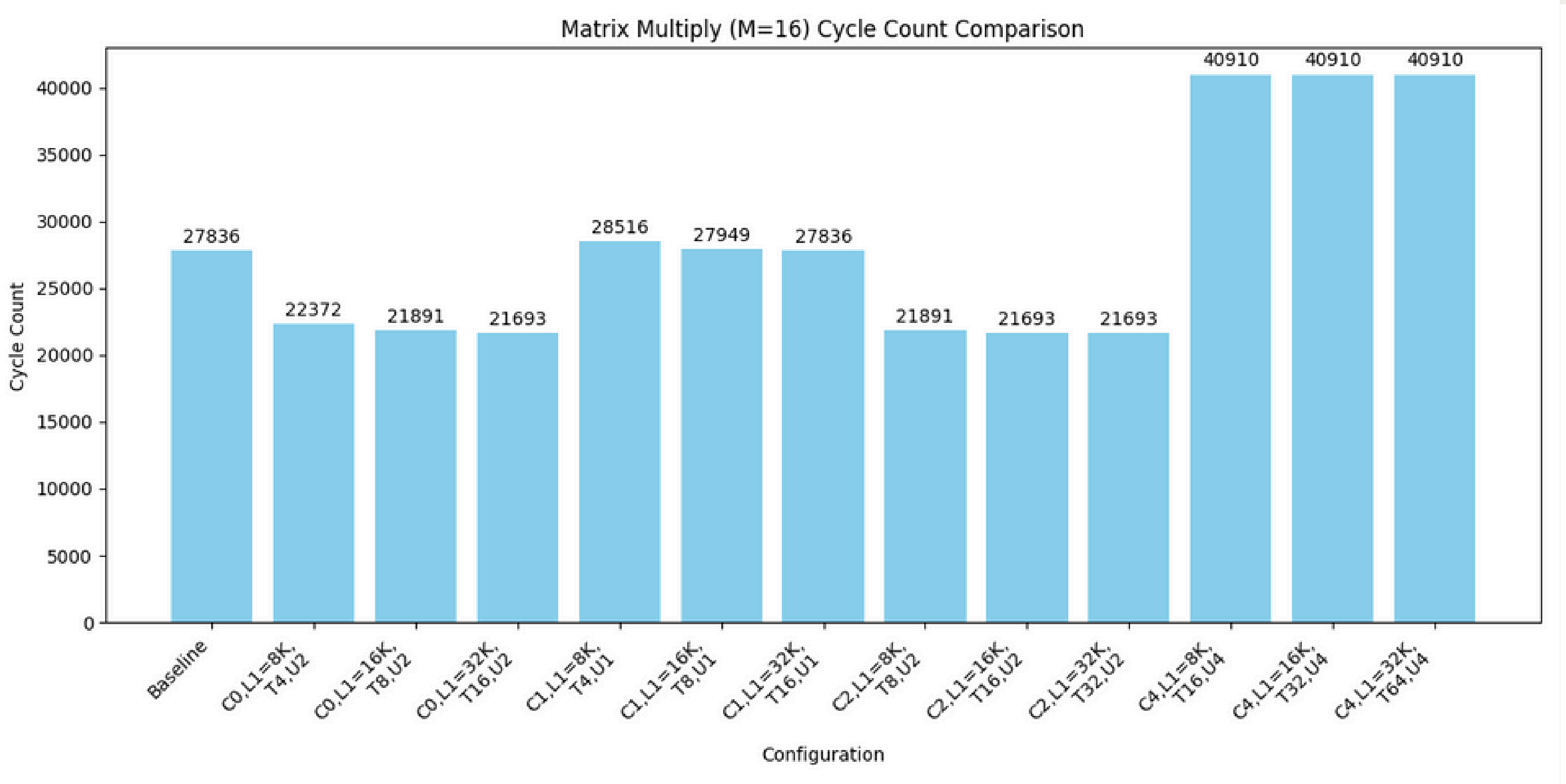
- *For an N×N input, if the SoC has bigger L1, we do bigger tile chunking over (i, j)*
- *If TOTAL_CORES is large, we might unroll the kernel loops more aggressively*
- *Code picks these thresholds from macros; easy to extend for new SoCs*

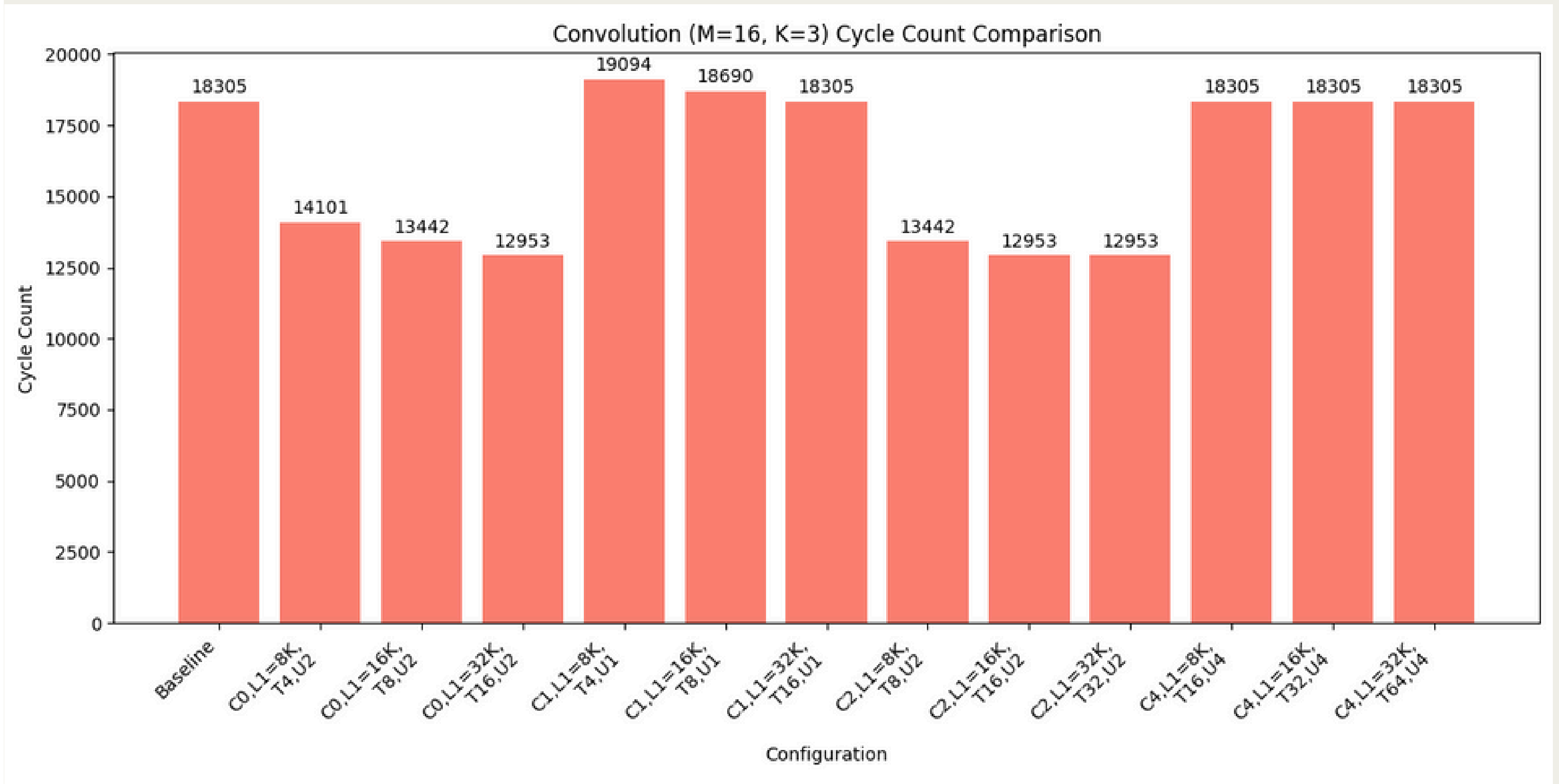
Collecting Cycle Counts on Arbitrary SoCs

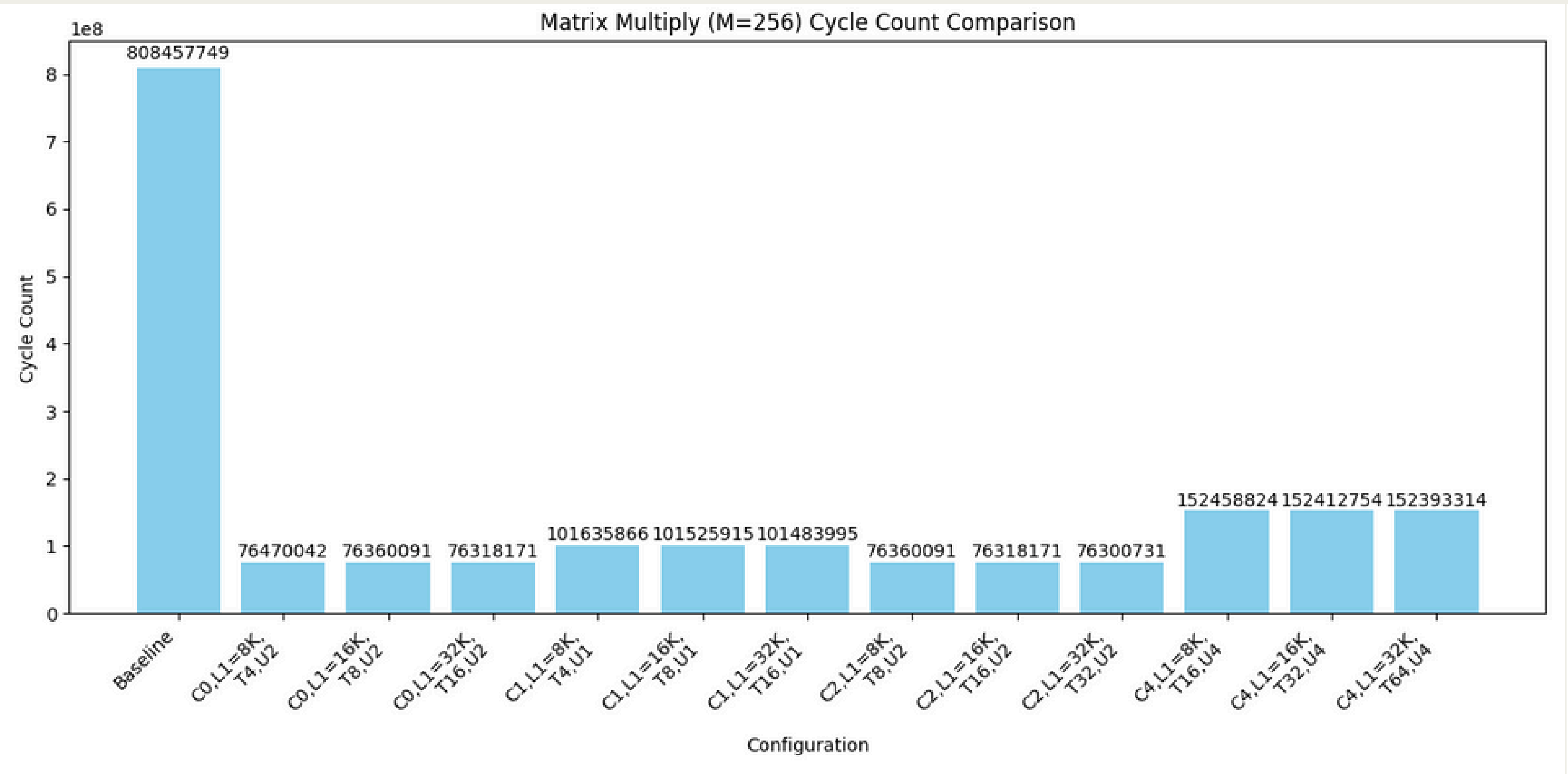
- *Each new Chipyard config → Rebuild the SoC + produce new JSON*
- *Our script regenerates config_params.h*
- *Recompile benchmarks → run on spike/verilator/FPGA*
- *Record cycle counts or other metrics (like instructions, energy, etc.)*

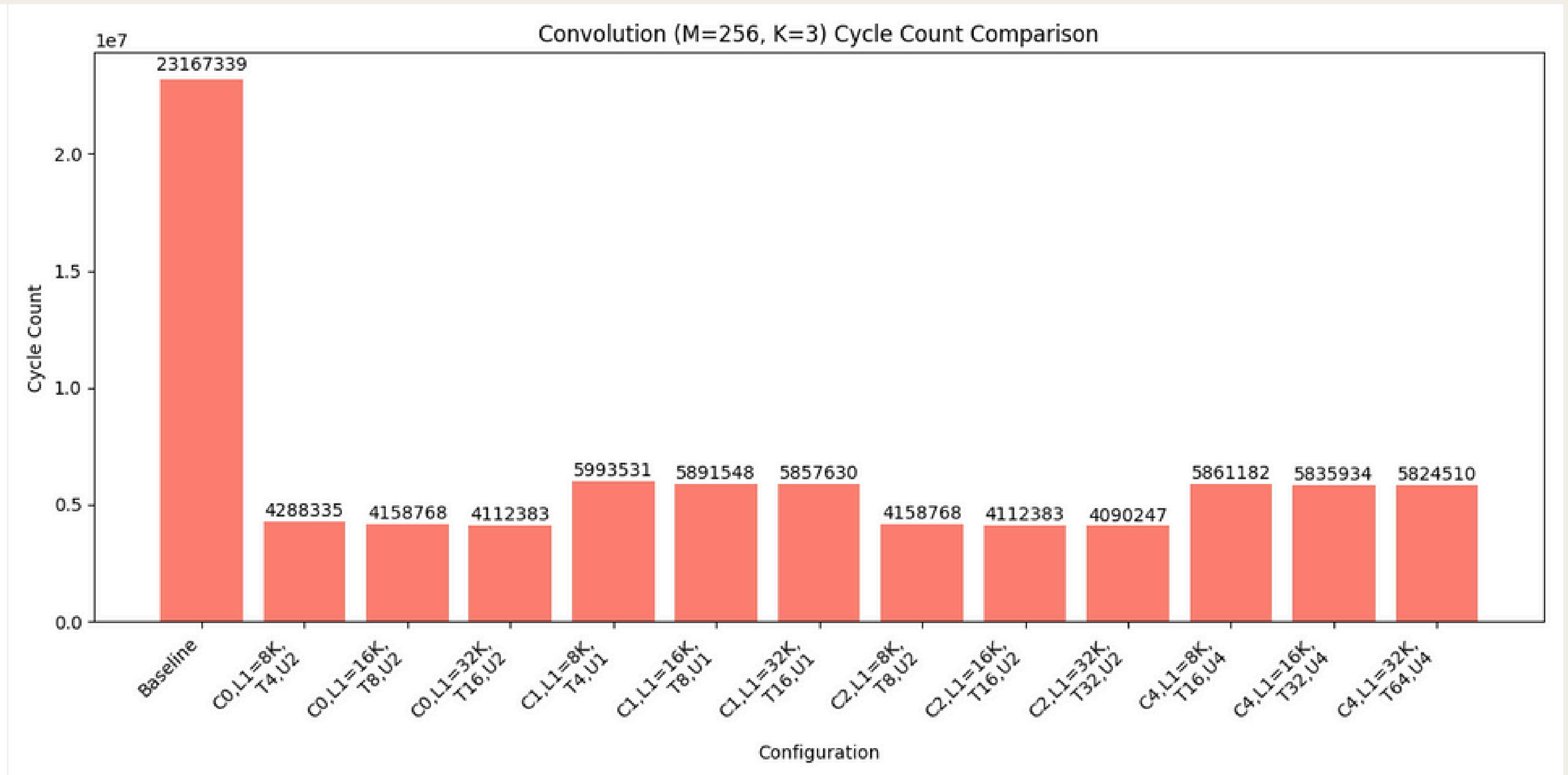
Plotting the Results

- *Bar plots / line charts vs. matrix size, kernel size, tile/unroll combos*
- *Identify best transformations for each SoC design*









CHIPYARD

Key Takeaways (Generalized)

- *Loop transformations remain crucial for large problem sizes and varied SoC designs*
- *Cache + core combos drive tiling/unrolling decisions*
- *Gains are currently single-threaded — true multicore parallelization is a next step*
- *Auto-tuning loop parameters or HPC frameworks is possible*
- *Any SoC config integrated with minimal changes to the pipeline*

Wrapping Up: A Universal Approach

- *We built an open pipeline where any Chipyard config → auto param → optimized code*
- *Extensible to more loops or HPC libraries*
- *Next: multi-thread concurrency, advanced auto-tuning, or hardware-specific instructions*