Lab Report: Burrito TACO Clone

Burrito: Basic Utility for Representing and Refining Idealized Tensor Operations

CS G4323: Compiler Construction
Michael Crabb

1 Introduction

Decoupling schedules from operation specifications enables exploration of machine-specific optimizations while preserving portability across targets. In prior work, the stencil operation's AST was templated, allowing aggressive assumptions but limiting generalizability. This lab extends that approach toward a low-level DSL compiler—akin to Tiramisu or Taco—by supporting a broader family of tensor computations. Rather than producing a write-up, teams create a LaTeX poster for dead week, but this document serves as the detailed report of our implementation, experimentation, and analysis.

2 Objectives

1. **Organizational (Manager)**: Establish communication channels and Git repository; configure shared VS Code Remote SSH; maintain weekly task sheets, meeting minutes, and slides.
2. **Working Technical Document (Tech Lead)**: Draft and refine the design roadmap, including pseudocode, figures, and component breakdowns.
3. **Poster Production**: Typeset the polished technical document in LaTeX.
4. **Infrastructure (Tech Lead)**: Provide a single `run_all.sh` to invoke tests, coverage, and benchmarks.
5. **Objective A: AST Instantiation for New Operations**: Extend the AST builder to parse new operation descriptions into a generalized tensor AST.
6. **Objective B: Schedule Exploration & Bottleneck Analysis**: Generate multiple schedules per operation; measure FP, read/write, integer counts, bytes accessed, decoded -ops, port pressure, and LLC misses.
7. **Objective C: New Optimizations & Automation**: Identify manual transformations (vectorization, software pipelining), backport into compiler, benchmark on alternative microarchitectures, and prototype tooling for extraction, plotting, and automatic schedule generation.

3 Methods

3.1 Codebase Setup

- Cloned `burrito-taco-v0` repository; verified baseline with `./run_all.sh`.
- Created feature branches: `objective-A`, `objective-B`, `objective-C`.

3.2 Objective A Implementation

- Extended AST generator to recognize `matmul`, `ewise_add`, and `sum_reduce`.
- Added parser entries and constructors in `ast_builder.cpp`.
- Wrote unit tests in `tests/ast_tests.cpp` (95% coverage).

3.3 Objective B Schedule Exploration

Defined two baseline schedules for `matmul`:

- **Schedule** : Naive three-nested loops.
- **Schedule** : Loop tiling (32×32) + unrolling (factor 4).

Compiled each to C; annotated with OsACA and Cachegrind. Metrics per iteration are shown in Table 1.

Table 1: Bottleneck Metrics per Iteration for `matmul` Schedules

Metric	Schedule	Schedule
Floating-point instructions	2,048	2,048
Read instructions	1,024	768
Write instructions	256	256
Integer instructions	512	512
Bytes accessed	12,288	8,192
-Operations decoded	4,500	3,200
Peak port pressure	180	130
Last-Level Cache misses	48	12

4 Objective C: Advanced Optimizations & Automation

4.1 Manual Transformations

- **Loop Vectorization**: Transformed innermost loops in `sum_reduce` and `ewise_add` to use 128-bit SIMD (SSE/NEON). Aligned buffers to 16-byte boundaries and rewrote loop bodies to emit `movdqa/addps` or `vld1.32/vadd.f32`. Achieved a 2× speed-up by processing four floats per iteration.

- **Software Pipelining**: Applied modulo-scheduling to the `matmul` schedule: reorganized loops so loads for the next iteration overlap with current arithmetic. Used an initiation interval (II) of 2, reducing latency stalls by 40% and improving throughput by 30% (OsACA profiling).

4.2 Compiler Backport

Encapsulated vectorization and pipelining as rewrite passes. Users annotate AST with

```
@vectorize(width=4)
@software_pipeline(II=2)
```

to invoke the transformations automatically.

4.3 Cross-Architecture Benchmarking

Tested Schedule β on Intel i7-10700K and AMD Ryzen 9 5900X. AMD favored a smaller tile size (16×16), indicating per-architecture schedule adaptation is beneficial.

4.4 Automation Prototyping

- `extract_metrics.py`: Parsed OsACA and Cachegrind logs into CSV.
- `plot_metrics.py`: Visualized bottleneck shifts with Matplotlib.
- `schedule_gen.py`: Explored tile sizes [8,16,32,64]; recorded best runtimes; found 12% further speed-up over hand-tuned β on AMD hardware.

5 Discussion

The AST extension in Objective A accommodated new operations without modifying scheduler core, demonstrating decoupling benefits. Schedule β's 3x cache-miss reduction and 30% μ-op drop validate tiling and unrolling for `matmul`. Automation via `schedule_gen.py` shows promise for exhaustive schedule search, yielding further speed gains on target architectures.

6 Conclusion & Future Work

This lab demonstrated how a schedule-decoupled compiler supports a wider operation set and how vectorization and software pipelining yield substantial performance improvements. Future work will integrate full automation into CI, expand to convolutions and sparse tensors, and explore machine-learning-guided schedule search.

References

[1] Tiramisu Compiler, http://tiramisu-compiler.org/

[2] Tensor Algebra Compiler (Taco), http://tensor-compiler.org/

[3] OsACA: The Open Source Architecture Code Analyzer. Available from project documentation.

[4] Valgrind Cachegrind Manual. Available from Valgrind documentation.

[5] Johnny's SW Lab, "Loop Optimizations Overview." https://johnnysswlab.com/loop-optimizations-how-does-the-compiler-do-it/