Optimization of SIMD Accelerator for Matrix Multiplication

Luke Qiao and Aarav Wattal

Hardware Reuse

Description: In the original processing element implementation, we had separate multiplication operations for each data mode (INT8, INT16, INT32). The naive approach would instantiate 4 separate 8-bit multipliers, 2 separate 16-bit multipliers, and 1 separate 32-bit multiplier. Since only one mode is active at any given time, this results in significant hardware waste.

Our optimized implementation consolidates multiplier hardware by using four shared 32-bit multipliers (`mult0` through `mult3`) that are dynamically configured based on the operating mode. In INT32 mode, only `mult0` is used for the full 32-bit×32-bit multiplication. In INT16 mode, `mult0` and `mult1` handle the two 16-bit×16-bit operations with sign extension. In INT8 mode, all four multipliers process the four 8-bit×8-bit operations in parallel.

The multiplier operands are selected through a combinational block that sign-extends the smaller operands to 32 bits before multiplication, for example:

```
mult0_a = {{24{vec8_0[7]}}, vec8_0}; // sign-extend to 32-bit
mult0_b = {{24{mat8_0[7]}}, mat8_0};
```

FoM Change: This optimization reduced the average PE cell area from approximately 11,500.9 μm^2 to 10,202.4 μm^2, representing a 11.29% reduction in area, leading to a decrease in total chip area and power. As a result, the overall FoM improved by approximately **13.56%**.

Reflection: These optimizations demonstrated that matching multiplier hardware to actual needs instead of over allocating resources result in large reductions in area. This makes sense, as unnecessary multiplier hardware is removed in the optimization.

Register Re-Timing

Description: We enabled automatic register retiming in Design Compiler by adding the `-retime` flag to the `compile_ultra` command in our synthesis script:

```
compile_ultra -retime -gate_clock -timing_high_effort_script
```

Register retiming allows the synthesis tool to automatically move registers across combinational logic to balance the delays between pipeline stages. This is particularly useful when the critical path has uneven logic depth across different stages.

FoM Change: Register retiming contributed to achieving timing closure at a 2.0 ns clock period (500 MHz). The synthesis log shows the retiming process moving registers to balance paths, stating lower bound estimate and critical path length during the process. As a result, the overall FoM improved by approximately **5.28%**.

Reflection: Register retiming is powerful because it leverages the synthesis tool's global view of timing to make optimizations that would be tedious to perform manually. The tool can identify where pipeline stages are unbalanced and automatically relocate registers without changing functionality. However, the synthesis log showed warnings about pipeline detection being aborted due to feedback loops in the accumulator registers, which limited the full effectiveness of this optimization. To address this, we implemented manual pipelining of the multiplier results to break the critical paths that register retiming couldn't automatically optimize due to the feedback constraints. This hybrid approach of letting the tool handle what it could while manually intervening on the problematic paths was effective for achieving timing closure.

Timing Optimization

Description: We implemented manual pipelining in the processing element to break up the long combinational paths through the multipliers and accumulators. The critical path in our design ran from the input operands through the multiplier, into the adder for accumulation, and finally to the accumulator register. This path was too long to meet our target clock period.

Our solution was to insert pipeline registers between the multiplication stage and the accumulation stage, creating a two-stage pipeline. The first stage performs the multiplication and registers the results. The second stage takes these registered multiplication results and performs the accumulation. We also pipelined the instruction decode signals (such as the MAC enable and mode selection bits) to ensure they arrive at the accumulation stage synchronized with the pipelined multiplier results.

An important consideration was that not all instructions require the multiplier pipeline. Instructions like PASS, CLR, RND, and OUT have much shorter combinational paths since they don't use the multipliers. For these instructions, we implemented a bypass path that skips the multiplier pipeline stage, avoiding unnecessary latency for simple operations.

We see that the final synthesized design has no timing errors, as required. This

is shown in the `Timing Path Group 'clk'` subsection of the timing report in `compile.log`.

```
...
Critical Path Slack:           0.00
Critical Path Clk Period:      2.00
Total Negative Slack:          0.00
No. of Violating Paths:        0.00
Worst Hold Violation:          0.00
Total Hold Violation:          0.00
No. of Hold Violations:        0.00
```

FoM Change: This optimization was critical for meeting timing at the target clock period. The final timing report shows zero slack with a critical path length of 1.96 ns against a 2.0 ns clock period. The timing optimization enabled a **31.55%** improvement in clock frequency compared to the unpipelined design.

Reflection: Manual pipelining required careful analysis of which instruction paths needed additional pipeline stages and which could bypass them. The key insight was that MAC operations dominate execution time and have the longest combinational paths, while simpler operations have much shorter paths. By selectively pipelining only the MAC path, we achieved timing closure without adding unnecessary latency to other operations. The synthesis timing report was invaluable for identifying the exact critical paths that needed attention.

Clock Gating

Description: We enabled automatic clock gating in Design Compiler by configuring the clock gating style and adding the `-gate_clock` flag:

```
set_clock_gating_style -sequential_cell latch \
    -positive_edge_logic {integrated} \
    -negative_edge_logic {or} \
    -control_point before \
    -control_signal scan_enable
set_clock_gating_check -setup 0.5 -hold 0.1
compile_ultra -retime -gate_clock -timing_high_effort_script
```

Clock gating inserts enable logic before registers to prevent unnecessary clock transitions when the registers don't need to be updated. The synthesis log confirms clock gating was applied to all 16 processing elements:

```
Information: Performing clock-gating on design \
    processing_element_0. (PWR-730)
Information: Performing clock-gating on design \
    processing_element_1. (PWR-730)
...
```

The tool inserted integrated clock-gating cells (`CLKGATETST_X1`) for the accumulator registers in each PE.

FoM Change: Clock gating reduced the power from approximately 32.69 mW to 8.77 mW. As a result, the overall FoM improved by approximately **73.17%**. Note that we define power as $P = P_{\text{switching}} + P_{\text{leakage}}$.

Reflection: Clock gating is particularly effective for our SIMD design because many processing elements may be idle during certain operations. The accumulator registers only need to update during MAC, PASS, RND, and CLR operations, so clock gating prevents wasteful switching during NOP cycles and memory operations. The synthesis tool automatically identified enable conditions from our RTL and inserted appropriate gating logic.

Design Sizing

Description: We configured our design with 16 processing elements by setting the `--pe_count 16` parameter in the build flow. This choice represents a balance between parallelism and area constraints. We can calculate from the `compile.log` synthesis report that

```
Total cell area:   164985.969242 um^2
Processing elements account for ~98% of total area
```

Each processing element contributes approximately 10,202.4 μm^2 on average to the total area. With 16 PEs, we remain well under the 1,000,000 μm^2 maximum while providing sufficient parallelism for the matrix multiplication workloads.

FoM: For this project, our FoM formula weights execution time with an exponent of 2.3, making parallelism highly valuable. With 16 PEs operating at 500 MHz:

- INT8 test cycles: $\sim$ 16503 cycles $\rightarrow$ 33.01 μs
- INT16 test cycles: $\sim$ 8311 cycles $\rightarrow$ 16.62 μs
- INT32 test cycles: $\sim$ 4215 cycles $\rightarrow$ 8.43 μs

Reflection: Design sizing required understanding the FoM formula's sensitivity to different parameters. Since time is raised to the power of 2.3, reducing execution time has a disproportionately large impact on FoM compared to linear reductions in area or power. However, adding more PEs has diminishing returns because area increases linearly while time reduction depends on how well the workload parallelizes. We found that 16 PEs is a sweet spot where we achieve good parallelization of the test workloads without excessive area overhead. Future optimization could explore heterogeneous PE configurations or dynamically configurable PE counts.

Final FoM

Our final FoM is

$$\begin{aligned}\text{FoM} &= (33.01\,\mu\text{s} + 16.62\,\mu\text{s} + 8.43\,\mu\text{s})^{2.3}(164985.9\,\mu\text{m}^2)(8.77\,\text{mW}) \cdot 10^{-15} \\ &\approx \mathbf{131.03},\end{aligned}$$

which is a significant improvement from our initial FoM of 871.35.