Final Report: Fused Multiply-Add (FMA) Optimization Pass for LLVM

(CPU – ARM64, Apple M1)

Samin Islam

December 12, 2025

1 Introduction

Modern processors speed up math using Fused Multiply-Add (FMA), which combines multiplication and addition ($a \times b + c$) into a single, more accurate step. This project implements a custom LLVM compiler pass that automatically finds these separate operations in the code and fuses them into efficient FMA instructions. The report details how the pass detects candidates, verifies safety, and transforms the code to improve performance.

2 Methodology

The FMA Optimization is implemented as a transformation pass within the LLVM infrastructure. The pass, defined as `FMAContractPass`, operates at the function level. It systematically scans the Intermediate Representation (IR) to identify opportunities where separate floating-point multiplication and addition/subtraction instructions can be contracted into a single hardware-accelerated Fused Multiply-Add (FMA) intrinsic.

The implementation follows a four-stage pipeline: Pattern Detection, Feasibility Analysis, Compile-Time Evaluation, and IR Transformation.

2.1 Pattern Detection and Semantic Checks

The pass safely iterates through the function to identify FAdd and FSub instructions as potential FMA roots. It strictly checks for the `allow_contract` flag [1], aborting fusion if absent to preserve strict IEEE rounding. Once a root is validated, the logic locates the corresponding FMul and automatically flags operands for negation to handle subtraction cases correctly.

```
// Logic to handle: C - (A*B) -> - (A*B) + C
if (Opcode == Instruction::FSub) {
    if (MulOp == Op1) { // If the multiplier is the
        subtrahend
        NegateMul = true;
    } else { // If the multiplier is the minuend: (A*B) - C
        NegateAddend = true;
    }
}
```

Listing 1: Subtraction Handling Logic

2.2 Feasibility Analysis

Before modifying the code, the pass determines if the transformation is beneficial.

1. **Hardware Cost Model:** The pass queries LLVM's `TargetTransformInfo (TTI)`. It calculates the cost of the `llvm.fma` intrinsic for the specific target architecture. If the cost is maximal (indicating the hardware would emulate the instruction in software), the optimization is skipped to prevent performance regression.

2. **Multi-Use Validity:** Unlike basic implementations, this pass supports fusing multiplication instructions that are used by multiple consumers. The pass proceeds with fusion even if the `FMul` has other users, ensuring maximum coverage.

2.3 Compile-Time Constant Folding

To optimize performance, the pass attempts to calculate the result at compile time. The goal is to eliminate runtime execution entirely: if all inputs are known constants, generating an FMA instruction is unnecessary overhead. Instead, the compiler computes the answer immediately and replaces the entire sequence with a single value. This reduces the instruction count and simplifies the code structure for future optimization passes.

Using LLVM's `APFloat` engine, the result is computed immediately using infinite precision for the intermediate step (mimicking hardware FMA behavior) and rounded once [1].

```
if (auto *CA = dyn_cast<ConstantFP>(MulOp->getOperand(0))) {
    // ... check other operands ...
    APFloat ValA = CA->getValueAPF();

    // Perform calculation in compiler memory
    ValA.fusedMultiplyAdd(ValB, ValC, APFloat::
        rmNearestTiesToEven);

    // Replace instruction with calculated constant
    Value *ConstRes = ConstantFP::get(F.getContext(), ValA);
    BinOp->replaceAllUsesWith(ConstRes);
    // ...
}
```

Listing 2: Static Evaluation using APFloat

2.4 IR Transformation and Cleanup

If static evaluation is not possible, the pass generates the runtime IR. An `IRBuilder` is initialized with the fast-math flags of the original instruction.

1. **Negation Insertion:** If the detection phase flagged ‘NegateMul‘ or ‘NegateAddend‘, `FNeg` instructions are inserted to adjust the signs of the operands.

2. **Intrinsic Call:** An `llvm.fma` intrinsic call is created and inserted.

3. **Graph Cleanup:** The original addition/subtraction instruction is removed. The source multiplication instruction is removed *conditionally*: it is deleted only if it has no remaining users (`use_empty()`). This guarantees that the IR remains valid for other instructions that may still depend on the original multiplication result.

```
// Always remove the Add/Sub because we replaced it
BinOp->eraseFromParent();

// Only remove the Mul if it has no other users (Multi-Use
    Support)
if (MulOp->use_empty()) {
    MulOp->eraseFromParent();
}
```

Listing 3: Conditional Cleanup

3 Results and Evaluation

3.1 Functional Verification

We verified the correctness of the pass using a custom micro-benchmark suite. As shown in our test logs, the pass successfully identified valid FMA patterns while respecting safety constraints:

- **Correctness:** Valid patterns (e.g., `base.c`, `negate.c`, `multiUse.c`) were successfully transformed into `@llvm.fma` intrinsics.

- **Safety:** The `noContract.c` test confirmed that the pass correctly ignores code when the user forbids contraction, preserving strict IEEE semantics.

- **Constant Folding:** The `constFold.c` test verified that constant operands are calculated at compile-time, removing instructions entirely.

3.2 Performance Benchmarking

We evaluated performance on an Apple M1 using the PolyBench/C suite. We compared our FMA-optimized code against a strict-IEEE baseline.

Kernel	**Baseline (s)**	**Optimized (s)**	**Speedup**
GEMM	0.260	0.261	**1.00x**
BICG	0.017	0.017	**0.99x**
2MM	1.644	2.215	**0.74x**

Table 1: Performance Results on Apple M1.

3.3 Analysis of Results

- **Memory Bound (GEMM - 1.00x):** There was no performance change because the bottleneck was memory, not math. The processor spent most of its time waiting for data, which masked the speed benefits of the fused instructions.
- **Latency Issues (2MM - 0.74x):** Performance actually dropped due to instruction latency. FMA operations can take longer to complete than simple additions. In this specific workload, the CPU pipeline frequently stalled waiting for results, canceling out the throughput gains.

  While the optimization is functionally correct, its real-world benefit depends on the workload. In these specific tests, memory limits and instruction dependencies prevented any speedup on the M1 chip.

4 Future Work

In future, I plan to add support for SIMD vectors in this pass plugin. The current pass only handles single values (scalars). Extending this to vectors would ensure the optimization works even if the code has already been vectorized by previous passes. Morevover, combine this pass with loop unrolling can create a larger window of visible instructions, which would give the pass more opportunities to find and fuse operations. Lastly, I do plan to use a custom PAPI [2] library to define specialized metrics. This will allow me to precisely track specific floating-point behaviors and verify the optimization's impact.

References

[1] "Ieee standard for floating-point arithmetic," *IEEE Std 754-2019 (Revision of IEEE 754-2008)*, pp. 1–84, 2019.

[2] A. Danalis, H. Jagode, T. Herault, P. Luszczek, and J. Dongarra, "Software-defined Events through PAPI," in *2019 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*. IEEE, 2019, pp. 363–372.