Fused Multiply-Add Optimization Pass

Francisco F Cavazos Jr

Motivation

- Floating point intensive workloads often use multiply + add sequences
- FMA instruction: combines A * B + C into a single instruction
- Benefits:
  - Higher Throughput
  - Reduced rounding error

What is FMA?

The operation fusedMultiplyAdd(x, y, z) computes $(x \times y) + z$ as if with unbounded range and precision, rounding only once to the destination format.

– IEEE-754-2008 (Definition 2.1.28)

Regular

```
%0 = fmul float %x, %y
%1 = fadd float %0, %z
ret float %1

= round(round(x * y) + z)
```

FMA

```
%0 = call float @llvm.fma.f32(float %x, float %y, float %z)
ret float %0


= round( x * y + z)
```

What is FMA?

```c
#include <stdio.h>
#include <math.h>


float nofma(float x, float y, float z) {
    return x * y + z;
}

float withfma(float x, float y, float z) {
    return fmaf(x, y, z);
}

int main(void) {
    float x = 1e8f;
    float y = 1.0000001f;
    float z = -1e8f;

    printf("No FMA: %.10f\n", nofma(x, y, z));
    printf("   FMA: %.10f\n", withfma(x, y, z));

    return 0;
}
```

```llvm
@.str = private unnamed_addr constant [15 x i8] c"No FMA: %.10f\0A\00", align 1
@.str.1 = private unnamed_addr constant [14 x i8] c"   FMA: %.10f\00", align 1

; Function Attrs: noinline nounwind ssp uwtable(sync)
define float @nofma(float noundef %0, float noundef %1, float noundef %2) #0 {
  %4 = fmul float %0, %1
  %5 = fadd float %4, %2
  ret float %5
}

; Function Attrs: noinline nounwind ssp uwtable(sync)
define float @withfma(float noundef %0, float noundef %1, float noundef %2) #0 {
  %4 = call float @llvm.fma.f32(float %0, float %1, float %2)
  ret float %4
}

; Function Attrs: nocallback nofree nosync nounwind speculatable willreturn memory(none)
declare float @llvm.fma.f32(float, float, float) #1

; Function Attrs: noinline nounwind ssp uwtable(sync)
define i32 @main() #0 {
  %1 = call float @nofma(float noundef 1.000000e+08, float noundef 0x3FF0000020000000, float noundef -1.000000e+08)
  %2 = fpext float %1 to double
  %3 = call i32 (ptr, ...) @printf(ptr noundef @.str, double noundef %2)
  %4 = call float @withfma(float noundef 1.000000e+08, float noundef 0x3FF0000020000000, float noundef -1.000000e+08)
  %5 = fpext float %4 to double
  %6 = call i32 (ptr, ...) @printf(ptr noundef @.str.1, double noundef %5)
  ret i32 0
}

declare i32 @printf(ptr noundef, ...) #2
```

What is FMA?

We’re computing:

$$1.0000001 \times (1 \times 10^8) - (1 \times 10^8) = 10$$

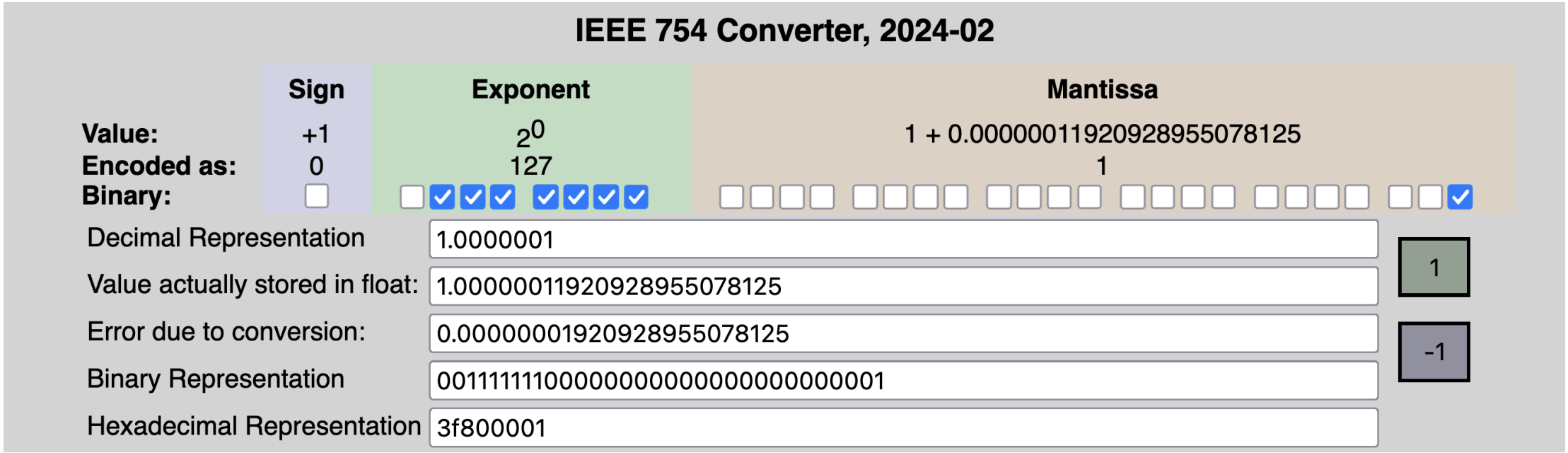
Results:

- No FMA = 8
- With FMA = 11.9209289551

```
Ferny tests % llc fma.ll -o fma.s; clang -o fma fma.s; ./fma
No FMA: 8.0000000000
   FMA: 11.9209289551
```

What is FMA?

Floating points aren’t an exact representation, just the closest estimate



What is FMA?

What we want to compute:

$$1.0000001 \times (1 \times 10^8) - (1 \times 10^8) = 10$$

What we actually compute:

$$1.00000011920928955 \times (1 \times 10^8) - (1 \times 10^8) = 11.9209289551$$

Results:

- No FMA = 8
- With FMA = 11.9209289551

Project Objective

- Implement a custom LLVM pass that:
  - Detects `fmul` + `fadd` patterns
  - Replaces them with LLVM’s FMA intrinsic
  - Checks hardware support
  - Ensures transformation is legal and profitable
- Evaluate across different tests

Creating the LLVM Pass

Pattern Detection

We want to detect the following pattern:

%mul = fmul double %x, %y
%add = fadd double %mul, %z

How?

1. Search the LLVM IR for all `fadd`
2. For each `fadd`, check if one of the operands opcode is `fmul`
3. If true, add the `fadd` + `fmul` combo to candidate list

Pattern Detection

We want to detect the following pattern:

%mul = fmul double %x, %y
%add = fadd double %mul, %z

```
bool matchesFMAPattern(BinaryOperator *FAdd) {
  Value *Op0 = FAdd->getOperand(0);
  Value *Op1 = FAdd->getOperand(1);

  // Check if either operand is an fmul instruction
  if (auto *FMul = dyn_cast<BinaryOperator>(Op0)) {
    if (FMul->getOpcode() == Instruction::FMul) {
      return true;
    }
  }

  if (auto *FMul = dyn_cast<BinaryOperator>(Op1)) {
    if (FMul->getOpcode() == Instruction::FMul) {
      return true;
    }
  }
  return false;
}
```

Pattern Detection

We can use the “llvm/IR/PatternMatch.h” header functions to improve it (using match)

```
1 bool matchesFMAPattern(BinaryOperator *FAdd) {
2   Value *X, *Y, *Z;
3
4   // Pattern 1: fadd (fmul X, Y), Z
5   if (match(FAdd, m_FAdd(m_FMul(m_Value(X), m_Value(Y)), m_Value(Z)))) {
6     return true;
7   }
8
9   // Pattern 2: fadd Z, (fmul X, Y) - commutative case
10  if (match(FAdd, m_FAdd(m_Value(Z), m_FMul(m_Value(X), m_Value(Y))))) {
11    return true;
12  }
13
14  return false;
15 }
```

Legality Conditions

- Test if FMA is supported on the target machine
- Implement a profitability check (only transform if beneficial, e.g., not in precise FP mode)

Legality Conditions

- Test if FMA is supported on the target machine

1. Get the intrinsic cost attributes for float and double FMA
2. Get the instruction cost for the FMA’s
3. Check if the cost is valid
4. If valid, FMA is supported. If not, FMA not supported

Legality Conditions

- Test if FMA is supported on the target machine

```
bool isFMASupported(TargetTransformInfo &TTI, Function &F) {
  LLVMContext &Ctx = F.getContext();

  // Check for both float (f32) and double (f64) FMA support
  Type *DoubleTy = Type::getDoubleTy(&C: Ctx);
  Type *FloatTy = Type::getFloatTy(&C: Ctx);

  // We check if the intrinsic is available and has reasonable cost
  IntrinsicCostAttributes ICADouble(Id: Intrinsic::fma, RTy: DoubleTy,
                                    Tys: {[0]=DoubleTy, [1]=DoubleTy, [2]=DoubleTy});
  IntrinsicCostAttributes ICAFloat(Id: Intrinsic::fma, RTy: FloatTy,
                                   Tys: {[0]=FloatTy, [1]=FloatTy, [2]=FloatTy});

  InstructionCost CostDouble =
      TTI.getIntrinsicInstrCost(ICA: ICADouble, CostKind: TargetTransformInfo::TCK_CodeSize);
  InstructionCost CostFloat =
      TTI.getIntrinsicInstrCost(ICA: ICAFloat, CostKind: TargetTransformInfo::TCK_CodeSize);

  // FMA is supported if the intrinsic has a valid, non-infinite cost
  // An invalid or very high cost indicates lack of hardware support
  bool DoubleSupported = CostDouble.isValid() &&
                         CostDouble != InstructionCost::getInvalid() &&
                         CostDouble.getValue() == 1;
  bool FloatSupported = CostFloat.isValid() &&
                        CostFloat != InstructionCost::getInvalid() &&
                        CostFloat.getValue() == 1;

  // FMA is supported if either float or double version is supported
  bool Supported = DoubleSupported || FloatSupported;

  return Supported;
}
```

Legality Conditions

Implement a profitability check (only transform if beneficial, e.g., not in precise FP mode)

1. Check if fast-math flags are enabled
2. Check count of uses for FMUL
3. Check if in precise FP mode
4. Default case: don’t optimize

Legality Conditions

- Implement a profitability check

1. Check if fast-math flags are enabled
2. Check count of uses for FMUL
3. Check if in precise FP mode
4. Default case: don’t optimize

```
bool isProfitable(BinaryOperator *FAdd, BinaryOperator *FMul) {
  // Check 1: Fast math flags
  // FMA changes the rounding behavior, so we should only apply it if:
  // - Fast math flags allow it (contract or fast)
  // - Or if both instructions already allow reassociation
  FastMathFlags FAddFlags = FAdd->getFastMathFlags();
  FastMathFlags FMulFlags = FMul->getFastMathFlags();

  // Precise FP mode check
  // If we're in precise FP mode (no fast math flags at all), don't transform
  if (!FAddFlags.any() && !FMulFlags.any()) {
    return false;
  }

  // If either operation allows contraction, FMA is profitable
  if (FAddFlags.allowContract() || FMulFlags.allowContract()) {
    return true;
  }

  // If both operations are marked as fast math, FMA is profitable
  if (FAddFlags.isFast() && FMulFlags.isFast()) {
    return true;
  }

  // Check 2: Single use of multiply result
  // If the multiply result is only used once (by this add), combining is
  // beneficial This reduces register pressure and eliminates an intermediate
  // value
  if (FMul->hasOneUse()) {
    return true;
  }

  // Default: if we have some fast math flags but not contract/fast, be
  // conservative and don't transform
  return false;
}
```

Transformation Implementation

Transform all candidate `fadd` + `fmul` into FMA instructions

1. Extract operators and operands from pattern
2. Get appropriate FMA intrinsic type based on pattern
3. Create FMA intrinsic call using extracted operands
4. Preserve flags from original instruction in FMA call
5. Replace the `fadd` with the new FMA call
6. Remove dead `fmul` code

Transformation Implementation

Transform all candidate `fadd` + `fmul` into FMA instructions

1. Extract operators and operands from pattern

```
86    Value *X = nullptr, *Y = nullptr, *Z = nullptr;
87    BinaryOperator *FMul = nullptr;
88
89    // Extract the operands from the pattern
90    // Pattern 1: fadd (fmul X, Y), Z
91    if (match(V: FAdd, P: m_FAdd(L: m_FMul(L: m_Value(&V: X), R: m_Value(&V: Y)), R: m_Value(&V: Z)))) {
92      FMul = dyn_cast<BinaryOperator>(Val: FAdd->getOperand(i_nocapture: 0));
93    }
94    // Pattern 2: fadd Z, (fmul X, Y) -- commutative case
95    else if (match(V: FAdd, P: m_FAdd(L: m_Value(&V: Z), R: m_FMul(L: m_Value(&V: X), R: m_Value(&V: Y))))) {
96      FMul = dyn_cast<BinaryOperator>(Val: FAdd->getOperand(i_nocapture: 1));
97    }
98
99    // Failed to extract operands
100   if (!FMul || !X || !Y || !Z) {
101     return;
102   }
```

Transformation Implementation

Transform all candidate `fadd` + `fmul` into FMA instructions

1. Extract operators and operands from pattern
2. Get appropriate FMA intrinsic type based on pattern

```
104    // Get the type of the operation (float or double)
105    Type *Ty = FAdd->getType();
106
```

Transformation Implementation

Transform all candidate `fadd` + `fmul` into FMA instructions

1. Extract operators and operands from pattern
2. Get appropriate FMA intrinsic type based on pattern
3. Create FMA intrinsic call using extracted operands

```
110     // Get the FMA intrinsic for the appropriate type
111     Module *M = FAdd->getModule();
112     Function *FMAIntrinsic =
113         Intrinsic::getOrInsertDeclaration(M, id: Intrinsic::fma, Tys: Ty);
114
115     // Create the FMA intrinsic call: fma(X, Y, Z) computes X * Y + Z
116     Value *FMACall = Builder.CreateCall(Callee: FMAIntrinsic, Args: {[0]=X, [1]=Y, [2]=Z});
```

Transformation Implementation

Transform all candidate `fadd` + `fmul` into FMA instructions

1. Extract operators and operands from pattern
2. Get appropriate FMA intrinsic type based on pattern
3. Create FMA intrinsic call using extracted operands
4. Preserve flags from original instruction in FMA call

```
118     // Preserve fast math flags from the original instructions
119     if (auto *FMAInst: Instruction * = dyn_cast<Instruction>(Val: FMACall)) {
120         FastMathFlags FMF = FAdd->getFastMathFlags();
121         FMF |= FMul->getFastMathFlags(); // Combine flags from both operations
122         FMAInst->setFastMathFlags(FMF);
123     }
```

Transformation Implementation

Transform all candidate `fadd` + `fmul` into FMA instructions

1. Extract operators and operands from pattern
2. Get appropriate FMA intrinsic type based on pattern
3. Create FMA intrinsic call using extracted operands
4. Preserve flags from original instruction in FMA call
5. Replace the `fadd` with the new FMA call

```
125    ··//·Replace·all·uses·of·the·FAdd·with·the·FMA·call↓
126    ··FAdd->replaceAllUsesWith(V: FMACall);↓
```

Transformation Implementation

Transform all candidate `fadd` + `fmul` into FMA instructions

1. Extract operators and operands from pattern
2. Get appropriate FMA intrinsic type based on pattern
3. Create FMA intrinsic call using extracted operands
4. Preserve flags from original instruction in FMA call
5. Replace the `fadd` with the new FMA call
6. Remove dead code

```
128     // If the FMul has no other uses, we can remove it
129     if (FMul->hasOneUse())
130       FMul->eraseFromParent();
131
132     // Remove the original FAdd instruction
133     FAdd->eraseFromParent();
134   }
```

Testing

- Compiled with –O0

```
1 double test_basic_fma(double x, double y, double z) {
2   double mul_result = x * y;
3   return mul_result + z;
4 }
```

clang -emit-llvm
opt -passes=mem2reg

```
1 ; Function Attrs: noinline nounwind ssp uwtable(sync)
2 define double @test_basic_fma(double noundef %0, double noundef %1, double noundef %2)
3   %4 = fmul double %0, %1
4   %5 = fadd double %4, %2
5   ret double %5
6 }
```

opt -passes=fma

```
1 ; Function Attrs: noinline nounwind ssp uwtable(sync)
2 define double @test_basic_fma(double noundef %0, double noundef %1, double noundef %2)
3   %4 = call double @llvm.fma.f64(double %0, double %1, double %2)
4   ret double %4
5 }
```

Testing

- Compiled with –O0

```
1 float test_basic_fma_float(float x, float y, float z) {
2   float mul_result = x * y;
3   return mul_result + z;
4 }
```

clang -emit-llvm
opt -passes=mem2reg

```
1 ; Function Attrs: noinline nounwind ssp uwtable(sync)
2 define float @test_basic_fma_float(float noundef %0, float noundef %1, float noundef %2
3   %4 = fmul float %0, %1
4   %5 = fadd float %4, %2
5   ret float %5
6 }
```

opt -passes=fma

```
1 ; Function Attrs: noinline nounwind ssp uwtable(sync)
2 define float @test_basic_fma_float(float noundef %0, float noundef %1, float noundef %2
3   %4 = call float @llvm.fma.f32(float %0, float %1, float %2)
4   ret float %4
5 }
```

Next Steps

- Properly benchmark to see if there's performance improvements
- Test on different architectures (only have tested on my mac)
- Could add support for different FMA patterns (-x * y +z OR x * y – z)