Seamless Compiler Integration of Variable Precision Floating-Point Arithmetic

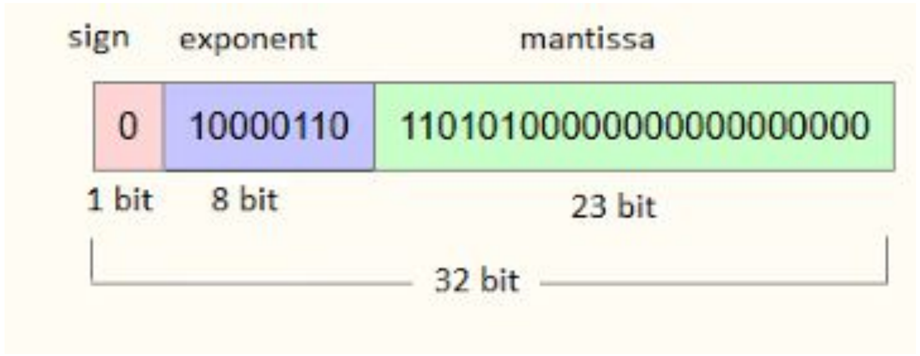
Group 17
Presented By: Bobby Palmer, Conner Rose, Hrishikesh Bagalkote, Shiv Govil, Jay Landers

Background

- Modern processor Floating-Point (FP) hardware is fixed and limited to standardized formats defined by IEEE 754 (e.g., FP32 and FP64).

- Many scientific/HPC applications require precision beyond 64 bits to ensure numerical stability and correct results.
  - Some algorithms (e.g., linear solvers, n-body problems, certain particle simulations) often require temporarily higher precision to maintain accuracy or can achieve faster convergence by adjusting precision dynamically.

- The fixed-size design prioritizes speed and efficiency (as a result of standardization) but doesn't fit the dynamic needs of many modern scientific computational workloads.



Background

Libraries like MPFR treat every arithmetic operation as a black-box function call, resulting in:

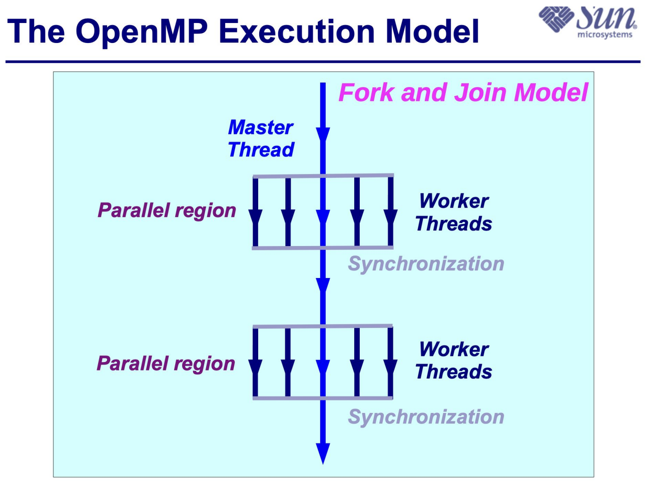
- Tedious, manual memory allocation and deallocation.
- Compiler optimization (e.g., loop unrolling) is impossible since the compiler cannot see the math inside the function call.

Goal is to bring the efficiency and optimization of C-style compilation to high-level, variable-precision floating-point types and bridge the **Productivity-Performance Gap.**

Approach	**Developer Productivity**	**Performance (Throughput)**	**Drawback**
High-Level Abstractions (e.g. Julia)	High	Low	Slow execution due to runtime overhead and lack of access to low-level compiler optimizations.
Software Libraries (e.g. MPFR)	Low	High (Theoretically)	Requires manual memory management, leads to code bloat, and provides opaque function calls that block aggressive compiler optimizations.

Research Goals

- Provide natural C bindings (vpfloat) using standard operators, eliminating the manual memory management and "code bloat" required by libraries like MPFR.
- Apply Intermediate Representation (IR)-level optimizations to high-precision, variable-length FP operations. Done by translating the new types into generic LLVM IR intrinsics to expose the code to aggressive tools like LLVM's Polly (Loop Nest Optimizer).
- Ensure seamless integration with existing standards like OpenMP to efficiently scale high-precision code across multi-core systems.
- Create a system where a single codebase can target software libraries (MPFR) today and future variable-precision hardware ISA (like UNUM) tomorrow, simply by changing the compiler backend.



OpenMP, MPFR, and UNUM

OpenMP (Open Multi-Processing)

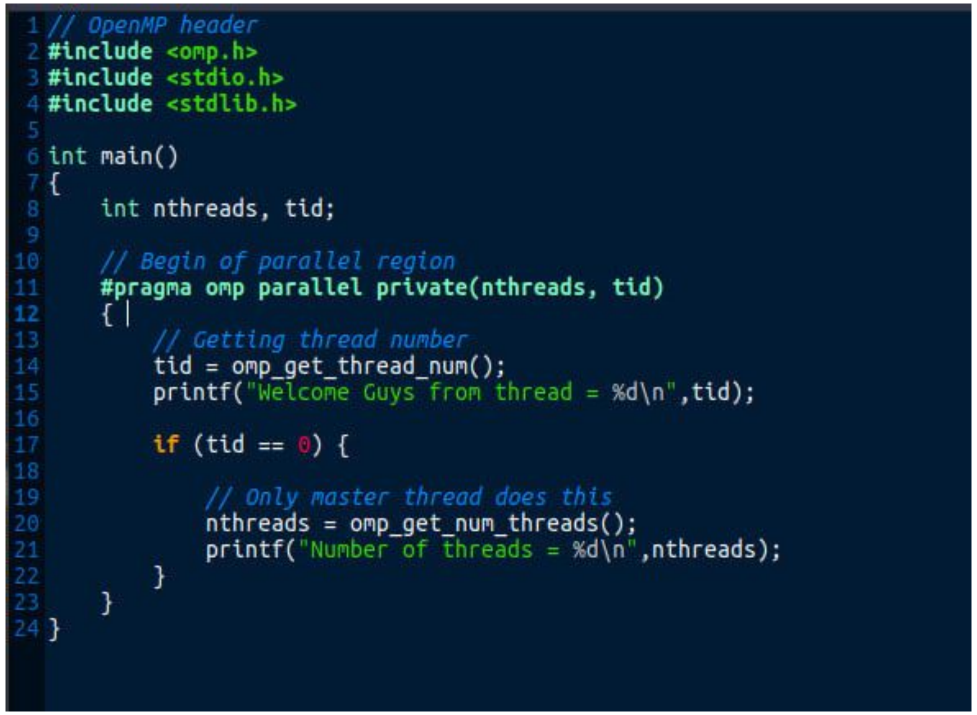
- Helper for generating optimized parallelism
- `#pragma omp parallel`

MPFR (Multiple Precision Floating-Point Reliable Library)

- The gold standard for fast, high-precision math in software.
- Harder to optimize since it is opaque to the compiler
- Inconvenient / requires specific library knowledge

UNUM (Universal Number)

- An alternative floating point format (ie. IEEE 754) with arbitrary precision
- Providing an IR representation of the types in MPFR



```
// OpenMP header
#include <omp.h>
#include <stdio.h>
#include <stdlib.h>

int main()
{
    int nthreads, tid;

    // Begin of parallel region
    #pragma omp parallel private(nthreads, tid)
    {
        // Getting thread number
        tid = omp_get_thread_num();
        printf("Welcome Guys from thread = %d\n",tid);

        if (tid == 0) {

            // Only master thread does this
            nthreads = omp_get_num_threads();
            printf("Number of threads = %d\n",nthreads);
        }
    }
}
```

Core Innovation: `vpfloat` Type System

- **Key Innovation**: Arbitrary precision as a first-class language feature with full compiler optimization support

```
vpfloat<format, exponent_bits, precision_bits, [size]>
```

Format: mpfr (software) | unum (hardware)
Exponent bits: Size of exponent field (e.g. 16)
Precision bits: Size of mantissa (e.g. 256)
Size: Optional memory footprint limit

Capabilities: Static Precision (compile-time)

```
void compute_physics(int N,
    vpfloat<mpfr, 16, 256> *positions,
    vpfloat<mpfr, 16, 256> *velocities) {

    for (int i = 0; i < N; i++) {
        velocities[i] = positions[i] * 0.5y + velocities[i];
    }
}
```

- Type-checking at compile time
- Maximum optimization
- Zero runtime overhead
- Performance like a regular float/double

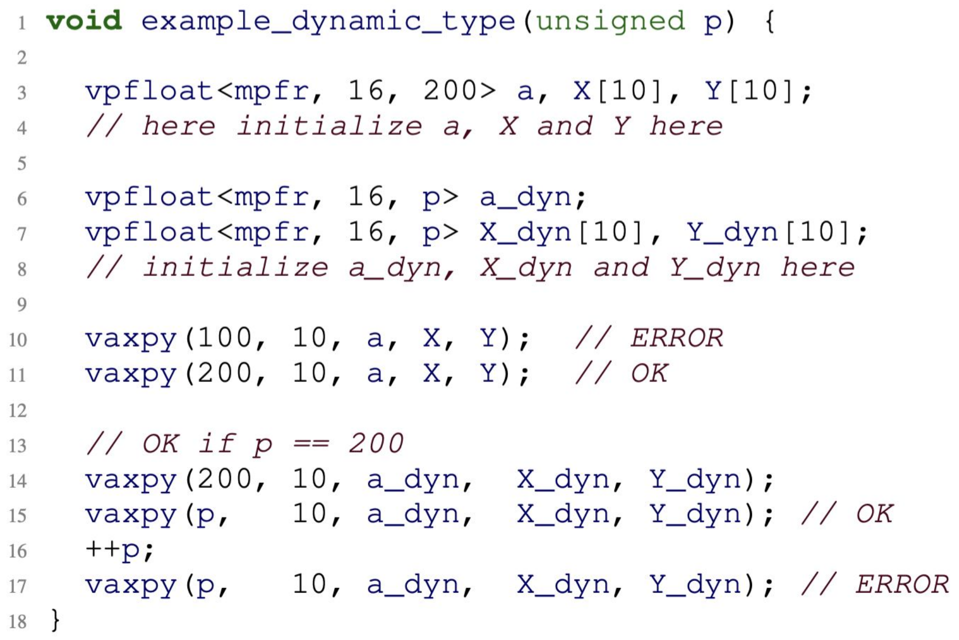
MPFR vs. vpfloat

```
mpfr_t x, y, result;
mpfr_init2(x, 256);
mpfr_init2(y, 256);
mpfr_init2(result, 256);

mpfr_set_d(x, 1.5. MPFR_RNDN);

mpfr_set_d(y, 2.3, MPFR_RNDN);

mpfr_mul(result, x, y, MPFR_RNDN);

mpfr_clear(x);
mpfr_clear(y);
mpfr_clear(result);
```

```
vpfloat<mpfr, 16, 256> x = 1.5y;
vpfloat<mpfr, 16, 256> y = 2.3y;
vpfloat<mpfr, 16, 256> result = x * y;
```

Capabilities: *Dynamic* Precision (runtime)

```
void adaptive_solve(unsigned precision, int N,
    vpfloat<mpfr, 16, precision> *data) {

    vpfloat<mpfr, 16, precision> temp = 0.0y;
    for (int i = 0; i < N; i++) {
        temp += data[i] * data[i];
    }
    return temp;
}
adaptive_solve(128, N, data);
adaptive_solve(512, N, data);
```



```
void example_dynamic_type(unsigned p) {

  vpfloat<mpfr, 16, 200> a, X[10], Y[10];
  // here initialize a, X and Y here

  vpfloat<mpfr, 16, p> a_dyn;
  vpfloat<mpfr, 16, p> X_dyn[10], Y_dyn[10];
  // initialize a_dyn, X_dyn and Y_dyn here

  vaxpy(100, 10, a, X, Y);  // ERROR
  vaxpy(200, 10, a, X, Y);  // OK

  // OK if p == 200
  vaxpy(200, 10, a_dyn,  X_dyn, Y_dyn);
  vaxpy(p,   10, a_dyn,  X_dyn, Y_dyn); // OK
  ++p;
  vaxpy(p,   10, a_dyn,  X_dyn, Y_dyn); // ERROR
}
```

- Precision determined at runtime
  - __sizeof_vpfloat calls generated on declaration of dynamically-sized type
- Single binary, multiple configurations
- No recompilation needed
- Enables adaptive algorithms

Compilation Flow: Integrating into LLVM IR

1) \`vpfloat\` types in C are translated into LLVM intrinsics as a temporary placeholder function that expresses high-level intent.

2) Code remains in this generic intrinsic form while LLVM's highest-level optimizers, like Polly (Loop Nest Optimizer), are run.
   - The compiler can now perform performance-critical transformations (e.g., loop unrolling, fusion, vectorization) on the generic operations, a capability impossible with opaque \`library calls.

3) The optimized intrinsics are resolved in a final compilation pass by a custom backend generator for one of two targets
   - MPFR: Expanded instruction sequences that manage memory/make low-level calls to the MPFR library.
   - UNUM: Instructions for a variable-precision hardware coprocessor ISA.

Elevated Intrinsics/Operations & LLVM Polly

- This “intrinsic modeling intent” strategy keeps the code in a generic, optimizable form for a critical phase of compilation, bypassing the traditional barrier created by opaque library function calls.

- Polly performs transformations to ensure optimal use of processor cache and maximize throughput including:
    - **Loop Tiling:** Reorganizing loop iterations to reuse data already in cache memory.
    - **Data Layout Reorganization:** Optimizing array access patterns for contiguous memory reads.
    - **Loop Fusion:** Merging loops to reduce overhead and increase computational density.
    - **Vectorization:** Converting sequential arithmetic into SIMD instructions that execute multiple operations simultaneously.

- Late lowering strategy keeps MPFR object management until after optimization, reducing memory pressure and enabling up to 90× reduction in cache misses for parallel workloads

Code Examples & Results

```
void axpy_mpfrconst(int N,
                    vpfloat<mpfr, 16, 256> alpha,
                    vpfloat<mpfr, 16, 256> *X,
                    vpfloat<mpfr, 16, 256> *Y) {
    for (unsigned i = 0; i < N; ++i)
        Y[i] = alpha * X[i] + Y[i];
}

void axpy_mpfr(unsigned prec, int N,
               vpfloat<mpfr, 16, prec> alpha,
               vpfloat<mpfr, 16, prec> *X,
               vpfloat<mpfr, 16, prec> *Y) {
    for (unsigned i = 0; i < N; ++i)
        Y[i] = alpha * X[i] + Y[i];
}

void axpy_unumconst(int N,
                    vpfloat<unum, 4, 6, 8> alpha,
                    vpfloat<unum, 4, 6, 8> *X,
                    vpfloat<unum, 4, 6, 8> *Y) {
  for (unsigned i = 0; i < N; ++i)
    Y[i] = alpha * X[i] + Y[i];
}

void gemm_unum(unsigned prec, int M, int N,
               double *A,
               vpfloat<unum, 4, prec> alpha,
               vpfloat<unum, 4, prec> *X,
               vpfloat<unum, 4, prec> beta,
               vpfloat<unum, 4, prec> *Y) {
  for (unsigned i = 0; i < M; ++i) {
    // From III.A.5 "Dynamically-sized Types":
    // alphaAX's dynamic size is computed by
    // __sizeof_vpfloat(4, prec).
    vpfloat<unum, 4, prec> alphaAX = 0.0;
    for (unsigned j = 0; j < N; ++j)
      alphaAX += A[i*N + j] * X[j];
    Y[i] = alpha * alphaAX;
    // Free memory of alphaAX back to stack.
  }
}
```

Listing 2. Sample BLAS functions reimplemented with *mpfr* and *unum* types

```
void example_dynamic_type(unsigned p) {

  vpfloat<mpfr, 16, 200> a, X[10], Y[10];
  // here initialize a, X and Y here

  vpfloat<mpfr, 16, p> a_dyn;
  vpfloat<mpfr, 16, p> X_dyn[10], Y_dyn[10];
  // initialize a_dyn, X_dyn and Y_dyn here

  vaxpy(100, 10, a, X, Y);  // ERROR
  vaxpy(200, 10, a, X, Y);  // OK

  // OK if p == 200
  vaxpy(200, 10, a_dyn,  X_dyn, Y_dyn);
  vaxpy(p,   10, a_dyn,  X_dyn, Y_dyn); // OK
  ++p;
  vaxpy(p,   10, a_dyn,  X_dyn, Y_dyn); // ERROR
}

vpfloat<mpfr, 16, prec> // OK
  example_dyn_type_return (unsigned prec) {
  vpfloat <mpfr, 16, prec> a = 1.3;
  return a;
}

vpfloat<mpfr, 16, prec> // ERROR
  example_dyn_type_return_error (unsigned p) {
  vpfloat <mpfr, 16, p> a = 1.3;
  return a;
}
```

Listing 3. Uses of dynamically-sized types in function call and return

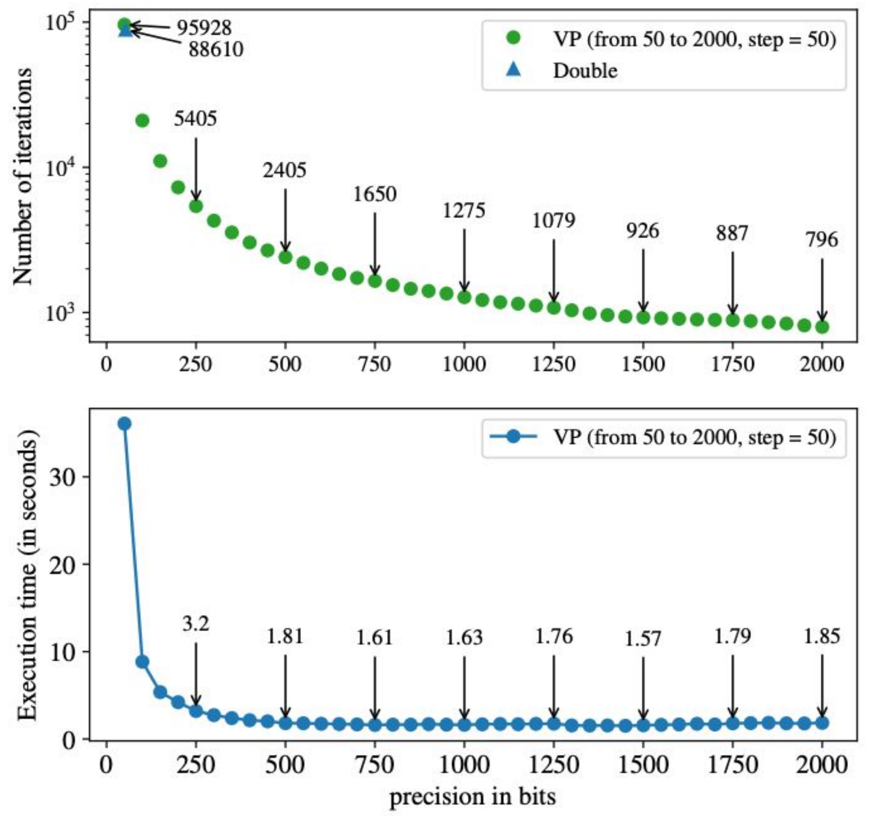


Fig. 3. Conjugate Gradient (CG) iterations and precision on the *bcsstk20* matrix

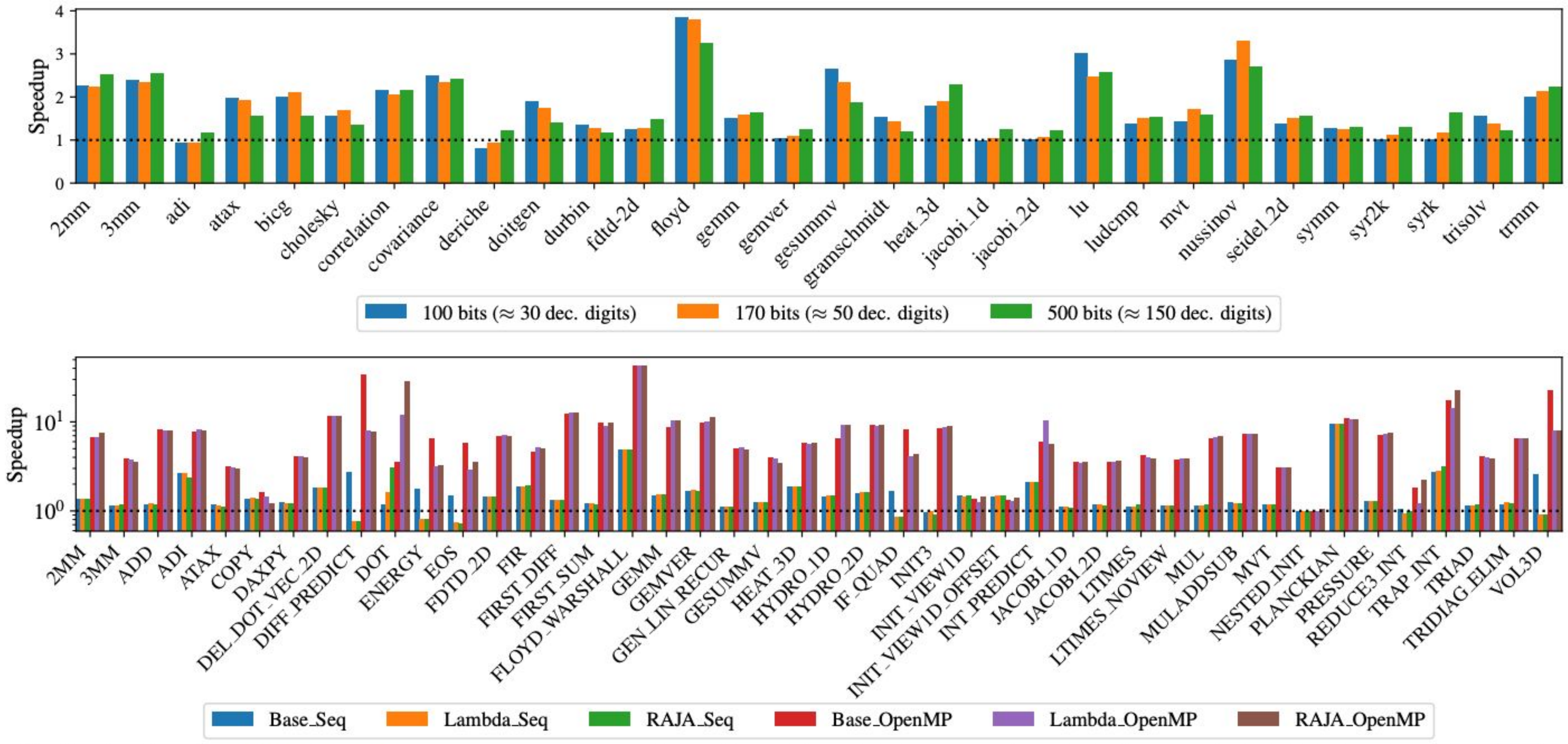


Fig. 1. Speedup of `vpfloat<mpfr, ...>` over the Boost library for multi-precision both targeting MPFR library calls for (1) the Polybench benchmark suite, and (2) the RAJAPerf benchmark suite.

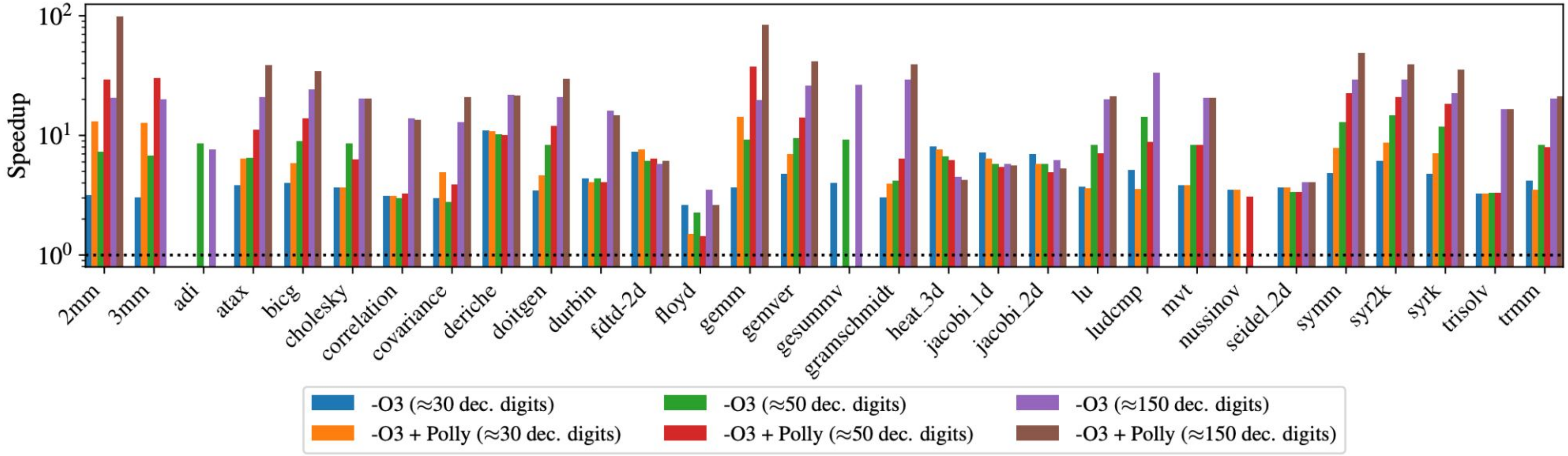


Fig. 2. Speedup of `vpfloat<unum, ...>` over `vpfloat<mpfr, ...>` on the PolyBench suite

More on UNUM Coprocessor

- Paper targets simplified version of **RISC-V extension for UNUM** proposed by Bocco et al.
  - **Loads/stores parametrized** by current **UNUM config**
  - Special **ess** and **fss** control registers determine how many bytes to move
  - **WGP** (Working G-Layer precision) controls bit of **precision used in computation** by ALU
  - **MBB** (Memory Byte Budget) controls how many bytes memory interface can touch per load/store
- Authors implement **two passes** on `vpfloat` types to support coprocessor
  - **FP Configuration**: Analyzes CFG to properly **configure ess, fss, WGP, and MBB** throughout the program, guaranteeing correctness
  - **Array Address Computation**: **replace** instances of ***GetElementPtr*** with address computation using **__sizeof_vpfloat** (only needed for dynamic precision)
- **FPGA implementation** of ISA extension showed **>20x speedup** compared to MPFR

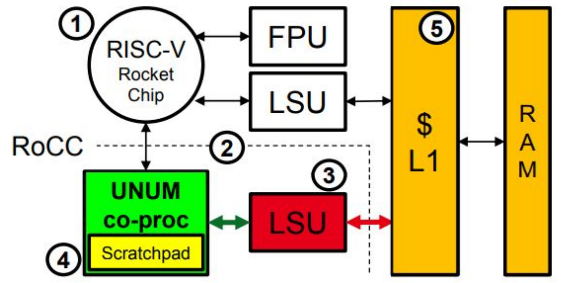


Figure 5: SMURF architecture overview

Comments

Key Results

- 1.8× speedup over Boost (sequential), 7.6× with OpenMP through late lowering and reduced cache misses.
- Hardware UNUM: 10-100× over software MPFR; Polly adds 2-20× for compute-intensive kernels Higher precision paradoxically reduces runtime—conjugate gradient solver: 100× fewer iterations, 10× faster execution.

Strengths

- First-class IR support enables transparent optimization (Polly, register allocation) without modifying passes.
- Retargetable: single type system works with software (MPFR) and hardware (UNUM) backends.
- Runtime precision exploration within single program run—essential for transprecision computing.

Auto-tuning Framework for Compiler-Driven Precision Selection

- Automatically analyze code to determine minimum precision needed for acceptable accuracy, eliminating manual tuning and making variable-precision accessible to non-experts

Complete C++ Dynamic Type Support for Advanced OOP Features

- Enable dynamically-sized vpfloat types in classes, virtual functions, and lambdas—currently limited because C++ object layout requires compile-time sizes