

Fast Arbitrary Precision Floating Point on FPGA

Johannes de Fine Licht^{*}, Christopher A. Pattison[†], Alexandros Nikolaos Ziogas^{*},
David Simmons-Duffin[‡], Torsten Hoefler^{*}

^{*}Department of Computer Science, ETH Zurich, Switzerland

[†]Institute for Quantum Information and Matter, Caltech, Pasadena, USA

[‡]Walter Burke Institute for Theoretical Physics, Caltech, USA

Abstract—Numerical codes that require arbitrary precision floating point (APFP) numbers for their core computation are dominated by elementary arithmetic operations due to the super-linear complexity of multiplication in the number of mantissa bits. APFP computations on conventional software-based architectures are made exceedingly expensive by the lack of native hardware support, requiring elementary operations to be emulated using instructions operating on machine-word-sized blocks. In this work, we show how APFP multiplication on compile-time fixed-precision operands can be implemented as deep FPGA pipelines with a recursively defined Karatsuba decomposition on top of native DSP multiplication. When comparing our design implemented on an Alveo U250 accelerator to a dual-socket 36-core Xeon node running the GNU Multiple Precision Floating-Point Reliable (MPFR) library, we achieve a $9.8\times$ speedup at 4.8 GOp/s for 512-bit multiplication, and a $5.3\times$ speedup at 1.2 GOp/s for 1024-bit multiplication, corresponding to the throughput of more than $351\times$ and $191\times$ CPU cores, respectively. We apply this architecture to general matrix-matrix multiplication, yielding a $10\times$ speedup at 2.0 GMAC/s over the Xeon node, equivalent to more than $375\times$ CPU cores, effectively allowing a single FPGA to replace a small CPU cluster. Due to the significant dependence of some numerical codes on APFP, such as semidefinite program solvers, we expect these gains to translate into real-world speedups. Our configurable and flexible HLS-based code provides a high-level software interface for plug-and-play acceleration, published as an open source project.

I. INTRODUCTION

Arbitrary precision arithmetic, such as that implemented by the GNU Multiple Precision (GMP) [1] and Multiple Precision Floating-Point Reliable (MPFR) [2] libraries (where it is referred to as “multi-precision” arithmetic), allows increasing precision by extending the number of bits used to represent numbers beyond the machine word size natively supported by software processors. This can be necessary to accurately investigate domains where information is found in small differences between numbers (i.e., numbers that are very similar and nearly cancel each other out), which cannot be effectively captured by the dynamic precision of floating-point arithmetic.

As motivation for this work, we consider semidefinite programs (SDPs). SDPs are ubiquitous and efficiently solvable convex optimization problems involving a linear cost function of a positive-semidefinite matrix subject to affine constraints [3]. SDPs have myriad applications in fields such as control theory, combinatorial optimization, algebraic geometry, and operations research [4], [5], [6], [7]. A popular approach to solving the resulting SDPs is primal-dual interior-point methods, which rely on matrix decompositions and matrix-

matrix multiplication. However, such methods frequently encounter ill-conditioned matrices, and consequently, several solvers have been implemented to solve SDPs using high-precision arithmetic [8], [9], [10], [11]. A state-of-the-art SDP library is SDPB [11], [12], an interior-point solver designed to handle semidefinite programs that arise in the conformal bootstrap. The conformal bootstrap is a powerful framework for studying phase transitions in a wide variety of physical systems [13], [14], [15]. Its central strategy is to solve a series of SDPs to derive rigorous bounds on physical quantities [16], [17]. In addition to phase transitions, SDPB has been applied to problems in sphere packing [18], scattering amplitudes [19], [20], and differential geometry [21], [22]. In all of these cases, the relevant physical or mathematical system satisfies an infinite set of consistency conditions, only a finite subset of which are used in a given SDP. *High-precision arithmetic enables one to easily and robustly obtain stronger constraints by systematically enlarging the number of consistency conditions (and the size and complexity of the corresponding SDPs).*

Unfortunately, moving from 64-bit machine word arithmetic to arbitrary precision comes at an immense computational cost. Fundamental operations, such as addition and multiplication, can no longer be implemented with single instructions and must instead be emulated using a (potentially long) sequence of instructions operating on individual machine-word-sized blocks of the number. As a result, the runtime of numerical codes that require arbitrary precision arithmetic in their core computation can quickly become dominated by elementary arithmetic operations. This is exacerbated by the super-linear complexity of multiplication (and consequently dependent operations such as division) in the number of mantissa bits, which, depending on the instruction mix, can result in arbitrary precision multiplication *alone* dominating workloads such as linear algebra. While specialized instructions have been introduced to x86 to mitigate this, namely ADCX (add with carry) and MULX (unsigned integer multiplication with double-width output), the issue of emulation and complexity remains.

The reconfigurable hardware fabric in FPGA devices allows deploying custom circuits in terms of the elementary components available on the chip. Due to the importance of machine learning workloads, recent work in both fixed and reconfigurable hardware acceleration has focused on low precision types [23], [24], [25]. However, FPGAs are also an excellent platform for going in the other direction: While they often cannot compete with GPUs on accelerating traditional

floating-point-dominated workloads, they have a significant advantage on data types that are not natively supported by the instruction sets of other architectures, such as arbitrary precision arithmetic, as these operations can be unrolled and deeply pipelined on the chip. By accelerating the basic arbitrary precision operators on FPGA, the speedup achieved can then directly translate into real-world speedup in codes that are dominated by arbitrary precision arithmetic.

In this work, we show how a Karatsuba-based arbitrary precision floating point (APFP) multiplier implemented on a single FPGA device can outperform $351\times$ CPU cores executing MPFR. We deploy this architecture in a general matrix-matrix multiplication (GEMM) accelerator, which is a crucial component of many numerical workloads, yielding a design that outperforms $375\times$ CPU cores. The accelerator is exposed through a BLAS-like software interface and published as open-source code on GitHub¹, allowing plug-and-play FPGA acceleration of existing APFP-dominated workloads by modifying a few lines of code. The HLS-based code is highly configurable to support different precisions, tile sizes, FPGA architectures, DRAM layouts and more, and can target any platform supported by Xilinx’ Vitis toolflow and the Xilinx Runtime (XRT), including both current and future devices. Following our approach, this acceleration can be extended to other APFP routines in linear algebra and beyond, providing significant speedup that can enable new science in practice.

II. ARBITRARY PRECISION FLOATING POINT OPERATORS

The most fundamental arithmetic building blocks of most computations in high-performance computing (HPC) are addition (including subtraction) and multiplication. The most common performance metric used to evaluate and rank HPC systems is their throughput in terms of these two operators. For arbitrary precision-based codes, they are the most critical to accelerate. In the following, we cover our FPGA implementation for APFP addition and multiplication.

We base the functional behavior of our arithmetic on that implemented in the GNU MPFR library, using the round-to-zero mode (MPFR_RNDZ). In MPFR, an APFP number is implemented as a struct containing four runtime fields: the number of bits used for the mantissa; the sign, stored as a machine word; the exponent, stored as a machine word; and a pointer to a heap-allocated array of “limbs”, where each limb is a machine word-sized chunk of the mantissa.

To adapt the MPFR representation to a hardware-suitable format, we apply the following changes to the representation, without affecting the functional semantics of the operators:

- The number of bits used for the mantissa is kept configurable but fixed at compile-time, allowing us to omit this field from the data type at runtime.
- The sign is packed into a single bit of the exponent, reducing the exponent to a $(b_{\text{limb}} - 1)$ -bit signed integer (e.g., 63 bits), where b_{limb} is the machine word size that MPFR is configured with (typically 64 bits).

¹<https://github.com/spcl/apfp>

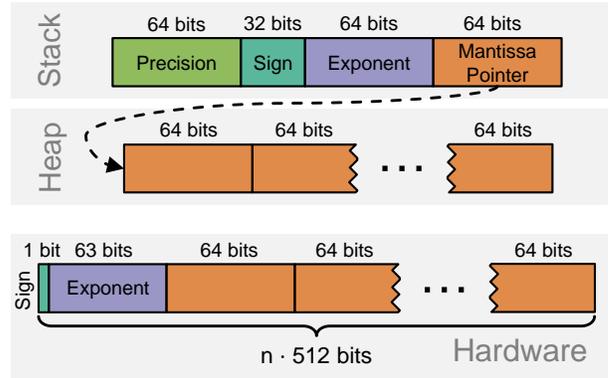


Fig. 1: The MPFR software representation on a 64-bit system (top) transformed into a hardware-friendly format (bottom) for integer n , where $n \cdot 512$ bit – 64 bit represent the mantissa.

- The mantissa is packed tightly with the sign and exponent rather than being allocated separately, which is possible due to the precision being configured at compile-time.
- The combined sign, exponent, and mantissa are packed into a multiple of 512 bits to enable efficient memory accesses.

The format transformation is illustrated in Fig. 1 for a system with 64-bit machine words. The provided `ap_uint` arbitrary precision integer type in Vitis HLS is used to pack the sign, exponent, and mantissa tightly and ensure that wide buses are generated on the FPGA. Our operators will maintain **full bit-compatibility** in the mantissa with MPFR, and their output will be compared to the equivalent MPFR software computation to verify correctness of the implementation.

A. Floating-Point Multiplier

The majority of work involved in floating-point multiplication lies in the underlying unsigned integer multiplication of the two mantissas. Consequently, multiplying the mantissas will account for the majority of hardware utilization in the floating-point multiplication kernel, and the majority of hardware utilization in all the kernels benchmarked in this work.

Naive multiplication of integers (commonly referred to as the “textbook” algorithm) requires $O(b^2)$ work in the number of bits b used to represent the integers. However, by recursively decomposing and reorganizing the multiplication into subcomponents, some redundant subcomputations can be eliminated to reduce the asymptotic complexity at the cost of higher constants, first described by Karatsuba [26] achieving $O(b^{\log_2 3})$, and later generalized by Toom [27] and described by Cook [28] (the scheme is now commonly referred to as Toom–Cook multiplication). For very high b (not considered in this work), FFT-based methods become practical [29], [30].

In this work, we consider bit widths that are “large” from a hardware perspective (i.e., an order of magnitude wider than the 64-bit words natively supported in CPU and GPU architectures), but “small” relative to the overhead imposed by higher-order Toom–Cook and FFT-based methods. To this end, we employ the Karatsuba algorithm for our hardware

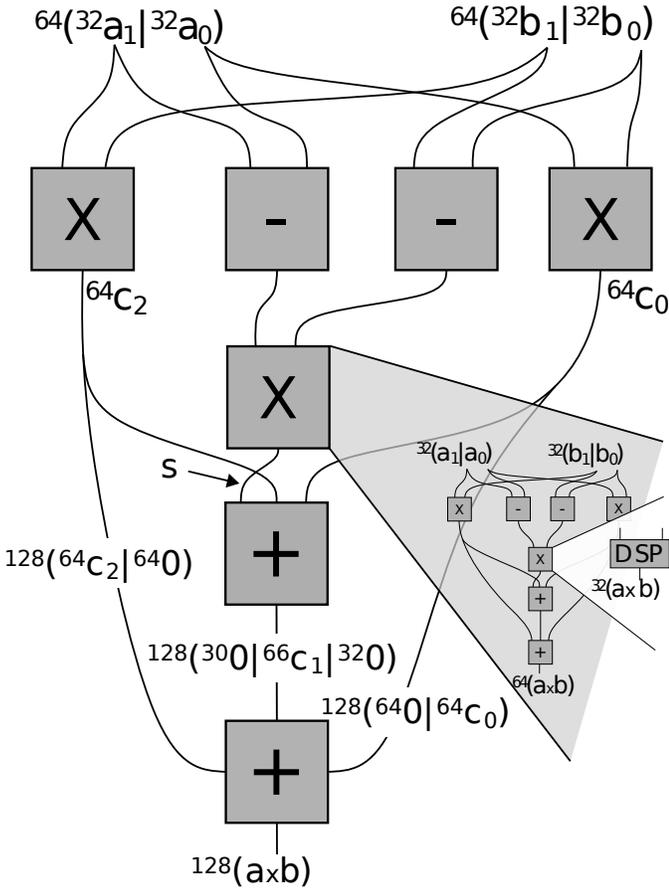


Fig. 2: Recursive Karatsuba decomposition of 64×64 -bit integer multiplication. The explicitly tracked sign bit s in the intermediate computation of c_1 is indicated as an arrow. Sums with three terms are represented with a single “+” box. Note that the 16×16 -bit multiplication at the lowest level is computed in hardened DSP48E2 units.

implementation (also known as Toom-2), which offers a hardware-friendly power-of-two decomposition and is efficient in this middle-ground domain.

The Karatsuba multiplication algorithm and its generalization utilize techniques for fast multiplication of polynomials by viewing the result and operands as a polynomial in a base $B = 2^n$ i.e. $a = a_0 + Ba_1$. Here, we describe a single recursive step with bitwidths annotated as superscripts. Each step splits the input operands of $c = ab$ into two halves, the high and low bits ${}^{(2n)}a = {}^{(n)}a_0 + B {}^{(n)}a_1$, and requires three multiplications at half-bit width per recursive step. Writing c as $c = c_0 + Bc_1 + B^2c_2$ and b as ${}^{(2n)}b = {}^{(n)}b_0 + B {}^{(n)}b_1$, the coefficients are computed as follows:

$$\begin{aligned} {}^{(2n)}c_0 &= {}^{(n)}a_0 {}^{(n)}b_0 \\ {}^{(2n)}c_2 &= {}^{(n)}a_1 {}^{(n)}b_1 \\ {}^{(2n)}t &= {}^{(n)}|a_1 - a_0| {}^{(n)}|b_1 - b_0| \\ s &= \text{sign}[(a_1 - a_0)(b_1 - b_0)] \end{aligned}$$

$${}^{(2n+2)}c_1 = {}^{(2n)}c_0 + {}^{(2n)}c_2 - {}^{(1)}s {}^{(2n)}t$$

Explicitly tracking the sign bit in the computation of c_1 allows all multiplications to be carried out at n bits. Note that only one multiplication per coefficient is required (c_0 , c_2 , and t). We recombine the outputs using multiplication by B implemented as shifts. Since a product cannot have more than double the number of digits of the operands, one can see that this addition will not overflow. Combining the contributions yields:

$${}^{(4n)}c = {}^{(2n)}c_0 + B {}^{(2n+2)}c_1 + B^2 {}^{(2n)}c_2$$

The decomposition may then be repeated iteratively for the three half-bit width multiplications until reaching a small enough bit width to perform the multiplication as a primitive operation. Our recursive implementation of the decomposition is sketched for 64×64 -bit example inputs in Fig. 2.

DSPs in modern FPGAs can natively (and thus efficiently) perform integer multiplication up to a given bit width, and will be used when “bottoming out” our decomposition. The DSP48E2 units on the Xilinx UltraScale+ architecture support 18×18 -bit integer multiplication. On this architecture, we thus recursively split the domain until the subcomponents are at most ≤ 18 bits in size, after which they can be directly dispatched to DSP units rather than being decomposed further. However, the bottom out bit width is left as a configuration parameter, as falling back on $O(n^2)$ multiplication at a higher bit width can be beneficial (see Sec. V-A).

To implement the Karatsuba decomposition in a general manner to support any input bit width, we exploit C++ template metaprogramming to define a static template recursion that bottoms out on bit widths under the defined bottom out width using an SFINAE [31] pattern. This is illustrated in Lst. 1, where the `ap_uint` type is used to represent arbitrary bit widths, and `MULT_BASE_BITS` is the chosen threshold where Karatsuba falls back on naive multiplication using DSPs (we will optimize the choice of this threshold in Sec. V-A).

When combining contributions to the final mantissa, we perform integer additions on bit widths up to $2 \times$ the number of input bits (e.g., 1024-bit operands for 512-bit numbers). Vitis HLS 2021.2 allows splitting the adder into multiple stages using the `BIND_OP` pragma, but only allows a maximum of 4 additional pipeline stages. To avoid deep combinatorial logic and aid routing, we implement an additional pipelined addition/subtraction function that partitions the wide additions into chunks of a configurable base width. We use this to make sure that no more than a fixed number of bits are added in a single cycle, and will show how this impacts resource usage and frequency in Sec. V-A.

B. Floating-Point Adder

Addition of mantissas can be accomplished in a time complexity linear in the number of bits. In the same way as for adding up contributions in Karatsuba multiplication, we partition the integer addition into a configurable number of stages. To perform a *floating-point* addition, we shift the operands by the difference of the exponents before passing them into

```

1 template <int bits>
2 auto Karatsuba(ap_uint<bits> const &a,
3               ap_uint<bits> const &b) ->
4     typename std::enable_if<(bits > MULT_BASE_BITS),
5               ap_uint<2*bits>>::type {
6     using Full = ap_uint<bits>;
7     using Half = ap_uint<bits / 2>;
8     Half a0 = a(bits/2-1, 0); Half a1 = a(bits-1, bits/2);
9     Half b0 = b(bits/2-1, 0); Half b1 = b(bits-1, bits/2);
10    Full c0 = Karatsuba<bits / 2>(a0, b0); // Recurse
11    Full c2 = Karatsuba<bits / 2>(a1, b1); // Recurse
12    // ...compute |a1-a0| and |b1-b0|...
13    Full c1 = Karatsuba<bits / 2>(a1_a0, b1_b0); // Recurse
14    // ...combine all contributions and return...
15 }
16
17 template <int bits>
18 auto Karatsuba(ap_uint<bits> const &a,
19               ap_uint<bits> const &b) ->
20     typename std::enable_if<(bits <= MULT_BASE_BITS),
21               ap_uint<2*bits>>::type {
22     return a * b; // Bottom out using naive mult
23 }

```

Listing 1: Static recursion pattern implemented in C++ bottoming out at `MULT_BASE_BITS` with `SFINAE` used to implement Karatsuba decomposition for arbitrary bit widths.

the integer adder. Due to the sign-magnitude format of the floating-point format, we must explicitly subtract the operands when the signs differ. When subtracting, the output may become denormalized, requiring us to left-shift the resulting mantissa such that the most significant bit of the mantissa is set, requiring us to count the number of introduced leading zeros and dynamically shift by this number.

We combine the floating-point adder with the multiplier to form a combined multiply-addition pipeline, which can serve as a building block for dense linear algebra kernels.

III. ARBITRARY PRECISION MATRIX MULTIPLICATION

With a fully pipelined multiply-addition unit that performs one operation per cycle, DRAM bandwidth is no longer sufficient to saturate the compute in a linear streaming computation. We thus need to increase the granularity of acceleration to routines that enable sufficient reuse to keep the compute saturated from buffers in on-chip memory. For use in the SDP solvers that motivate this work, general matrix multiplication (GEMM) and derived routines such as the symmetric rank-k update (SYRK) BLAS routine are major workhorses that can provide the necessary reuse.

We design a GEMM architecture that implements the operation $C = \alpha AB + \beta C$, where A is an $N \times K$ matrix, and B is a $K \times M$ matrix. For the purpose of this work, we fix $\alpha = \beta = 1$, but other values can be introduced at the cost of requiring additional multiplication pipelines, which would correspond to a nearly full replication of the circuit. Reuse is achieved through a 2D tiling scheme, where columns of size T_N from A and rows of size T_M from B are loaded and used to compute a $T_N \times T_M$ outer product, which is accumulated into an output tile of size $T_N \cdot T_M$ of matrix C stored in on-chip memory. This is repeated for the full common matrix dimension K until the output tile is complete and is written back to off-chip memory. By setting $T_N = T_M$ and maximizing this quantity,

we can achieve optimal fast memory usage in terms of the on-chip memory used [32], with an arithmetic intensity of $\frac{T_N T_M}{T_N + T_M}$ ($T_N T_M$ computations for each $T_N + T_M$ operands loaded from memory).

With the outer product scheme selected, one of the input matrices will be read column-wise, while the other will be read row-wise. For the matrix that is not read contiguously from memory, the accesses to DDR memory are less efficient as a result. Fortunately, because each entry occupies a much larger space in memory than traditional data types, even this suboptimal access pattern results in burst reads at least as wide as the floating point number. This is chosen as a multiple of 512 bits to match the $4 \times$ clock multiplier of DDR4 memory, the $2 \times$ data rate, and the 64-bit DDR4 interface.

When permitted by available resources and routing constraints, we can instantiate multiple GEMM compute units to improve overall throughput. Each compute unit will operate on a distinct partition of the output matrix, such that multiple GEMM accelerators collaborate on a single virtual GEMM call. For P compute units, N/P rows of the input matrix A and the output matrix C are allocated per accelerator and copied to the respective DRAM bank, while the full B -matrix is used by every compute unit to compute a complete set of N/P rows of the output matrix.

IV. ARTIFACTS AND WORKFLOW

We publish our HLS-based accelerator and the software integration code as open source software, to facilitate it being exploited in APFP-based numerical codes. The hardware accelerator is highly configurable, and once the appropriate bitstream has been built and installed, can be accessed through a high-level BLAS interface, or through CUDA-like device interaction for more fine-grained control.

A. Hardware Accelerator Configuration

Both software and hardware of our project is configured via CMake. Dependencies required to build the code are automatically detected, including the Xilinx toolchain as enabled by `FindVitis.cmake` provided by the `hlslib` [33] project, which also provides build targets for hardware and hardware emulation for our kernels.

As of writing, the matrix multiplication accelerator can be configured with the following parameters that customize its resource utilization and performance characteristics:

- `APFP_BITS` configures the number of bits used to represent floating point numbers, which includes the bits spent on exponent and sign (packed according to Fig. 1).
- `APFP_COMPUTE_UNITS` sets the replication factor of the multiply-addition pipeline, allowing performance to be scaled up with available resources on the target device.
- `APFP_TILE_SIZE_N` and `APFP_TILE_SIZE_M` configure the rows and columns of the output tile *per instantiated compute unit*, respectively, increasing memory reuse/reducing memory bandwidth at the cost of on-chip memory resources, as described in Sec. III.

```

1 El::DistMatrix<El::BigFloat> distr_a = ...;
2 El::DistMatrix<El::BigFloat> distr_b = ...;
3 El::DistMatrix<El::BigFloat> distr_c = ...;
4
5 // Elemental GEMM
6 El::Gemm(El::NORMAL, El::NORMAL, El::BigFloat(1),
7         distr_a, distr_b, El::BigFloat(1), distr_c);
8
9 // Obtain local copies
10 using LocalMatrix =
11     El::DistMatrix<El::BigFloat, El::CIRC, El::CIRC>;
12 LocalMatrix local_a = distr_a;
13 LocalMatrix local_b = distr_b;
14 LocalMatrix local_c = distr_c;
15
16 // Indexing functions into the matrices
17 using CIdxF = std::function<mpfr_srcptr(unsigned long)>;
18 using IdxF = std::function<mpfr_ptr(unsigned long)>;
19
20 CIdxF index_A = [&](unsigned long i) {
21     return local_a.Matrix().Buffer()[i].LockedPointer();
22 };
23
24 // ...define index_B and index_C...
25
26 // APFP Interface GEMM Call
27 apfp::Gemm(apfp::BlasTrans::normal,
28           apfp::BlasTrans::normal, m, n, k,
29           index_A, local_a.Matrix().LDim(),
30           index_B, local_b.Matrix().LDim(),
31           index_C, local_c.Matrix().LDim());

```

Listing 2: Example GEMM call for Elemental and for the BLAS compatibility interface. CPU codes relying on Elemental can be converted piece-by-piece by retaining the Elemental data structures.

- `APFP_MULT_BASE_BITS` and `APFP_ADD_BASE_BITS` configure the bit width at which the Karatsuba decomposition falls back on naive multiplication of operands, and the number of bits added in combinatorial logic per pipeline stage when performing wide additions, respectively.

By adapting these configuration options to the specific architecture being targeted, the accelerator can be tailored to fully exploit available resources, including logic resources and DRAM banks, and best utilize the characteristics of the underlying hardware components.

B. System Integration

To make it easy for numerical codes to exploit FPGA acceleration, we expose our accelerator as a high-level software library with BLAS-like API calls. The BLAS interface permits the FPGA acceleration to be a drop-in replacement for libraries such as LAPACK [34] or Elemental [35] when the transfer overhead is small relative to the computation size.

Elemental is a distributed memory dense linear algebra library, which supports arbitrary precision data types relying on MPFR data types, and uses MPI for parallelization and multi-node support. Using our BLAS interface, we are able to non-invasively accelerate a GEMM call in an Elemental program with minimal additional code (Lst. 2). The BLAS interface accepts a pointer to a buffer or an `std::function/lambda` function, accepting an integer and returning an MPFR pointer. This flexibility permits us to avoid copying MPFR data out of the Elemental datatypes while simultaneously avoiding a leaky

abstraction with respect to our internal packed floating-point format. The MPFR datatype stores limbs on the heap, so the extra indirection imposed by the indexing function is not a significant drawback.

In Lst. 2, we show a standard GEMM call in Elemental (line 6) operating on distributed matrices in addition to a call to the FPGA BLAS interface (line 27). In this example, the operands are distributed matrices, so they are copied to a single node using the `El::CIRC` distributed matrix distribution argument (line 10). The only additional code is to define indexing functions that abstract away the layout of the underlying MPFR numbers inside of Elemental (line 17).

While this example copies a distributed matrix to a single MPI process, the Elemental library could be used to facilitate a distributed, multi-FPGA computation.

When data movement to/from the accelerator must be explicitly managed, we provide a fine-grained interface exposing a CUDA-like API to launch kernels and move data between host and device. Workloads with many small matrices will need to keep operands on the FPGA for multiple kernel invocations to amortize the transfer time.

V. EVALUATION

We evaluate our architecture on a Xilinx Alveo U250 accelerator, where we utilize 1–4 DDR4 DRAM banks with a peak bandwidth of 19.2 GByte/s per bank. The C++-based kernels are implemented in Vitis HLS with `hlslib` [33] extensions, and compiled for hardware with Vitis/Vivado 2021.2, targeting the `xilinx_u250_gen3x16_xdma_3_1_202020` shell through the OpenCL-based interface relying on the Xilinx Runtime (XRT) version 2.9.317 for host/device interaction. The U250 consists of $4\times$ chiplets called “Super Logical Regions” (SLRs) that have limited connectivity between them. We thus force kernel instantiations to stay within the bounds of a chiplet to avoid frequency degradation.

To compare against software, we run APFP computations in software using MPFR 4.1.0 and GMP 6.2.1. For dense linear algebra, we run commit 6eb15a0 of Elemental² [35] with MPFR/GMP and MPI support. Benchmarks are run on Cray XC40 compute nodes on the Piz Daint supercomputer at the Swiss National Supercomputing Center (CSCS), where each node is equipped with $2\times$ Intel Xeon E5-2695 v4 18-core CPUs in a dual-socket configuration (36 cores per node). The Broadwell-based CPU supports the specialized `ADCX` add-with-carry instruction from the Intel `ADX` x86 instruction set extension targeting arbitrary-precision arithmetic, as well as `MULX` instruction from the BMI2 extension for 64×64 -bit multiplication with 128-bit output. GMP, MPFR, and Elemental are compiled directly on the compute nodes with (Cray) GCC 10.3.0 to exploit these and other architecture-specific optimizations. Elemental is built with Cray-MPICH 7.7.16. MPI processes are fixed to CPU cores through Slurm to avoid rescheduling of threads across the NUMA boundary.

²<https://github.com/elemental/Elemental>



Fig. 3: Resource utilization (on the color scale, where brighter colors use more resources) and frequency (annotated on each rectangle, in MHz) for different number of bits added per pipeline stage, and different thresholds for falling back from Karatsuba onto DSP-based naive multiplication. Pareto efficient configurations are marked in **underlined bold font**.

A. Tuning the Multiplier for Resources and Frequency

When configuring the APFP multiplier, there are two tunable parameters that represent a trade-off between frequency and resource usage: the threshold at which the Karatsuba decomposition bottoms out and calls naive multiplication using DSPs (APFP_MULT_BASE_BITS); and the number of bits added/subtracted in a single pipeline stage when adding up contributions (APFP_ADD_BASE_BITS). To find the best configurations, we perform a full sweep of this design space for a single 512-bit APFP multiplier, and use this to guide our other experiments. We choose the number of configurable logic blocks (CLBs) as the metric for resource usage, as this is the most utilized resource in our designs, and captures both LUT and register usage. This results in a 2D design space (multiplication and addition configuration) with two evaluation metrics (frequency and CLBs used).

Fig. 3 shows resource utilization (on the color scale) and frequency (annotated) for different combinations of addition and multiplication configurations for the Karatsuba-based multiplier. For multiplication, the best results are obtained when falling back on DSP-based naive multiplication after 72 bits (lowest resource usage with high frequencies), or 36 bits (consistently high frequencies, but higher resource usage). At 144 bits, the naive multiplication significantly hampers the achievable frequency, while 288 bits fails synthesis altogether. For addition, the best results are obtained when bottoming out at more than 64 bits per pipeline stage. We will target permutations of these configurations of widths for obtaining the best results in the experiments below.

B. Benchmarking Floating-Point Multiplication

To evaluate and compare the performance of the APFP multiplication in isolation, we construct a microbenchmark for

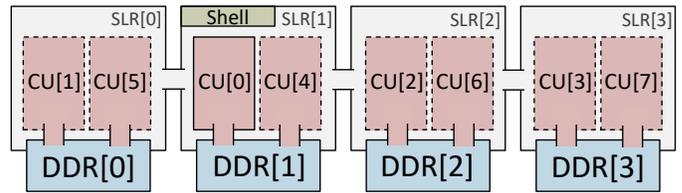


Fig. 4: Example mapping of compute units to SLRs/DDR banks on the U250. Only CU[0] is functionally required (solid outline). Round robin continues after the first 8 CUs.

both FPGA and CPU that streams from two arrays of operands through the multiplier and writes to an output array in a purely linear fashion. In this setting, a fully pipelined FPGA multiplier will quickly become memory bound, as it requires 2 reads and 1 write per cycle, which corresponds to 57.6 GByte/s for a *single* 512-bit pipeline at 300 MHz, or 115.2 GByte/s for a single 1024-bit pipeline. Two compute units would thus already grossly exceed the 76.8 GByte/s peak memory bandwidth of the U250. To evaluate the performance when the compute *can* be fully saturated through memory reuse and/or higher memory bandwidth, we artificially removed the memory bottleneck for the sake of this comparison, by repeatedly feeding the same single data element to the computational kernel. Similarly, although we expect the CPU to primarily be compute bound when running MPFR, we negate any impact from cache misses by constructing the benchmark such that it loops over a dataset that fits in the L1 cache of each Xeon core to ensure the highest possible multiplication throughput for our comparison, representing its true peak running MPFR.

For the FPGA accelerator, we replicate the multiplication pipeline to increase the utilization of the FPGA and partition the input problem across the replications. Each compute unit is assigned to a DDR bank in a round-robin fashion, resulting in each unit being assigned to a distinct SLR (chiplet) on the device. We start at DDR bank 1 where the logic interacting with the host is located, then cycle through 0, 2, and 3. Once a compute unit has been assigned to each bank/SLR, the assignment repeats from the first bank. The SLR/bank assignment is illustrated in Fig. 4 for up to 8 compute units.

We compare an increasing number of compute units instantiated on the FPGA against the full 36-core node running MPFR in Tab. I and Tab. II for 512 bits (448-bit mantissa) and 1024 bits (960-bit mantissa) of precision, respectively. The

Configuration	Freq.	CLBs	DSPs	Throughput	Speedup	#Cores
36-core CPU	2100 MHz	-	-	490 MOp/s	1.0×	36×
FPGA 1 CU	456 MHz	16%	4%	451 MOp/s	0.9×	33.1×
FPGA 4 CUs	376 MHz	37%	14%	1502 MOp/s	3.1×	110.3×
FPGA 8 CUs	300 MHz	48%	28%	2401 MOp/s	4.9×	176.3×
FPGA 12 CUs	300 MHz	62%	42%	3595 MOp/s	7.3×	264.0×
FPGA 16 CUs	300 MHz	75%	56%	4784 MOp/s	9.8×	351.3×

TABLE I: Our 512-bit (448-bit mantissa) floating-point multiplier executed in hardware, compared to MPFR executed fully in L1 cache on a 36-core CPU node. #Cores denotes speedup over a single core (i.e., equivalent number of CPU cores).

Configuration	Freq.	CLBs	DSPs	Throughput	Speedup	#Cores
36-core CPU	-	-	-	227 MOp/s	1×	36×
FPGA 1 CU	361 MHz	27%	8%	361 MOp/s	1.6×	57.3×
FPGA 4 CUs	293 MHz	58%	42%	1202 MOp/s	5.3×	190.9×

TABLE II: Our 1024-bit (960-bit mantissa) floating-point multiplier executed in hardware, compared to MPFR executed fully in L1 cache on a 36-core CPU node.

512-bit multiplier fits up 4 times on each SLR for a total of 16 compute units, yielding 4.8 GOp/s for a speedup over the full 36-core Xeon node of 9.8×, corresponding to a throughput of more than 351× CPU cores at 75% CLB usage and 56% DSP usage. The 1024-bit multiplier can be instantiated once per SLR, yielding 1.2 GOp/s for a 5.3× speedup over the Xeon node (corresponding to 191× CPU cores).

In the following, we will extend our accelerator to perform matrix multiplication, where we can saturate the computational pipeline without artificially removing the memory bound.

C. Benchmarking Matrix Multiplication

We evaluate the accelerator described in Sec. III, where we maximize the number of compute units that can be instantiated within the resource constraints and allowed by routing according to the SLR/DDR bank assignment scheme in Fig. 4. For the CPU comparison, we run the `E1::Gemm` implementation from Elemental, which is parallelized using MPI. We use a tile size of 32×32 for the FPGA compute units, which balances the trade-off between avoiding useless work on sizes that are not a multiple of the tile size with the reduction in required memory bandwidth at larger tile sizes.

Fig. 5 plots the performance of our accelerator for 512-bit APFP numbers with 448-bit mantissas against the matrix dimension for $n \times n$ matrices for different numbers of replications of the compute unit instantiated on the chip, compared to 1–8 Xeon compute nodes running Elemental/MPFR (dashed lines), in multiply-additions per second (we annotate the more commonly used “multiply-accumulate” throughput (MMAC/s), but note that our addition is not restricted to accumulation). Resource usage is dominated by multiplication, making it the primary constraint on how far we can scale the design (in contrast to machine word-sized floating-point, where additions and multiplications are typically weighted the same when reporting performance). For the MPFR/Elemental performance, we run both 448-bit and 512-bit mantissas and take the maximum performance between each pair, to account for performance effects that can occur when the mantissa size is not a power of two.

A single replication of the 512-bit accelerator exhibits performance corresponding to ~ 1 – 2 Xeon nodes (~ 60 cores), while the 8-way replicated accelerator corresponds to the throughput of $>10 \times$ Xeon nodes ($375 \times$ CPU cores). The FPGA GEMM can thus *outperform a small cluster* of dual-socket CPUs, and offers considerable speedup even at small matrix sizes. Introducing more compute units to a fixed size problem (strong scaling along a vertical line in Fig. 5) reduces the amount of work *per compute unit*, resulting in

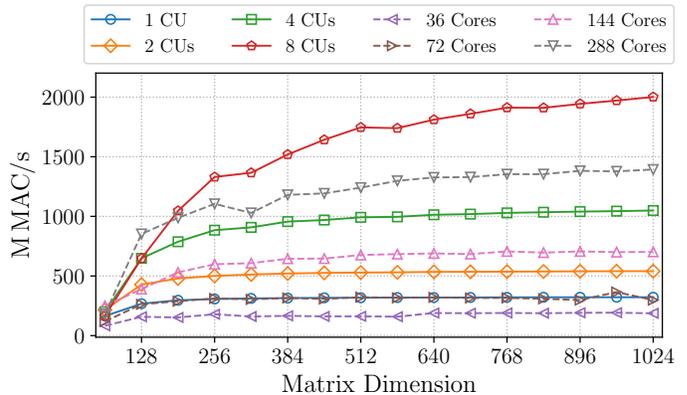


Fig. 5: Multiply-addition performance multiplying two matrices of size $n \times n$ with 448-bit mantissas (512 bits total).

Precision	CUs	Frequency	CLBs	DSPs	Max. Performance
512 (448)	1	327 MHz	18.9%	4.5%	322 MMAC/s
512 (448)	2	278 MHz	31.7%	9.0%	540 MMAC/s
512 (448)	4	278 MHz	46.6%	14.4%	1049 MMAC/s
512 (448)	8	293 MHz	65.8%	35.8%	2002 MMAC/s

TABLE III: Overview of 512-bit GEMM designs.

more replications requiring larger matrix inputs to reach peak performance. An overview of all designs evaluated is shown in Tab. III, including their logic utilization and the highest performance achieved across different matrix sizes. Although there is still some resource headroom, further replication is prevented by the number of DDR4 memory interfaces available on the shell used.

D. Extending Matrix Multiplication to 1024 bits

Extending the matrix multiplier to 1024 bit APFP numbers introduces additional challenges on the target FPGA platform, as a single 1024-bit matrix multiplication compute unit occupies nearly a full SLR on the U250 chip. Based on the results for 512-bit multiplication, two or three 1024-bit multipliers should fit on the device, as this roughly corresponds to six or nine 512-bit multipliers (since each level of Karatsuba

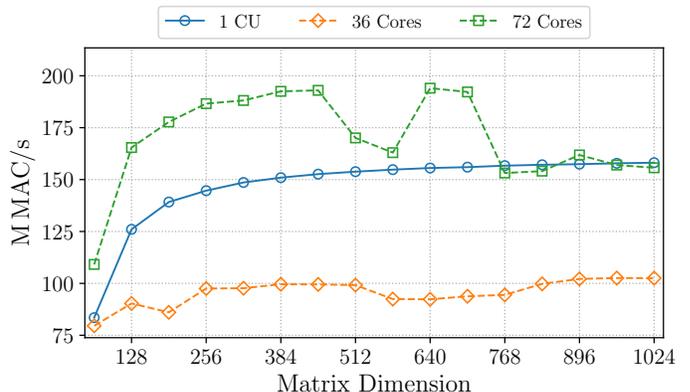


Fig. 6: Multiply-addition performance multiplying two matrices of size $n \times n$ with 960-bit mantissas (1024 bits total)

decomposition requires 3 half-width multipliers), respectively. However, since these subcomponents are no longer independent and are scheduled as a single pipeline, they are scheduled in a monolithic manner.

We include a preliminary result for 1024-bit (960-bit mantissa) matrix multiplication in Fig. 6 for a single compute unit. Due to excessive congestion within the multiplication pipeline, the design is downclocked to 212 MHz. The throughput at this frequency exceeds the performance of Elemental executed on a 36-core Xeon node, with a peak throughput of 158 MMAC/s. At 29.8% CLB utilization, we expect that a more appropriately floorplanned design would allow instantiating 4 compute units.

VI. RELATED WORK

Various previous work has proposed accelerators for APFP arithmetics. CAMPARY [36] accelerates up to 424 bits of mantissa using CUDA. The authors show up to $19\times$ speedup on a Fermi-based Tesla C2075 GPU over a consumer-grade quad-core Sandy Bridge CPU running MPFR, dropping to $\sim 1\times$ for 424-bit mantissas. MPRES-BLAS [37] presents GPU acceleration of APFP dense linear algebra, showing $\sim 2\times$ speedup over CAMPARY for GEMM, reporting $\sim 100\text{--}120$ MOp/s for 424-bit precision on a GTX 1080 GPU. Lei et al. [38] implement an APFP accelerator on a Virtex 6 FPGA and report $11.6\times$ speedup for 1024-bit multiplication over MPFR running on a dual-core Core i3 530 Clarkdale-based CPU. Lu et al. [39] accelerate 500-2000 digits of precision on a GTX 280 GPU on the Tesla architecture and compare it to a quad-core Kentfield CPU running ARPREC [40], reporting $8\text{--}9\times$ speedup on multiplication. As of writing, the source code published by the authors has not been updated to support modern GPUs. Common for the above work is that comparisons are made to consumer-grade CPUs, which lack the core count of the server-grade CPUs that are typically employed for larger-scale numerical workloads. Furthermore, Broadwell-based CPUs and onwards received support for the Intel ADX instruction set in addition to BMI2 introduced with Haswell, which significantly increases CPU performance on arbitrary precision workloads. Chow et al. [41] implement a Montgomery multiplier for modular arithmetic based on Karatsuba decomposition. The authors estimate that 400 MOp/s of Montgomery multiplication throughput is achievable on a Virtex-6 FPGA based on synthesis results, but do not build their design for execution in hardware.

Based on the results presented in this work, our FPGA-based accelerator outperforms all the above accelerators in terms of absolute throughput in hardware, and in terms of speedup when executed in hardware relative to server-grade CPUs of each corresponding generation of hardware at the time of their publication. Furthermore, our work is published as a configurable HLS-based implementation, which can dynamically scale performance by replicating compute units, and compiles for any Vitis/XRT-based Xilinx platform.

VII. CONCLUSION

In this work, we showed how FPGAs provide an excellent platform for accelerating fundamental operators for arbitrary precision floating point (APFP) arithmetic. We present a deeply pipelined design implementing APFP multiplication using a Karatsuba decomposition bottoming out at naive multiplication in DSPs, yielding a multiplication throughput of up to 4.8 GOp/s for 512-bit and 1.2 GOp/s for 1024-bit numbers on an Alveo U250 accelerator, corresponding to the throughput of more than $351\times$ and $191\times$ CPU cores running MPFR, respectively. We combine the multiplier with our APFP adder to perform general matrix-matrix multiplication in hardware, showing 2.0 GMAC/s on 512-bit numbers, which corresponds to the throughput of more than $375\times$ CPU cores, matching the performance of a 10-node Xeon cluster. For numerical codes that are dominated by arbitrary precision arithmetic, such as semidefinite program (SDP) solvers, we expect these gains to translate into real-world speedups on applications such as the conformal bootstrap studying phase transitions in quantum field theory. To facilitate this, we publish the accelerator code as an open-source HLS-based project, configurable for any Vitis/XRT-supported Xilinx FPGA. We provide a plug-and-play software interface that can be dropped into existing numerical codes, allowing scientists to tap into FPGA acceleration of APFP with minimal code changes.

ACKNOWLEDGMENTS

 This project has received funding from the European Research Council (ERC) under the European Union's Horizon 2020 research and innovation programme grant agreement no. 101002047 and from the European High-Performance Computing Joint Undertaking (JU) under grant agreement no. 101034126. Christopher A. Pattison is supported by Air Force Office of Scientific Research (AFOSR), FA9550-19-1-0360, and thanks Dustin Kenefake for inspiring discussions. David Simmons-Duffin is supported by Simons Foundation grant 488657 (Simons Collaboration on the Non-perturbative Bootstrap) and a DOE Early Career Award under grant no. DE-SC0019085.

REFERENCES

- [1] T. Granlund, "GNU MP," *The GNU Multiple Precision Arithmetic Library*, 1996.
- [2] L. Fousse, G. Hanrot, V. Lefèvre, P. Pélicier, and P. Zimmermann, "MPFR: A multiple-precision binary floating-point library with correct rounding," *ACM Transactions on Mathematical Software (TOMS)*, vol. 33, no. 2, pp. 13–es, 2007.
- [3] L. Vandenberghe and S. Boyd, "Semidefinite programming," *SIAM Review*, vol. 38, no. 1, pp. 49–95, 1996.
- [4] S. Boyd, L. El Ghaoui, E. Feron, and V. Balakrishnan, *Linear Matrix Inequalities in System and Control Theory*. Society for Industrial and Applied Mathematics, 1994.
- [5] F. Alizadeh, "Interior point methods in semidefinite programming with applications to combinatorial optimization," *SIAM Journal on Optimization*, vol. 5, no. 1, pp. 13–51, 1995.
- [6] J. D. Hauenstein, A. C. Liddell, S. McPherson, and Y. Zhang, "Numerical algebraic geometry and semidefinite programming," *Results in Applied Mathematics*, vol. 11, p. 100166, 2021.
- [7] H. Wolkowicz, R. Saigal, and L. Vandenberghe, *Handbook of Semidefinite Programming: Theory, Algorithms, and Applications*. Boston, MA: Springer US, 2000.

- [8] M. Nakata, "A numerical evaluation of highly accurate multiple-precision arithmetic version of semidefinite programming solver: SDPA-GMP, -QD and -DD." in *2010 IEEE International Symposium on Computer-Aided Control System Design*, 2010, pp. 29–34.
- [9] M. Joldes, J.-M. Muller, and V. Popescu, "Implementation and performance evaluation of an extended precision floating-point arithmetic library for high-accuracy semidefinite programming," in *2017 IEEE 24th Symposium on Computer Arithmetic (ARITH)*, 2017, pp. 27–34.
- [10] M. Garstka, M. Cannon, and P. Goulart, "COSMO: A conic operator splitting method for convex conic problems," *Journal of Optimization Theory and Applications*, vol. 190, no. 3, pp. 779–810, 2021.
- [11] D. Simmons-Duffin, "A semidefinite program solver for the conformal bootstrap," *Journal of High Energy Physics*, vol. 2015, no. 6, 2015.
- [12] W. Landry and D. Simmons-Duffin, "Scaling the semidefinite program solver SDPB," 9 2019.
- [13] D. Simmons-Duffin, "The Conformal Bootstrap," in *Theoretical Advanced Study Institute in Elementary Particle Physics: New Frontiers in Fields and Strings*, 2017, pp. 1–74.
- [14] S. M. Chester, "Weizmann Lectures on the Numerical Conformal Bootstrap," 7 2019.
- [15] D. Poland, S. Rychkov, and A. Vichi, "The conformal bootstrap: Theory, numerical techniques, and applications," *Rev. Mod. Phys.*, vol. 91, p. 015002, Jan 2019.
- [16] R. Rattazzi, V. S. Rychkov, E. Tonni, and A. Vichi, "Bounding scalar operator dimensions in 4D CFT," *JHEP*, vol. 12, p. 031, 2008.
- [17] D. Poland, D. Simmons-Duffin, and A. Vichi, "Carving Out the Space of 4D CFTs," *JHEP*, vol. 05, p. 110, 2012.
- [18] N. Afkhami-Jeddi, H. Cohn, T. Hartman, and A. Tajdini, "Free partition functions and an averaged holographic duality," *JHEP*, vol. 01, p. 130, 2021.
- [19] M. F. Paulos, J. Penedones, J. Toledo, B. C. van Rees, and P. Vieira, "The S-matrix bootstrap II: two dimensional amplitudes," *JHEP*, vol. 11, p. 143, 2017.
- [20] S. Caron-Huot and V. Van Duong, "Extremal Effective Field Theories," *JHEP*, vol. 05, p. 280, 2021.
- [21] J. Bonifacio and K. Hinterbichler, "Bootstrap Bounds on Closed Einstein Manifolds," *JHEP*, vol. 10, p. 069, 2020.
- [22] P. Kravchuk, D. Mazac, and S. Pal, "Automorphic Spectra and the Conformal Bootstrap," 11 2021.
- [23] Y. Umuroglu, N. J. Fraser, G. Gambardella, M. Blott, P. Leong, M. Jahre, and K. Vissers, "FINN: A framework for fast, scalable binarized neural network inference," in *Proceedings of the 2017 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays (FPGA'17)*, 2017, pp. 65–74.
- [24] S. Wang and P. Kanwar, "BFloat16: The secret to high performance on cloud TPUs," Google, 2019, accessed on January 16, 2022. [Online]. Available: <https://cloud.google.com/blog/products/ai-machine-learning/bfloat16-the-secret-to-high-performance-on-cloud-tpus>
- [25] B. Rouhani, D. Burger, E. Chung, R. Majumder, S. Shekar, S. Tiwary, S. Lanka, and S. Reinhardt, "A Microsoft custom data type for efficient inference," 2020, accessed on January 16, 2022. [Online]. Available: <https://www.microsoft.com/en-us/research/blog/a-microsoft-custom-data-type-for-efficient-inference/>
- [26] A. A. Karatsuba and Y. P. Ofman, "Multiplication of many-digit numbers by automatic computers," in *Doklady Akademii Nauk*, vol. 145, no. 2. USSR Academy of Sciences, 1962, pp. 293–294.
- [27] A. Toom, "The complexity of a scheme of functional elements realizing the multiplication of integers," in *Doklady Akademii Nauk*, vol. 3, no. 4. USSR Academy of Sciences, 1963, pp. 714–716.
- [28] S. A. Cook, "On the minimum computation time of functions," Ph.D. dissertation, Harvard University, 1966.
- [29] A. Schönhage and V. Strassen, "Schnelle Multiplikation grosser Zahlen," *Computing*, vol. 7, no. 3, pp. 281–292, 1971.
- [30] D. Harvey and J. Van Der Hoeven, "Integer multiplication in time $O(n \log n)$," *Annals of Mathematics*, vol. 193, no. 2, pp. 563–617, 2021.
- [31] D. Vandevoorde and N. M. Josuttis, *C++ Templates: The Complete Guide*. Addison-Wesley Professional, 2002.
- [32] J. de Fine Licht, G. Kwasniewski, and T. Hoefler, "Flexible communication avoiding matrix multiplication on FPGA with high-level synthesis," in *Proceedings of the 2020 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays (FPGA'20)*, 2020, pp. 244–254.
- [33] J. de Fine Licht and T. Hoefler, "hlslib: Software engineering for hardware design," *arXiv:1910.04436*, 2019.
- [34] M. Nakata, "MPLAPACK version 1.0.0 user manual," 2021.
- [35] J. Poulson, B. Marker, R. A. Van de Geijn, J. R. Hammond, and N. A. Romero, "Elemental: A new framework for distributed memory dense matrix computations," *ACM Transactions on Mathematical Software (TOMS)*, vol. 39, no. 2, pp. 1–24, 2013.
- [36] M. Joldes, J.-M. Muller, V. Popescu, and W. Tucker, "CAMPARY: CUDA multiple precision arithmetic library and applications," in *International Congress on Mathematical Software (ICMS)*. Springer, 2016, pp. 232–240.
- [37] K. Isupov and V. Knyazkov, "Multiple-precision blas library for graphics processing units," in *Supercomputing*, V. Voevodin and S. Sobolev, Eds. Cham: Springer International Publishing, 2020, pp. 37–49.
- [38] Y. Lei, Y. Dou, and J. Zhou, "FPGA-specific custom VLIW architecture for arbitrary precision floating-point arithmetic," *IEICE TRANSACTIONS on Information and Systems*, vol. 94, no. 11, pp. 2173–2183, 2011.
- [39] M. Lu, B. He, and Q. Luo, "Supporting extended precision on graphics processors," in *Proceedings of the Sixth International Workshop on Data Management on New Hardware (DaMoN'10)*, 2010, pp. 19–26.
- [40] D. H. Bailey, H. Yozo, X. S. Li, and B. Thompson, "ARPREC: An arbitrary precision computation package," 2002.
- [41] G. C. Chow, K. Eguro, W. Luk, and P. Leong, "A Karatsuba-based Montgomery multiplier," in *IEEE 20th International Conference on Field Programmable Logic and Applications (FPL'10)*. IEEE, 2010, pp. 434–437.