GPU-Accelerated High-Level Synthesis for Bitwidth Optimization of FPGA Datapaths

***Abstract*—Bitwidth optimization of FPGA datapaths can save hardware resources by choosing the fewest number of bits required for each datapath variable to achieve a desired quality of result. However, it is an NP-hard problem that requires unacceptably long runtimes when using sequential CPU-based heuristics. We show how to parallelize the key steps of bitwidth optimization on the GPU by performing a fast brute-force search over a carefully constrained search space. We develop a high-level synthesis methodology suitable for rapid prototyping of bitwidth-annotated RTL code generation using `gcc`'s `GIMPLE` backend. For range analysis, we perform parallel evaluation of sub-intervals to provide tighter bounds compared to ordinary interval arithmetic. For bitwidth allocation, we enumerate the different bitwidth combinations in parallel by assigning each combination to a GPU thread. We demonstrate up to 10–1000× speedups for range analysis and 50–200× speedups for bitwidth allocation when comparing NVIDIA K20 GPU implementation to an Intel Core i5-4570 CPU while maintaining identical solution quality across various benchmarks. This allows us to generate tailor-made RTL with minimum bitwidths in hundreds of milliseconds instead of hundreds of minutes when starting from high-level C descriptions of dataflow computations.**

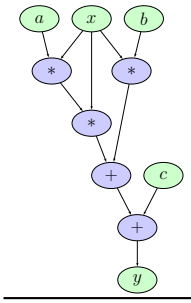
I. Introduction

FPGAs have long empowered the circuit designer to build hardware tailored completely to the particular performance, cost and accuracy requirements of the application. A key aspect of this flexibility is the ability to select the data representation for signals in the circuit. As different applications have varying accuracy requirements, the exact number of bits necessary may change. This is in stark contrast to ISA-based processors where a small set of types are supported (char, short, int, long, float, double). While the low-level tuning of bits may seem time-consuming and excessive, it is a well-known approach [8], [13], [20], [21], [2], [1], [22] for saving resource costs by as much as 2–4× while improving circuit performance by as much as 20–30% without compromising quality of the output signals. In particular, with the promise of approximate computing, we must ask how many bits are actually needed to represent data (or signals) to deliver an acceptable computational outcome for our designs. The objective of the bitwidth optimization problem is to determine the cheapest data representation (fewest bits) for all variables in the computation subject to user-supplied accuracy constraint. This is achieved through static compile-time analysis of variable bounds and error.

While the benefits are clear, unfortunately, bitwidth allocation is an NP-hard problem [8]. Broadly speaking, we can classify existing approaches for solving this problem based on the focus of their optimization: (1) tighter analysis of error [2], [1], [14], [15] that focuses on fast arithmetic techniques for producing tight error bounds; (2) intelligent search algorithms [13], [20], [21], [10] that use specially-formulated automated heuristics. To perform optimized bitwidth allocation on CPUs, all heuristics typically examine a limited set of bitwidth combinations during the search process to keep runtimes low. More importantly, these approaches solve the problem in a sequential manner by refining the bitwidth combination in each sequential iteration while learning from the previous trials.

Commercial tools such as Matlab HDL Coder provide a floating-point to fixed-point conversion toolflow but exposes the accuracy analysis to a simulation-driven workflow that requires developer involvement. LegUp [7] provides no precision analysis engine while Vivado HLS merely allows expression of templated types for fixed-point arithmetic without providing the necessary automation to analyze error and select bitwidths. A key limitation that prevents integration of automated analysis engines in these tools is the large computational cost of the automation which translates into minutes to hours of runtime for even simple dataflow blocks of code. In fact, pure CPU-based fixed-point simulations to determine bitwidths of a simple FIR filter with 10^5-element-long input test vector can take 40 days [4] of runtime. In this paper, we provide an automated precision analysis engine for `gcc` based on `GIMPLE` backend. We also show how to speedup the optimization problem using GPUs with a carefully constrained brute-force approach backed by an intelligent pruning of the search space. The use of GPUs in the FPGA CAD process needs to expand and grow to help tackle the series of slow, NP-hard heuristics that have traditionally constrained the design development process. The bitwidth optimization problem is one such slow heuristic that needs faster evaluation for broader adoption and seamless integration with high-level synthesis (HLS). The key enabling idea here is rapid evaluation of dynamic range, error and resource costs estimates for multiple bitwidth combinations in data-parallel fashion. By choosing *where* to explore and by evaluating multiple candidate solutions in parallel, we can accelerate the time to optimized solution. Unlike multi-core CPUs, GPUs are well-suited for this kind of brute-force exploration as they support hundreds of thousands of concurrent threads. We formulate our search space in a manner that fits entirely within the GPU memory capacity while maximizing the likelihood of finding the optimal solution fast. We develop heuristics that prune the search space to bound variable bit width ranges based on error propagation. For large problem sizes, we use simulated annealing-based heuristics running on the CPU to constrain the search space before transferring the

TABLE I: Different Phases of the Precision Analysis Flow for $y = a * x^2 + b * x + c$.
Assume a, b, c are integer constants, W_i represents the number of bits required to represent variable i

Dataflow Graph (DFG)	DFG Node	Range Analysis	Error Propagation	Resource Modeling
	x	[0, 1]	$2^{(-W_x-1)}$	W_x
	a	$[a, a]$	0	W_a
	b	$[b, b]$	0	W_b
	c	$[c, c]$	0	W_c
	$a * x$	$[0, a]$	$\max(a * 2^{(-W_x-1)}, 2^{-W_{m1}}))$	$W_a * W_x$
	$b * x$	$[0, b]$	$\max(b * 2^{(-W_x-1)}, 2^{-W_{m2}}))$	$W_b * W_x$
	$a * x^2$	$[0, a]$	$\max(a * 2^{-W_x}, 2^{-W_{m3}}))$	$W_a * {W_x}^2$
	$a * x^2 + b * x$	$[0, a + b]$	$\max((a + b/2) * 2^{-W_x}, 2^{-W_{a1}}))$	$\max(W_a * {W_x}^2, W_b * W_x)$
	$a * x^2 + b * x + c$	$[c, a + b + c]$	$\max((a + b/2) * 2^{-W_x}, 2^{-W_{a2}}))$	$\max(W_a * {W_x}^2, W_b * W_x, W_c)$

computation to the GPU for acceleration. This proposal is in the same spirit as end-case optimal layout algorithms for ASICs [5], where the final branch of the design space is solved optimally while the parent heuristic is still retained.

In this paper, we make the following key contributions:

- Development of a high-level synthesis flow suitable for rapid prototyping of bitwidth optimization transforms that compile C descriptions of dataflow computations into RTL by exploiting `gcc`'s `GIMPLE` backend.
- Development of optimized GPU-based kernels for (1) accelerated sub-interval analysis to derive higher dynamic range bounds, and (2) parallel error analysis and FPGA resource model kernels for bitwidth optimization.
- Engineering of CPU-based pruning heuristics that intelligently constrain the search to make it feasible to perform brute-force exploration on the GPU.
- Quantification of speed and quality of bitwidth optimization when comparing the NVIDIA K20 GPU to state-of-the-art simulated annealing-based heuristics on an Intel Core i5-4570 CPU across a variety of benchmarks.

II. Background

Precision analysis is typically performed before detailed RTL design and pipelining. We must determine the exact number of bits required for various datapath variables and encode these into the RTL descriptions of our computation. This static analysis procedure can be automated and potentially integrated with high-level synthesis to generate tailor-made RTL for a given accuracy constraint. We now describe the basic concepts underlying precision analysis that are important to explain our parallelization approach. The precision analysis flow for bitwidth optimization can be broken down into a sequence of three steps (1) range analysis, (2) error propagation, and (3) resource modeling. We will use the example expression $a \cdot x^2 + b \cdot x + c$ to illustrate this flow as shown in Table I.

Range Analysis: We need to first identify the dynamic range of all variables in our computation which requires calculating the smallest and the largest values they may attain. This is essential as the implemented circuit only needs to be correct over this range which is typically much smaller than the range of real numbers (or floating-point for canonical implementations). The underlying algorithm to achieve this simply propagates the interval of the input iteratively through the feed-forward dataflow computation while evaluating the range at the output of each arithmetic operator. For example, in the polynomial expression of Table I (column Range Analysis), if $x \in [0, 1]$, then $y \in [c, a + b + c]$ can be computed through a top-down range propagation using Interval Arithmetic (IA). However, purely relying on IA often produces bounds that are over-estimated (loose), because they ignore input correlations. Loose bounds affect our final solution quality by requiring more bits than strictly necessary to meet the error constraints. One option is to use Affine Arithmetic (AA) which allows us to produce accurate estimates of intervals by identifying correlations in the inputs at the expense of tracking correlation terms that are proportional to the number of inputs to the expression (thereby trickier for parallelism). However, Affine Arithmetic (AA) formulations are trickier to parallelize on GPUs due to the growth in correlation terms. In this paper, we choose to use the well-known sub-interval analysis approach to addresses this limitation of traditional IA. This is achieved by splitting each input interval into multiple sub-intervals, performing range analysis on all input sub-interval combinations, and composing the final range by merging the results of each sub-interval evaluation. This allows us to trivially exploit parallel processing capacity on the GPU across sub-intervals even while an individual sub-interval is sequentially evaluated. Our formulation using sub-interval arithmetic helps us generate tight bounds while exposing GPU-friendly parallel behavior.

Error Propagation: Once we know the range of each variable, we can estimate rounding and truncation errors due to each arithmetic operation. These errors depend on the number of fraction bits used in the variable representation. We assume sufficient number of integer bits are allocated for the largest fixed-point number to eliminate overflow errors. While many error models are available such as mean squared error (MSE) in signal processing systems, we use the worst-case quantization error models for fixed-point implementations that are evaluated over operating intervals of the variable. For our implementation, we exact and verify the error model

equations as expressed in the error analysis tool Gappa [3]. The errors are propagated from the input to output of the arithmetic expression based on the number of bits for each arithmetic operation as shown in Table I (column Error Propagation). Here, we observe that each arithmetic operation introduces an error term that is then carried forward downstream in the expression tree. Structurally this is similar to range propagation, but here we need to repeat this analysis for each proposed bitwidth combination. For a given combination, this is a sequential operation that processes error values from the inputs and calculates output error of that operation. CPU-based analysis tools are typically driven by heuristics such as simulated annealing that iteratively generate bitwidth combinations for evaluation and terminate once a sufficiently low-cost solution is found. Our insight here is that we can generate and evaluate multiple precision combination proposals for the variables in the arithmetic expression and process them in parallel. Again, this form of large-scale exploration is inappropriate on multi-core platforms due to the limited number of threads that are possible and the large scope of the combinations that must be considered. However, GPU-based systems with thousands of lightweight data-parallel threads are ideally suited to exploit this parallel pattern.

Resource Modeling: In this paper, our optimization problem is geared towards minimization of resource utilization of the implemented circuit. The goal is flexible, and we can substitute resource minimization for speed, latency or power if required (or some combined figure of merit). To support resource minimization, we develop highly-accurate resource models of arithmetic operators by performing a complete FPGA implementation flow (synthesis, place and route). This accounts for internal optimizations and interactions between the various FPGA CAD stages. Based on these compilations, we build regression-fit resource models for LUT and DSP count usage as a function of the number of bits in the arithmetic operation using Weka [1], a popular data mining toolkit. This approach is in stark contrast to the simplistic models in [8], [13], [20], [21], [2], [1] and helps improve our optimization accuracy. For a given precision combination, this computation is a simple accumulation of cost estimates for each operation as shown in Figure I (column Resource Modeling). For all precision combinations processed in parallel, we filter those that satisfy a user-supplied error constraint, and calculate the cheapest cost implementation using a GPU-optimized `reduce` operation.

III. GPU Acceleration

Bitwidth optimization on sequential CPUs is typically performed using heuristics such as simulated annealing (SA) that help drive the optimization towards a final solution while avoiding getting trapped in local minima through hill climbing. The state-of-the-art precision analysis tools [13], [20], [22] use this approach for formulating and tuning the optimization. They use the Adaptive Simulated Annealing (ASA) [9] package which is an highly-tuned optimization software that uses SA with intelligent selection of solution candidates to approach the optimal solutions. We use this state-of-the-art tool as reference for comparing time and solution quality.

[1] http://www.cs.waikato.ac.nz/ml/weka/

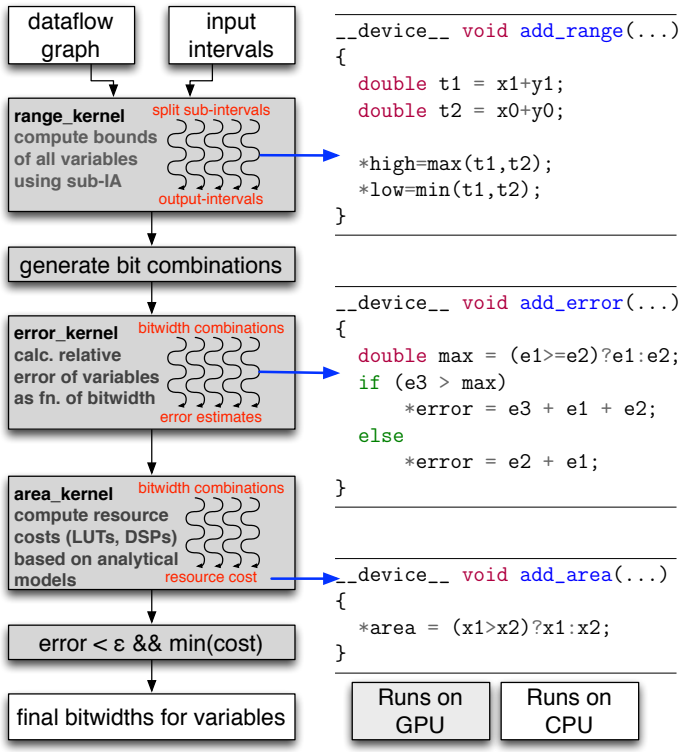


Fig. 1: Compute Flow for GPU-Accelerated Precision Analysis (with sample GPU code for `add` operator)

In this section, we discuss the design of the key kernels used in our GPU approach that implements this search in a different manner. Unlike sequential CPU implementations of ASA, where a single combination of bitwidths is considered in a given iteration, on the GPU, we exploit the opportunity to consider multiple combinations in parallel.

A. GPU Kernel Design

The parallel bitwidth allocation problem on GPUs is implemented across three kernels (1) sub-interval range analysis, (2) error propagation, and (3) resource modeling calculations. We show a high-level view of the flow in Figure 1 for the `add` operator. Each of these three kernels is data-parallel. Sub-interval Analysis is parallelized by trivially distributing the sub-interval combinations across the GPU threads while the Error propagation and Resource Modeling phases are parallelized across multiple bitwidth combinations that are considered. These approaches are *brute-force* approaches and are ultimately limited by the number of parallel threads that can run on your GPU and the amount of DRAM space available to store all intermediate results. With the availability of GPUs that can process thousands of threads in parallel and store GBs of data in the DRAM (*e.g.* NVIDIA K20 can handle ≈26K threads and 5–6 GB of RAM), we can start tackling

brute-force approaches to optimization problems with relative ease.

For range analysis, each GPU thread is programmed to evaluate a small sub-interval slice of the complete input range. For multiple inputs, this means we must consider all possible combinations of input sub-intervals across all threads. Each thread uses a unique thread index to automatically identify the sub-interval it is supposed to explore. An individual GPU thread propagates the sub-intervals through the graph to calculate the sub-ranges of the intermediate and output variables. Once this is done, we fuse the results of the multiple sub-intervals to form complete tight bounds on all variables in the program. While this procedure is highly-parallel, we are ultimately limited by the memory capacity required to hold all intermediate sub-ranges. For error analysis, we allocate each GPU thread to evaluate one combination of bitwidths. Each thread locally determines the combination of bitwidths it must explore based on the limit of precision choices possible for all variables and the unique thread index. We discuss how the precision limits for the variables are identified shortly in Section III-B. Given a bitwidth combination, a thread can process the error equations and resource model expressions to determine the cumulative error of the output variable and resource costs. Finally, across all threads, we filter out those threads that do not satisfy the user-supplied error constraint. From this set of threads (precision combinations), we pick the thread with the cheapest implementation cost as the solution.

When developing our GPU code for arithmetic operators we derive our error models from [19] and [3], except we consider rounding to the nearest while they use truncation directly. Furthermore we adapt the models to better account for constant and integer inputs. For area modeling of the arithmetic operators, instead of using analytical models based on theoretical expectations as used in [13], and [21], we use regression-fitted models derived from the real-world results of a complete FPGA CAD flow to capture the effect of CAD tools on implementation costs (LUTs and DSPs). In this regard, our approach requires a first-pass evaluation and mapping of all basic arithmetic building blocks through the FPGA CAD flow to help expose the DSP-LUT partitioning and allocation of resources that may differ from a simplistic theoretical models used in [13], and [21]. We quantify the impact of this accurate assessment of resource costs in Figure 4 later to show the resulting benefits. In these experiments, we observed a 30% improvement in prediction accuracy when considering the post-synthesis mapping results to select bitwidths.

In Table II, we show the preliminary performance results for the range, error and area evaluation kernels on the GPU across various arithmetic operations. We observe 100-400× faster processing on the GPU when compared to OpenMP-optimized, multi-threaded (8-thread) implementation on the CPU. This initial experiment suggests high parallel potential for GPU evaluation of the constituent kernels. As we scale to larger GPUs, we expect this gap over multi-cores to grow as multi-core CPU trends do not align with the sheer degree of data-parallel thread capacity of modern GPU architectures and their rising DRAM capacity and associated bandwidths.

TABLE II: NVIDIA K20 GPU Speedups over Intel Core i5-4570 CPU for Kernels used in Precision Analysis

Kernel	**Speedup**				
	add	mult	div	exp	log
Range analysis	312×	213×	119×	298×	254×
Error propagation	246×	80×	103×	261×	297×
Resource estimation	272×	266×	251×	414×	407×

B. Search Space Pruning

Instead of using sequential ASA-like heuristics directly on the GPU, we develop a GPU-optimized solution that uses a brute-force approach to explore multiple combinations in parallel. However, a naive brute force approach will quickly exhaust available GPU memory capacity, limiting the usefulness of this technique to toy problems. To allow the search space to fit within the memory limits of the GPU, we develop a pruning heuristic that restricts the combinations considered during the search. The pruning is also necessary for smooth execution of ASA running on the CPU. For our benchmark set introduced later in Section V-A, ASA without pruning simply fails to converge and runs for hours before aborting. On the GPU, what pruning allows us to do is to optimally solve the subspace extracted from the pruning, unlike the multi-core CPUs which still explore this space sub-optimally based on heuristics. We show a high-level sketch of our pruning heuristic in Algorithm 1. Bitwidth combinations that result in error larger than the user-supplied error constraint are invalid. Our pruning heuristic must maximize the coverage of valid bitwidth combinations in our search. At the start, we choose a uniform fixed point bitwidth (*target_fb* in line 1 in Algorithm 1) for all variables in the code and keep increasing precision until we satisfy the user-supplied error constraint (line 2–4 in Algorithm 1). The uniform bitwidth is designed to satisfy the worst case error and forces all variables to use the corresponding worst case precision. Thus, the uniform bitwidth $uniform\_bit$ will be over-provisioned for some variables that could be implemented with fewer bits. Hence, we now decrease the bitwidth one variable at a time (line 6–12 in Algorithm 1) while keeping the other bitwidths equal to $uniform\_bit$. By doing this across all variables, we get the lowest possible precision ($lowest$) for each variable independently. We now replace each variable's precision with their respective $lowest$ but in this configuration we will most likely violate the error constraint. As the last step, we increase the precisions of all variables simultaneously one bit at a time (line 14–15 in Algorithm 1) until it meets the required error criteria once again. In some instances, this pruning is excessively aggressive, and we relax the constraints by adding extra padding bits (*guard_bit* in line 17 in Algorithm 1) to help cover potentially better solutions. Overall, our heuristic is able to compress the number of potential precision choices of each variable into ranges that make them feasible for brute-force exploration within the GPU memory space.

Algorithm 1: Search Space Pruning Heuristic

Data: The number of variables N; Targeted Fixed-point Precision
Result: Bounded search space

```
bit_width(0:N-1) ← target_fb;
while current_error > error_constraint do
    bit_width(0:N-1) ++;
end
uniform_bit = bit_width[0];
foreach n=0:N-1 do
    while current_error ≤ error_constraint do
        bit_width(n)−−;
    end
    lowest(n)← bit_width(n);
    bit_width(n)← uniform_bit;
end
bit_width(0:N-1) ← lowest(0:N-1);
while current_error ≤ error_constraint do
    bit_width(0:N-1)++;
end
highest(0:N-1) ← bit_width(0:N-1) + guard_bit;
```

For very large problems, this pruning is still insufficient to make the design feasible for brute-force exploration on the GPU. In these cases, we rely on preliminary cost-reducing moves in sequential ASA to assist in the pruning process. In this arrangement, we split the computation between CPU and GPU by allowing the first 10–100 iterations to run sequentially on the CPU and then switch to the GPU once the search space becomes feasible for GPU exploration. We keep the lower bound of the possible bitwidth identical as before and use the best observed precision combinations as the new upper bound for search. We achieved identical solution quality as the CPU-only solution after $\approx$ 25 iterations.

C. Overcoming GPU Limits

While the GPU is a great platform for rapid parallel evaluation of data-parallel problems, we consider several architectural optimizations that enhance performance of the FPGA CAD computations.

GPU Global Memory Capacity: A key constraint to consider is the size of the search space. For instance, on the NVIDIA K20 GPU with a 5GB DRAM, we are able to explore search spaces with precision combinations as large as 2^{24} with a single GPU call. Similarly we were only able to scale from 1K–8K sub-intervals before we run out of memory resources to store intermediate sub-ranges.

Kernel Fusion: The GPU kernels for error propagation and resource modeling are logically separate and can be invoked independently. However, we achieve significant reductions in kernel invocation and synchronization time if we fuse them together into a single call.

CPU-GPU Offload: While it is tempting (and easy) to offload all data-parallel computations to the GPU, in our case, we saw significant degradation in performance of the error kernel when evaluating truncation error of the form 2^{-t} using the `pow` CUDA function. Since the possible values of precision we consider are limited ($t < 128$ bits), we precompute these values offline on the CPU only once and simply pass them to the kernels as a lookup table.

Memory Bandwidth: To avoid needless data transfers between the CPU and GPU, we allocate sub-intervals, error and resource arrays directly in the GPU main memory. We only need to access summarized data, such as the unified final interval and the minimum cost implementation result from these large arrays. Furthermore, we identify thread-local state for storing per-variable structures into fast memory (registers and shared memory) instead of allocating them on the GPU global memory space. When we stored intermediate results in GPU shared memory space we were able to minimize needless memory traffic to the off-chip DRAM and obtain additional $\approx 2\times$ speedup across our kernels.

IV. High-Level Synthesis Toolflow

While LegUp [7] is a popular contemporary choice for development of HLS compiler transformations, we choose an alternative methodology using `gcc`. The LLVM-based [12] LegUp license specifically restricts the use of the open-source compiler to non-commercial, not-for-profit development [2]. Instead, we are able to rapidly prototype our optimization algorithms based on the `GIMPLE` backend in the free GPLv3-licensed [3] `gcc` compiler, while retaining all the existing optimization benefits of the compiler flow (our tool will inherit this license). Transformations described on the three-register `GIMPLE` syntax can be developed independently of the compiler and need not be constrained to the language or development quirks of the compiler framework itself.

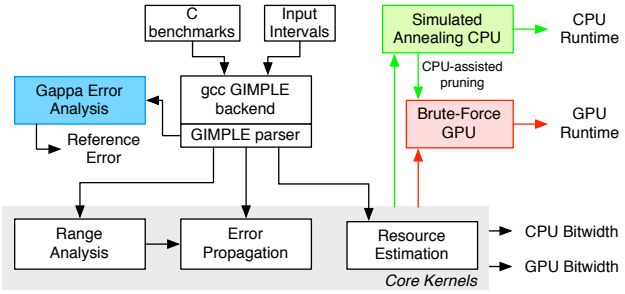


Fig. 2: Compilation Flow for Precision Analysis

In Figure 2, we show a high-level block diagram of our compiler flow. We support simple feed-forward computations written in C through the `gcc`'s `GIMPLE` [6] backend. It is an Intermediate Representation (IR) provided to support development of plugins and optimization passes. For precision analysis, we process the C input files along with user-annotated range information to generate intermediate `GIMPLE` IR for post-processing. We then translate the `GIMPLE` IR into a suitable assembly-like, dataflow format for GPU processing that captures the dependencies and range information in a compact data structure. We develop a generic interpreter on the GPU that iterates over this suitably encoded dataflow graph. This not only allows us to avoid needless recompilations on the

[2] http://legup.eecg.utoronto.ca/license.php
[3] https://www.gnu.org/licenses/licenses.html

GPU for each benchmark input, but enables optimized GPU kernel design that works across all benchmarks.

We verify the correctness of our GPU-calculated error bounds by generating Gappa scripts for the given benchmark. Once again, the simplicity of the `GIMPLE` IR makes it feasible to rapidly assemble a translator for Gappa syntax. Gappa [14] is a static analysis tool that proves numerical properties of programs through formal techniques. Our CPU-based ASA annealer also operates on the exact same `GIMPLE` IR and the exact same error and resource models as the GPU implementation for a fair comparison.

TABLE III: Auto-generated Assembly and Gappa Code

(a) ASM Code	(b) Gappa Script
LD, 0, 0, 1	x_fx = fx1 (x); a_fx =fx2 (a);
LD, 1, 1, 1;	b_fx =fx3 (b); c_fx =fx4 (c);
LD, 2, 1, 1;	d_fx =fx5 (a_fx * x_fx);
LD, 3, 1, 1;	e_fx =fx6 (b_fx * x_fx);
MUL, 4, 0, 1;	f_fx =fx7 (d_fx * x_fx);
MUL, 5, 0, 2;	g_fx =fx8 (e_fx + f_fx);
MUL, 6, 0, 4;	y_fx =fx9 (g_fx + c_fx);
ADD, 7, 5, 6;	y = a*x*x + b*x + c;
ADD, 8, 3, 7;	{ a in [1, 1] /\ b in [1, 1] /\
ST, 9, 8, -1;	c in [1, 1] /\ x in [0, 1]
	-> y_fx in ? /\ (y_fx - y) in ? }

In Table III, we show the auto-generated GPU dataflow code and Gappa code for an example circuit $y = a * x^2 + b * x + c$. Here, we assume a, b, c are constants all with numerical value 1, x is an input variable with range [0, 1]. Each entry in the GPU dataflow code contains 4 fields for – instruction code, destination register, first source register, and second source register. The opcodes cover common arithmetic operations that we wish to analyze such as addition, multiplication, division, square-root, exponential, logarithm and conditionals. In addition, we support load and store instructions that serve as a convenient place to pass in the various input intervals to the GPU threads. For CPU-based Gappa analysis, we generate a script that follows Gappa's custom syntax. Gappa analysis requires custom type specifications for each variable precision as indicated by the `fx` type-casts for each operation. We also generate VHDL (not shown) from the same input C to synthesize pipelined hardware. The full FPGA CAD flow is only invoked once after the optimized bitwidth combination is determined.

A. Compatibility with ASICs and need for FPGA-specific Flow

Like many tools developed for FPGAs, such as LegUp [7], our GPU-accelerated tool can also be made to work with ASICs while remaining a valuable and useful tool within the FPGA community. FPGA-based designs are likely to benefit to a greater extent from a complete per-variable customization of computation unlike ASIC-based designs that must factor in safety margins to handle changes to accuracy requirements after fabrication. Rapid prototyping in a high-level synthesis environment may be of stronger appeal to a time-conscious FPGA developer than an ASIC developer who may also be slightly more tolerant of longer development cycles. An unconstrained bitwidth optimization problem already takes minutes to hours for modest-sized problems which is comparable to FPGA CAD times. Our approach reduces this to seconds or minutes to help make this particularly attractive as an optimization for an FPGA developer hunting for resource wins. To adapt our flow for ASICs, we would need to engineer an appropriate ASIC backend toolchain and construct resource and cost models for the particular technology.

TABLE IV: Benchmark Problem Characteristics

Benchmark	Variables	Arithmetic Operations	Inputs	Search Space
Level1$_{linear}$[22]	10	8	3	1K
Poly[13]	12	8	1	2K
Diode[22]	8	4	2	2K
Bellido[17]	13	9	3	24K
Approx1[22]	12	9	3	35K
Poly6[13]	19	12	1	52K
Level1$_{satur}$[22]	14	10	3	2M
Caprasse[17]	16	10	4	8M
Poly8[13]	22	21	1	160M
Approx2[22]	19	17	4	15G

V. Experiment Setup

A. Benchmarks

We evaluate our framework using a variety of typical benchmarks that have been used frequently in previous range and precision analysis work [13], [22], [21]. These are taken from the Alias-COPRIN benchmark set, the Minibit circuits, Mix FX-SCORE examples and others. Our benchmark set contains a mixture of problems with multiple inputs and outputs, complex non-linear operations (e.g. exponential and logarithm) and varying search space sizes. We tabulate their characteristics in Table IV. While the variable count may seem small, these generate enormously large search spaces (≈15G data points) that are tricky to cover effectively. The key to enabling precision analysis as a routine optimization on larger problems (multiple basic blocks) in high-level synthesis flow is to make this exploration tractable – using GPUs as shown in this paper.

B. Tools and Hardware

We compare the sequential annealing (ASA) implementation on an Intel Core i5-4570 CPU @ 3.2 GHz against and our approach mapped to an NVIDIA K20 GPU. We use CUDA 6.5 along with Thrust library (version 1.7.0) for simplified transfer of data between CPU and GPU and also for efficient GPU reduction routines. For precision analysis on the CPU, we use two tools that we have mentioned previously (1) simulated annealing package ASA v30.15, and (2) numerical analysis package Gappa v1.1.1. ASA uses the same CPU versions of our GPU thread code for range analysis, error propagation and resource estimation. We modify Gappa to add support for `exp` and `log` operations. For FPGA compilations,

we use Vivado Design Suite v2013.4 targeting the Kintex-7 XC7KLX160 FPGA to generate regression models for the various fundamental arithmetic operators. We use off-the-shelf `gcc-4.8` compiler that ships with Ubuntu 14.04 but expect our toolflow to be compatible with even older versions of `gcc` (2007 onwards) but we have not tested this.

VI. Evaluation

In this section, we evaluate the experimental results of our GPU-assisted range analysis and bitwidth allocation. We show the results of FPGA resource utilization as a function of desired accuracy as well as the sensitivity of the FPGA mapping to fidelity of the resource models. For speedup calculations we compared optimized CPU and GPU implementations and include the pruning time on the CPU when calculating total GPU time for speedup calculations. Furthermore, our GPU timing calculations also include CPU-GPU memory transfer times.

A. Resource Utilization

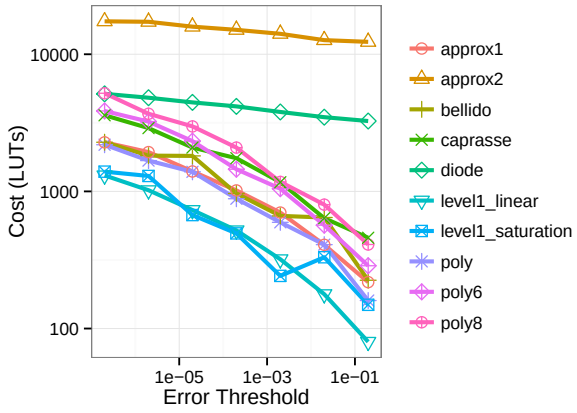


Fig. 3: Impact of Error Threshold on FPGA Resource Utilization (LUTs). As we relax the error threshold, we observe that smaller designs are possible

When the resulting computation accuracy is flexible, we can achieve significant reductions in FPGA resource utilization. In our compilation framework, we are able to provide a desired level of accuracy when driving the search for datapath variable precision. We show the sensitivity of the solution quality as a function of the reference error threshold for valid solutions in Figure 3. As is evident, we can generate 2–10× reduction in LUT count when considering various desired error thresholds. We observe that most benchmarks show reduced implementation cost at varying error thresholds indicating significant resource savings are possible for real-world accuracy requirements. Certain benchmarks with `exp` and `log` operators do not show as impressive reductions in cost as seen from the `approx2` and the `diode` benchmark. In these instances, the larger resource utilization of the `exp` and `log` operator dominates the system. Thus, for applications where approximate computations are allowed, our compiler can deliver a suitably optimized solution that aims to deliver exactly the accuracy that is desired and no more. Overall, our bitwidth optimization saves FPGA resource costs by as much as 4× (mean 2.5×) when compared to a baseline double-precision implementation. These qualitative results are better than the ones reported in [13], [20], [22], as we use a superior pruning heuristic and cover every single combination within the pruned search space.

B. Resource Model Accuracy

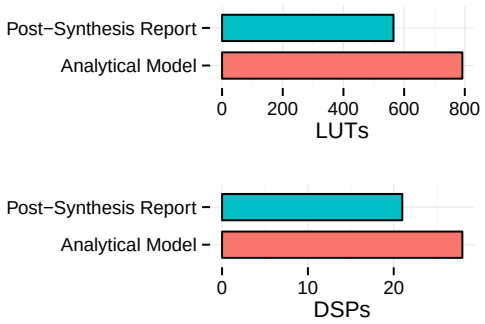


Fig. 4: Comparing Quality of Results

As mentioned earlier in Section III, due to the multi-stage nature of the FPGA CAD compilation flow, we need to build our resource model with particular care. We investigate the efficacy of using different area models on the final observed cost of the mixed precision results for the `level1-saturation` model. We compare: (1) approximate analytical area model, and (2) accurate post logic synthesis report. In each case, we run the full GPU-accelerated optimization to evaluate bitwidths and use the calculated resource numbers to filter out and select the best bitwidth. However, we generate final resource utilization after a complete place-and-route at the optimized bitwidths to compare the predicted model with actual real results. In Figure 4, we observe the reductions in LUTs and DSPs as we improve the fidelity of our resource model for `level1-saturation` by using downstream CAD tool results. If we perform bitwidth optimization solely relying on approximate analytical models, we need to spend 800 LUTs (post place-and-route). We can reduce this to 550 LUTs along with a DSP count reduction by 8 when we re-run the bitwidth optimization using the post-synthesis models. Logic synthesis tools are better at exploiting freedom of choice between LUTs vs. DSPs. Our post-synthesis models are better capable of exploiting this knowledge when driving the optimization.

C. Range Analysis

We first perform GPU-accelerated sub-interval analysis to improve the quality of the variable bounds. We obtain tighter interval bounds between 1–4× that of vanilla interval arithmetic across our benchmarks. In Figure 5, we show the performance impact of GPU acceleration on subinterval analysis as we vary the number of sub-intervals. Here, we compare optimized GPU runtime with sequential GPU Gappa runtimes when using Gappa's *dichotomy search* feature when keeping

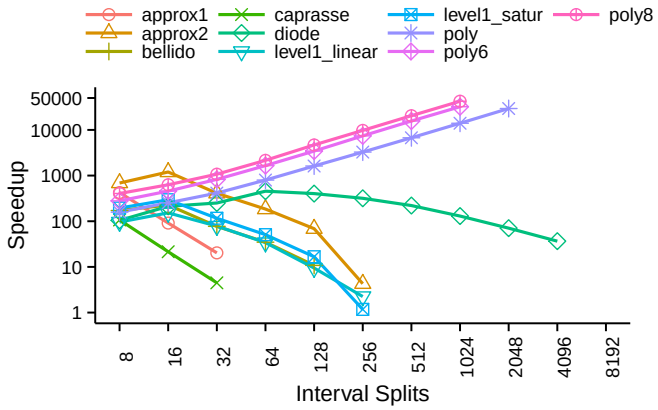


Fig. 5: GPU Speedup for Sub-Interval Analysis (16-thread CPU runtimes can be minutes to hours)

the number of sub-intervals identical in both cases. While we observe peak speedups of 3000×, we achieved best pruning results below 128 sub-intervals with associated speedups of 10–1000×. Fewer sub-intervals generate limited parallelism for GPU acceleration while larger sub-intervals exceed the GPU storage limits and become infeasible. The maximum number of permissible sub-intervals for GPU-compatible acceleration varies with the number of inputs to the benchmark (listed in the **Inputs** column of Table IV). Benchmarks with few inputs 1–2 such as `poly8`, `poly6`, and `poly` deliver continued speedups upto 8192 sub-intervals while the rest saturate between 8–64 sub-intervals.

D. Bitwidth Allocation

We now show results for the bitwidth exploration phase of the problem. The CPU-based method typically ran for dozens of minutes on average and was reduced to dozens of seconds when using GPU acceleration. Certain instances of ASA did not terminate even after running for hours, and we inserted an early exit condition to restrict runtimes. These runtimes are for the small dataflow kernels listed in Table IV which roughly correspond to the size of basic blocks in modern compilers. Larger programs in HLS must handle hundreds of such basic blocks resulting in a large overall runtime for the complete program when using CPUs alone. In Figure 6, we show the overall speedups for GPU accelerated brute-force exploration compared to sequential CPU implementation of ASA. We make the following key observations on analyzing the nature of speedups:

- For smallest benchmarks with very small search space sizes such as the `poly`, `diode`, and `level1_linear`, we observe that the single-shot GPU evaluation delivers a performance improvement greater than 10× without needing pruning assistance from ASA running on the CPU. This is to be expected and is unsurprising.
- For medium-sized benchmarks such as the `poly6`, `approx1`, `level1_satur`, and `bellido`, the GPU-only approach delivers larger but still modest speedups. We record substantial speedup improvements when using the first 10–25 ASA iterations as pruning assistance. Switching to the GPU after ≈25 iterations generally delivers the best balance between spending sequential pruning time on the CPU and fast evaluation of smaller search space on GPU.
- For the largest benchmarks such as the `approx2`, `poly8` and `caprasse`, if we purely rely on the pruning heuristic in Algorithm 1, the search space is too large to fit in the GPU memory capacity. In this setting, the GPU-only approach is simply infeasible (shown as 0 speedup in Figure 6). For these cases, we use the ASA-assisted pruning to move the search space into a feasible region. We observe significant speedups due the sheer size of the search space to be explored on the CPU.

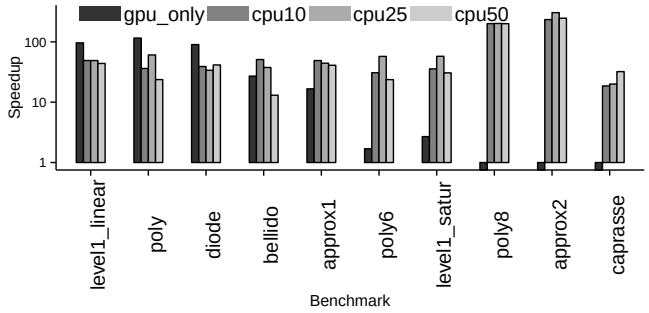


Fig. 6: Bitwidth Allocation Speedups (cpuX means X iterations of ASA on CPU run before GPU invoked), CPU runtimes again run into dozens of minutes and sometimes ASA must be terminated early as it runs for hours without termination.

E. Quality-Time Tradeoffs

Finally, we show the quality-time tradeoff trends in Figure 7. Here, we report the best-observed resource cost (cost = 1/quality) of the sequential ASA search after each trial as a function of CPU runtime across various benchmarks. It is useful to see the convergence towards the final results often shows significant gains in the first 10% of runtime. For the GPU-accelerated evaluation, our one-shot approach gives us the best answer as a single co-ordinate in this space at the same solution quality.

VII. Discussion and Context

A. Review of Previous Work

Research on range and precision analysis has evolved from simulation-based methods [11] to analytical approaches [13], [21], due to their speed in error modeling and range calculation. Interval Arithmetic (IA) [18] is the most well-known analytical method to calculate true bounds for general numerical algorithms. Some previous studies develop improved arithmetic techniques for producing more accurate error estimations. [15] presents quantized affine arithmetic (AA) which only considers inputs' uncertainty into affine expansions, thereby reducing the complexity of AA representation. [10] uses SAT-Modulo Theory (SMT) to break input ranges into sub-ranges for allocating optimal bit-width targeting a given precision specification, but is limited to range analysis only. [1] invents a

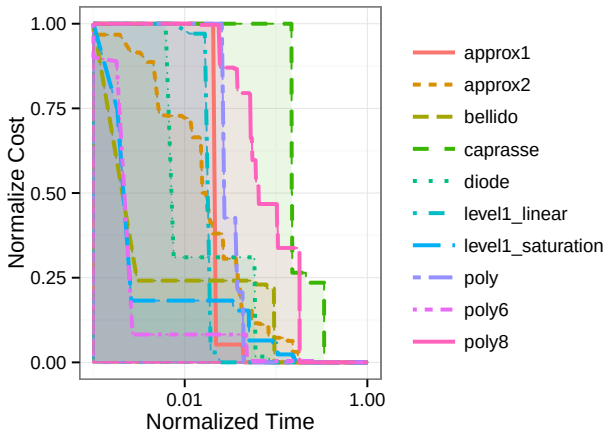


Fig. 7: Quality-Time Tradeoffs for ASA (CPU-only)

polynomial algebra-based analytical approach to find provable and tight error bounds. This line of work mainly handles the accuracy issue of bit-width error modeling, but does not help with the optimization speed directly. Another set of work focuses on how to speed up the process of bit-width optimization using custom heuristics. [13], [22] utilize simulated annealing for picking global optimized bits that minimize area cost. [20] performs partitioning followed by simulated annealing to improve runtime of [13]. [16] uses greedy algorithm for a coarse-gained search to find initial solutions, in tandem with Tabu search for refining the results' quality. [21] proposes two semi-analytical heuristics: progressive selection algorithm and tree-based search algorithm, dealing with fastness and optimality of precision analysis respectively. A GPU-based approach in the context of fixed-point analysis of VLSI circuits is presented in [4]. However, the speedups are compared against single-core CPUs on a single toy FIR-filter benchmark and the speedup is lower than the ones reported in this paper. They are only $\approx 58\times$ which is at the low end of our speedups and our speedups are over a faster baseline 16-threaded implementation.

B. Case for GPU-based Brute-Force Acceleration

More broadly, the use of GPU-based acceleration provides a model for limited brute-force exploration in FPGA CAD. At multiple heuristics stages of the CAD flow (*i.e.* placement, routing, synthesis), we are often forced to make sub-optimal decisions due to our inability to fully explore potential solutions. A GPU-accelerated approach can open the door to using single-shot brute-force techniques at leaf stages of the search tree inherent in the CAD algorithms. In the context of ASIC cell layout, the idea of fast, optimal end-case placement exploration has been explored earlier in [5]. This approach still uses branch-and-bound at the leaf but optimally solves the small end-case solution when the problem size becomes small enough. In our approach, we explore all-possible combinations in the end-case using parallel GPU-based threads. In the long run, exploring the entire search space will stay intractable even with advances in GPU technology (as FPGA problem sizes will keep growing), we expect this technique to be used at stages of the search trees when the search spaces are sufficiently small for brute-force exploration and optimal solutions to the sub-problem are desirable.

VIII. Conclusions

In this paper, we show how to accelerate precision analysis for bitwidth optimization by 10–1000$\times$ for range analysis and 50–200$\times$ for bitwidth allocation and when comparing an NVIDIA K20 GPU to an Intel i5-4570 CPU. We demonstrate the design and engineering a high-level synthesis approach suitable for rapid prototyping of bitwidth optimization algorithms. Using our HLS framework with tuneable accuracy targets, we are able to reduce FPGA resource utilization by as much as 10$\times$ if suitable approximate results are acceptable to the computation. To exploit GPU potential, we parallelize sub-interval analysis to improve interval analysis bounds and carefully prune the search space to enable single-shot exhaustive exploration of the search space. For range analysis, speedups correlated with the number of inputs and bitwidth allocation speedups tracked the size of the search space. As part of future work, we intend to extend this work to support Monte-Carlo sampling-based methods and integrate Affine Arithmetic models.

References

[1] D. Boland and G. A. Constantinides. Automated precision analysis: A polynomial algebraic approach. In *FCCM*, pages 157–164, 2010.

[2] D. Boland and G. A. Constantinides. A scalable approach for automated precision analysis. In *FPGA*, pages 185–194, 2012.

[3] S. Boldo, J.-C. Filliâtre, and G. Melquiond. Combining coq and gappa for certifying floating-point programs. *Intelligent Computer Mathematics*, pages 59–74, 2009.

[4] G. Caffarena and D. Menard. Many-core parallelization of fixed-point optimization of vlsi circuits through gpu devices. In *Design and Architectures for Signal and Image Processing (DASIP), 2012 Conference on*, pages 1–8, Oct 2012.

[5] A. E. Caldwell, A. B. Kahng, and I. L. Markov. Optimal partitioners and end-case placers for standard-cell layout. *Computer-Aided Design of Integrated Circuits and Systems, IEEE Transactions on*, 19(11):1304–1313, 2000.

[6] S. Callanan, D. J. Dean, and E. Zadok. Extending gcc with modular gimple optimizations. In *Proceedings of the 2007 GCC Developers Summit*, pages 31–37, 2007.

[7] A. Canis, J. Choi, M. Aldham, V. Zhang, A. Kammoona, J. H. Anderson, S. Brown, and T. Czajkowski. Legup: high-level synthesis for fpga-based processor/accelerator systems. In *Proceedings of the 19th ACM/SIGDA international symposium on Field programmable gate arrays*, pages 33–36. ACM, 2011.

[8] G. Constantinides and G. Woeginger. The complexity of multiple wordlength assignment. *Applied Mathematics Letters*, 15(2):137 – 140, 2002.

[9] L. Ingber. Very fast simulated re-annealing. *Mathematical and computer modelling*, 12(8):967–973, 1989.

[10] A. Kinsman and N. Nicolici. Finite precision bit-width allocation using sat-modulo theory. In *Design, Automation Test in Europe Conference Exhibition, 2009. DATE '09.*, pages 1106–1111, April 2009.

[11] K.-I. Kum and W. Sung. Combined word-length optimization and high-level synthesis of digital signal processing systems. *Computer-Aided Design of Integrated Circuits and Systems, IEEE Transactions on*, 20(8):921–930, Aug 2001.

[12] C. Lattner and V. Adve. Llvm: A compilation framework for lifelong program analysis & transformation. In *Code Generation and Optimization, 2004. CGO 2004. International Symposium on*, pages 75–86. IEEE, 2004.

[13] D.-U. Lee, A. A. Gaffar, R. C. C. Cheung, O. Mencer, W. Luk, and G. A. Constantinides. Accuracy-guaranteed bit-width optimization. *IEEE Trans. on CAD of Integrated Circuits and Systems*, 25(10):1990–2000, 2006.

[14] M. D. Linderman, M. Ho, D. L. Dill, T. H. Meng, and G. P. Nolan. Towards program optimization through automated analysis of numerical precision. In *Proc. IEEE/ACM Int. Symp. on Code Generation and Optimization*, CGO '10, pages 230–237, New York, NY, USA, 2010. ACM.

[15] J. Lopez, C. Carreras, and O. Nieto-Taladriz. Improved interval-based characterization of fixed-point lti systems with feedback loops. *Computer-Aided Design of Integrated Circuits and Systems, IEEE Transactions on*, 26(11):1923–1933, Nov 2007.

[16] D. Menard, N. Herve, O. Sentieys, and H.-N. Nguyen. High-level synthesis under fixed-point accuracy constraint. *JECE*, 2012, 2012.

[17] J.-P. Merlet. The COPRIN benchmarks. http://www-sop.inria.fr/coprin/logiciels/ALIAS/Benches/benches.html.

[18] R. E. Moore, R. B. Kearfott, and M. J. Cloud. *Introduction to Interval Analysis*. SIAM, 2009.

[19] A. Nayak, M. Haldar, A. Choudhary, and P. Banerjee. Precision and error analysis of MATLAB applications during automated hardware synthesis for FPGAs. In *Proc. of Conf. on Design, Automation and Test in Europe*, DATE '01, pages 722–728, 2001.

[20] W. G. Osborne, R. C. C. Cheung, J. Coutinho, W. Luk, and O. Mencer. Automatic accuracy-guaranteed bit-width optimization for fixed and floating-point systems. In *Int. Conf. on Field Programmable Logic and Applications (FPL)*, 2007.

[21] S. Vakili, J. Langlois, and G. Bois. Enhanced precision analysis for accuracy-aware bit-width optimization using affine arithmetic. *Computer-Aided Design of Integrated Circuits and Systems, IEEE Transactions on*, 32(12):1853–1865, 2013.

[22] D. Ye and N. Kapre. MixFX-SCORE: Heterogeneous fixed-point compilation of dataflow computations. In *Proceedings of the 2014 IEEE 22nd International Symposium on Field-Programmable Custom Computing Machines*, FCCM '14, 2014.