Design and Verification of an open-source SFU model for GPGPUs

Josie E. Rodriguez Condia*, Juan-David Guerrero-Balaguera†, Cristhian-Fernando Moreno-Manrique†,
Matteo Sonza Reorda*
*Politecnico di Torino - Department of Control and Computer Engineering (DAUIN)
{josie.rodriguez, matteo.sonzareorda}@polito.it
†Universidad Pedagogica y Tecnologica de Colombia (UPTC) - Electronic Engineering School
{juandavid.guerrero, Cristhian.Moreno}@uptc.edu.co

***Abstract*—General Purpose Graphic Processing Units (GPGPUs) are widely used in data-intensive applications, such as multimedia and high-performance computing. These technologies are currently used also to support safety-critical applications (e.g., in the automotive and industrial domains) to implement computer vision, sensor fusion, or machine learning algorithms, which often require the processing of complex transcendent or trigonometric functions. In these cases, an integrated special function unit in the GPGPU is utilized, which is intended to increase the performance in parallel operations. However, this complex module is not present in most of the available architectural and micro-architectural open-source models of GPGPUs, so limiting the characterization and analysis of applications using these units. In this work, we report about the design and functional verification of a Special Function Unit to execute transcendent and trigonometric operations in GPGPUs. We integrated the proposed module within an open-source GPGPU (FlexGripPlus) implementing the G80 micro-architecture. The experimental results show the significant improvements in performance and accuracy achievable by using these modules in parallel applications running in a GPGPU.**

***Index Terms*—Functional Verification, General Purpose Graphics Processing Units (GPGPUs), Special Function unit (SFU)**

I. Introduction

Nowadays, General Purpose Graphic Processing Units (GPGPUs) are the leading workhorse solution concerning data-intensive applications, such as multimedia, multi-signal analysis, and high-performance computing. Thanks to their highly parallel architecture, these technologies are now also used in embedded and safety-critical applications, where high levels of reliability are required. In automotive and robotics (to mention 2 major application areas), GPGPUs are adopted in sensor-fusion, computer vision [1] and *Advanced Driver-Assistance Systems* (ADAS).

Originally, Graphics Processing Units (GPUs) were devoted to managing large arrays of data and were commonly related as special-purpose accelerators to implement image processing techniques. These techniques include the processing of images in raster and vector formats, the processing of surfaces, the mapping of textures, and the rendering of images. These operations usually require the execution of transcendental and trigonometric functions, so specialized hardware accelerators were included in GPUs. The GPU technologies quickly evolved and started to be used with success in the general-purpose domain as GPGPUs.

The transcendental and trigonometric hardware accelerators (also known as *Special Function Units* or *SFUs*) are often present in GPGPU modern architectures, since complex processing algorithms are typical in modern applications [2]. Thus, most manufacturers usually include SFUs to process part of the complex operations. In principle, two main approaches are used to design and implement an SFU: *i)* iterative and *ii)* non-iterative. The first approach is devoted to executing iterative algorithms that converge linearly or quadratically to the result, such as the *COordinate Rotation DIgital Computer* (CORDIC), Newton-Raphson and Goldschmidt algorithms, which are more oriented to the highest accuracy and precision [3]. On the other hand, non-iterative architectures are based on table-based solutions and polynomial and rational approximations, such as those based on quadratic interpolation using enhanced minimax approximations, targeting the optimization of hardware and power consumption [4]. However, the intrinsic complexity of the SFUs and the missing details about commercial implementations in GPGPUs reduce the possibility of exploring alternative design approaches, analyzing the provided operational features, and observing their impact on the performance of the GPGPU in general and of specific applications, as well. Moreover, in other fields of study, such as reliability and test, commonly analyze reliability features, fault effects, and mitigation strategies applied to computer architectures. However, the analysis of SFUs is still considered an open case study by the lack of micro-architectural models to observe the fault effects and evaluate the effectiveness of mitigation and fault-tolerant techniques [5].

In this work, we report about the design, implementation, and functional verification of an open-source SFU module intended to support the design exploration of SFUs used as accelerators in processor-based systems and GPGPU devices. Moreover, the proposed SFU can also be used to support the reliability analysis and develop the quantitative assessment of mitigation and hardening solutions for applications running on GPGPUs.

The proposed SFU module follows most of the func-

This work has been partially supported by the European Commission through the Horizon 2020 RESCUE-ETN project under grant 722325.

978-1-7281-9444-8/20/$31.00 ©2020 IEEE

tional descriptions and guidelines of the SFU integrated into GPGPUs using the G80 micro-architecture of NVIDIA. The implementation was performed in VHDL language following a modular approach and incorporated in the FlexGripPlus open-source GPGPU model [6]. A set of assembly (SASS) instructions was implemented to control the SFU in the GPGPU. These instructions are compatible with the CUDA programming environment under the SM 1.0 compatibility. To the best of our knowledge, this is the first available open-source SFU module for GPGPUs.

Validation and verification were performed at the unit and system levels of abstraction. Firstly, a stand-alone unit verification was conducted on the SFU. Then, we integrated the SFU into the central core of FlexGripPlus, and the expected functionality was verified on the resulting system.

The paper is organized as follows. Section II provides some background about the organization of a GPGPU. This section also overviews the main features of the micro-architecture of the FlexGripPlus model where we integrated the newly developed SFU. Section III describes the architecture of the proposed SFU. Section IV describes the approaches used to verify and validate the SFU independently and integrated into the GPGPU. Section V reports the experimental results and their analysis, and Section VI draws some conclusions and lists some related future works .

II. Background

A. Organization of a GPGPU

GPGPUs are composed of a combination of multiple parallel architectures. In principle, these architecture are all based on the Single-Instruction Multiple-Data (SIMD) paradigm using several cores, also known as *Streaming Multiprocessors* (SMs), to compute instructions with high throughput. Inside each SM, one instruction is fetched from memory, and then decoded and processed in parallel using a set of available execution resources (Floating-point units (FPUs), Streaming Processors (SPs), and dedicated accelerators). These execution units perform the operations from each independent task (*thread*) with a static allocation. SMs are organized in groups of threads (called *Warps*) by a scheduler controller that manages and monitors each group of tasks. Internally, SMs also include several pipeline stages, local memories, and register files to process each thread in parallel.

B. The FlexGripPlus Architecture

The FlexGripPlus model [6] is an open-source soft-core GPGPU fully described in VHDL that implements the G80 architecture from NVIDIA [7]. This model is an improved version of the original FlexGrip model developed by the University of Massachusetts [8]. FlexGripPlus supports up to 28 assembly instructions, and it is fully compatible with the CUDA programming environment under the SM 1.0 compatibility.

FlexGripPlus is mainly composed of an array of SMs. Each SM executes instructions following variations of the SIMD and Single-Instruction Multiple-Thread (SIMT) taxonomies. A Block Scheduler assigns the tasks to every SM. Internally, an SM includes a Warp Scheduler Controller that submits the warps and dispatches them to the available SPs and FPUs. The flexibility of the model allows the selection of 8, 16, or 32 SPs. Each SM is organized in a five-stage pipeline (see Fig. 1). The SPs, FPUs, and a branch management unit are located in the *Execute* stage and operate in parallel. The data channels (feeding the operands) for the SPs, FPUs, and the branch module are shared and switched depending on the executed instruction.

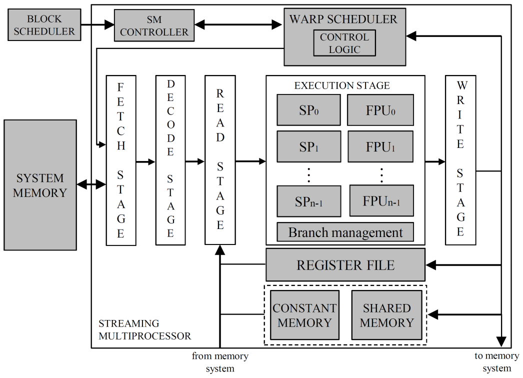

Fig. 1. A general scheme of the SM in FlexGripPlus.

FlexGripPlus does not include an SFU accelerator and has no the support for SFU operations. However, this model has the main features of common GPU architectures and is compatible with one commercial programming tool, so this model is an excellent candidate to integrate the new SFU and support the instructions from the programming environment.

III. SFU design

The design of the SFU takes into account the descriptions and specifications for the G80 architecture introduced in [7] to compute transcendental functions. The proposed SFU uses a modular scheme and is compatible with the IEEE 754 standard for single-precision operations. We used the iterative and the non-iterative to describe the functions in the SFU. For that purpose, we identified five special functions to be described and implemented in the SFU: $\sin x$, $\cos x$, $\log_2 x$, 2^x, and $\frac{1}{\sqrt{x}}$. A detailed review of the CUDA programming environment and the PTX manual under the SM 1.0 computer compatibility revealed that the execution of the $\sqrt{x}$ operation is replaced by two instructions, one reciprocal operation *RCP* and another instruction for the $\frac{1}{\sqrt{x}}$ function. It is worth noting that the reciprocal operation was previously introduced in the FPU of the model and is not considered in the present design. However, this operation is commonly included in multiple designs of SFUs.

The SFU is designed using a Bottom-Up modular approach, so each function is described individually, and then these are integrated as a multi-functional block. One golden SFU model was designed at the architectural level following the functional and operational specifications. The *Octave* framework is used to describe the golden SFU model. Then, each module is

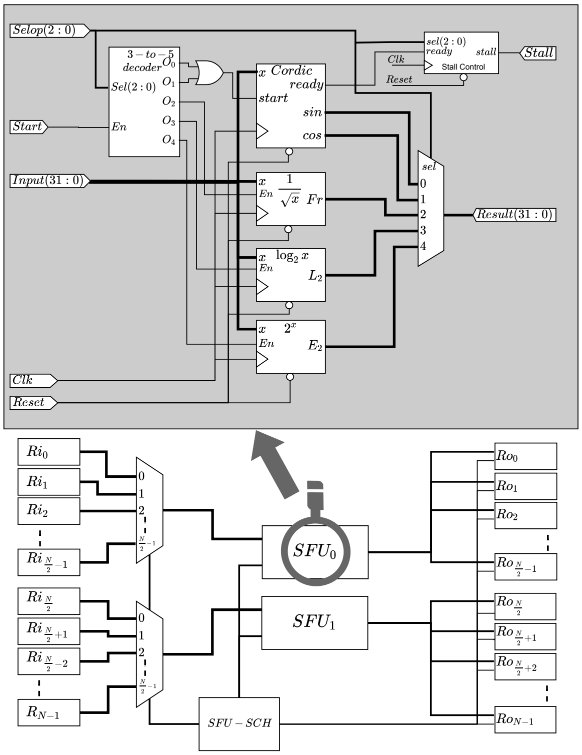

Fig. 2. A general scheme of the proposed SFU module (Top) and the integration with the GPGPU (Bottom)

detailed and represented at the RT level. Fig. 2 shows the proposed SFU design and its integration in the GPGPU architecture. As it can be observed, the SFU also includes some control signals to enable and select the function to be executed. The SFU is driven by a single clock. The input and output channels are limited to single-precision operands.

A. SIN and COS modules

The CORDIC algorithm is used to implement the *SIN* and *COS* functions. The proposed structure uses the circular coordinate system in the rotation mode with 16 iterations to determine the result [9]. This sub-module is composed of two single-precision floating-point multiplier and three adder/subtractor, compliant with the IEEE 754 standard. A routing logic is used to perform the operations of the intermediate results. The CORDIC function is implemented sequentially to reduce the number of floating-point modules in the module. Two lookup tables (LUTs) are included in the design to store the constant values of 2^{-n} y θ_n obtained following the equations in [9]. Each LUT stores a total of 16 constants.

B. $log_2 x$ module

The $log_2 x$ operation uses the Adaptable Logarithm Approximation (ALA) method [10]. ALA was conceived to process operands in integer and fixed-point formats and is inspired in a piece-wise linear approach proposed by Mitchell [11]. ALA uses an arbitrary number of line segments to combine computer efficiency and precision in approximating this function.

In this work, we adapted the ALA algorithm from integer to floating-point values, so in the proposed structure, ALA accepts operands compliant with the IEEE-754 standard.

The integer part of the result is directly forwarded from the exponent part (removing the bias) of the input value. On the other hand, the mantissa of the input value is directly considered as the first approximation. Then, a new approximation is derived in pieces of s segments of equivalent length (representing the distance between the first linear approximation and the ends of each segment). The results of the approximations are stored in the two LUTs of $s/2$ coefficients each one. The use of s segments (s a power of 2) allows the use of the most significant log2(s) bits, from the mantissa, to obtain two coefficients relating to the segment for the next approximation. In this implementation, we selected 64 segments. The most significant bits of the mantissa are used to address the LUTs and retrieve the representative coefficient.

The fractional part of the results is calculated through a multiplication between the difference of the two coefficients, and the mantissa without log2(s) most significant bits. Then, the result is added to the mantissa and the first coefficient of the segment. Finally, normalization is used to represent the result as a floating-point number in single precision.

C. 2^x module

The 2^x function employs the same ALA structure used for the $log_2 x$ operation, with minor changes in the hardware structure. Moreover, new segment coefficients are determined to be used during the operation of the 2^x function.

The implementation followed the same procedures explained before to calculate the fractional part of the result. However, final normalization is not required. On the other hand, the exponent of the results is calculated converting the input value into fixed-point format and taking the seven most significant of the input value.

D. Rsqrt module

The $\frac{1}{\sqrt{x}}$ function was implemented using the Fast Inverse Square Root algorithm (FISR), which was originally part of the source code of the QUAKE II video game [12], [13]. FISR algorithm is based on a simultaneous calculation of $log2$ and 2^x, which then are combined to calculate an approximation of $\frac{1}{\sqrt{x}}$. The algorithm neglects the floating-point format representation and considers the input bit patterns as an integer number to apply a linear approximation to the logarithmic function [14]. Thus, The entire floating-point range represents a piece-wise linear approximation of a monotonic logarithm function. The value is then subtracted (in integer) with the equivalent value of one in floating-point format (0x3F800000). A scale-down is then applied by dividing (as floating-point numbers) the resulting value by 0x00800000, so obtaining the approximation of the $log2$. The calculation of 2^x follows an inverse approach. The input value is first scaled with 0x00800000 as a floating-point number and then subtracted as an integer by 0x3F800000. Both results are then combined and therefore the reciprocal of square root can

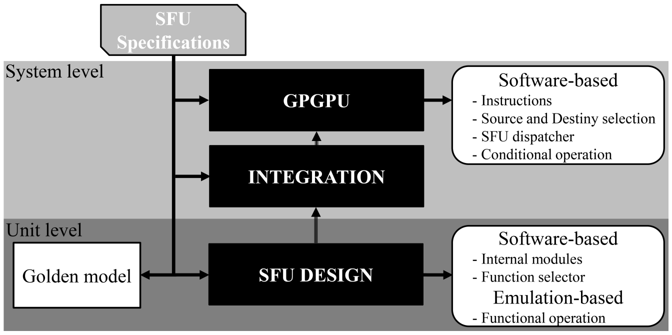

Fig. 3. Verification scheme for the SFU.

be expressed as $\frac{1}{\sqrt{x}} = 2^{-0.5*\log_2 x}$. Both operations ($log2$ and 2^x) are canceled, and a simplified version of the algorithm is obtained by processing as integers the subtraction of a constant (0x5F400000) by the input value shifted to the right on one position. According to [12], error minimization can be applied obtaining a more accurate constant, so replacing 0x5F400000 by 0x5F375A86. After applying the algorithm, the result is refined by using once the Newton-Raphson approximation. The parallel implementation of the module comprises four multipliers and one adder/subtractor modules in single-precision floating-point, one shifter, and one integer subtractor.

E. Integration of the SFU with the GPGPU

We followed most of the descriptions of the SFUs used in real GPGPUs. For this purpose, we used the intrinsic functions in CUDA combined with PTX instructions to force the compiler to generate the assembly instructions activating the SFU module. Then, we identified the opcodes of each instruction and implemented them in the FlexGripPlus model. The implemented instructions are fully compatible with the CUDA compiler. Further details about the implemented SFU instructions (*SIN*, *COS*, *LG2*, *EX2*, *RSQ*) can be found in [15]. The *Decode* and *Execute* pipeline stages were modified to implement the instructions and to integrate the SFUs.

In the original design of the G80 architecture, two SFUs are located in each SM, so in the *Execute* stage, two SFUs were added in parallel to the existing SPs, FPUs, and branch management modules. Moreover, it was included one thread dispatcher (SFU-SCH) to manage the operation of the parallel threads when executing instructions using the SFUs (see Fig. 2). Some additional input and output registers (*Rix* and *Rox*) and multiplexers are used to preserve and control the operands and results of each thread before submitting the results to the next stage and process another group of threads. Figure 2 shows the integration of the SFU within the GPGPU model.

A parametric description was used for the SFU-SCH module to proportionally change the number of SFUs when the number of SPs and FPUs in the system changes. It should be noted that the planar attribute interpolation is not considered part of the actual design and is left for future work. Moreover, in the integration of the SFU with the GPGPU core, the scheduler controller of the SM was not modified to allow the independent execution of instructions in the SFU by considering that the implemented controller does not include data-hazard algorithms to avoid the conflict in the submission of instructions.

IV. SFU VERIFICATION

We adopted a strategy based on a combination of simulation-based and emulation-based verification. The golden model (adopted as a reference during the design stage) was also used to perform the verification of the SFU. Figure 3 shows the scheme used for the functional verification flow. In this scheme, the verification is divided into two levels of abstraction: *i)* unit and *ii)* system verification.

At the unit level, simulation-based and emulation-based verification were applied. In contrast, simulation-based is used at the system level. The simulations are performed in the *ModelSim* and *Xcelium* frameworks. One FPGA and a soft-processor are used to synthesize the SFU and check its operation.

A. Unit verification

The SFU module is verified using the classical bottom-up approach, targeting the local verification of each submodule in the SFU design and then moves up to the complete module. In principle, the complexity of the SFU does not allow us to use exhaustive verification techniques using simulation-based methods in a reasonable time. The stand-alone unit would require a large set of combinations ($\approx 2.14\times 10^{10} = 2^{32}$ (input operands) $\cdot$ 5 (functions)).

The main targets of the unit verification are: *i)* the correct execution of the target function for several input operands, *ii)* the selector of the function and *iii)* the internal operation of each function. Under these conditions, we employed a compact set of patterns to verify the functionality of the SFU. A random set of 20,000 samples were used as inputs for each sub-module in the SFU. Some patterns were replaced by deterministic ones targeting edge conditions and exceptions. The previous patterns were applied at software and emulation levels. In the emulation case, two Linear Feedback Shift Register (LSFRs) modules and some LUTs were used to apply suitable patterns to the SFU in the FPGA. A soft-processor controlled the application of stimuli and retrieved results during the emulation.

B. System verification

The main targets are the structures added to integrate the SFU into the GPGPU. The objectives of the verification are: *i)* the implementation of the instructions and the functional selector in the SFU, *ii)* the correct selection of the input and output operands for each thread in the SFU (source and destination locations), *iii)* the SFU dispatcher module, *iv)* the conditional operation of the SFU, and *v)* the execution of the SFU instructions, even interleaved with other instructions.

An exhaustive verification of the SFU integrated in the GPGPU is clearly not feasible and ($\approx 8.79\times 10^{13}$ = 2^{32} (input operands) $\cdot$ 5 (functions) $\cdot$ 2^6 (source locations)

TABLE I
HARDWARE FEATURES ON THE SFU IN ASIC AND FPGA IMPLEMENTATIONS

	Module	Frequency (MHz)	Hardware		Power (mW)	Performance (ns)
			Area (μm^2)	Cells/LUTs		
ASIC	SP	500	1,149.91	3,219	0.88	0.75
	SFU		10,190.97	26,339	2.17	1.98
FPGA	SP	100	-	1,907	171.74	44.83
	SFU		-	7,791	144.18	34.5

$\cdot$ 2^6 (destiny locations)) combinations would be required. A simulation-based verification was performed using two specially developed programs written in the assembly language of the GPGPU and using the SFU instructions to verify the correct functional operation of the integration of the SFU and the GPGPU. Both programs were developed using a modular approach, so a set of sub-routines was defined targeting specific functional features, and then, these routines were assembled for each program. The modular routines allow the parametric change of operands, source and destination registers, memory addresses, and operations order.

The SFU instructions use the register file to load and store the operands and results, respectively. Thus, the first program varies the register locations (used to load or store the operand or the result) of one operation in the SFU. This program checks the implementation of the source and destination selectors in the instructions. This process is performed in parallel for a group of 32 threads. In both cases (Source and Destination), the register location is swapped from 1 to 63. Permutation techniques were employed in the modular code to generate the instructions used to check the source, destination, and sequential operation of each function in the SFU.

Regarding the dispatcher of the SFU, the programs only employ 32 threads, so the dispatcher distributes the task on each SFU by dividing into two parts the eight active threads (high-part and low-part), so the high-part tasks are submitted to the SFU$_0$, and the low-part tasks are submitted to the SFU$_1$. For this purpose, the second program takes into account this behavior and distributes the input operands in memory in such a way that each operand is the same for SFU$_0$ and SFU$_1$ but different among the threads, (i.e., thread 0 and thread 4 shares equal input operands, and similarly for thread 1 and thread 5).

The conditional execution of the SFU is checked through adding a part of the code generating intra-warp divergence. For this purpose, conditional operations are included, and two paths are devoted to reviewing the activation of the instructions when conditions from the predicate flags are required. The first path executes a simple operation *ADD*. Meanwhile, the second path executes one of the instructions using the SFU. The process is repeated for each instruction using the SFU.

V. EXPERIMENTAL RESULTS

The experiments were performed on the SFU module and the GPGPU integrating the SFU. The hardware costs for the SFU were evaluated for both, an ASIC and an FPGA implementation. For this purpose, the SFU was synthesized first using the Open-Cell 15nm technology library in the *Design Compiler* framework and then in the *Quartus II V13.0* environment targeting the Cyclone IV EP4CE115F29C7 FPGA platform.

TABLE II
ERROR AND STANDARD DEVIATION FOR THE SFU

Function	Input interval	Average relative error	σ
$\sin X$	$[0, \pi/2]$	0.47$\times10^{-5}$	4.16$\times10^{-5}$
	$[0, \pi/6)$	1.29$\times10^{-5}$	8.35$\times10^{-5}$
	$[\pi/6, \pi/3)$	0.10$\times10^{-5}$	0.07$\times10^{-5}$
	$[\pi/3, \pi/2]$	0.03$\times10^{-5}$	0.02$\times10^{-5}$
$\cos X$	$[0, \pi/2]$	0.60$\times10^{-5}$	10.71$\times10^{-5}$
	$[0, \pi/6)$	0.03$\times10^{-5}$	0.03$\times10^{-5}$
	$[\pi/6, \pi/3)$	0.10$\times10^{-5}$	0.07$\times10^{-5}$
	$[\pi/3, \pi/2]$	1.69$\times10^{-5}$	18.50$\times10^{-5}$
$\log_2 X$	[1,2)	6.17$\times10^{-5}$	38.41$\times10^{-5}$
	[1,4/3)	17.20$\times10^{-5}$	65.14$\times10^{-5}$
	[4/3,5/3)	0.92$\times10^{-5}$	0.61$\times10^{-5}$
	[5/3,2)	0.40$\times10^{-5}$	0.25$\times10^{-5}$
2^X	[0,1)	0.38$\times10^{-5}$	0.22$\times10^{-5}$
	[0,1/3)	0.39$\times10^{-5}$	0.24$\times10^{-5}$
	[1/3,2/3)	0.38$\times10^{-5}$	0.23$\times10^{-5}$
	[2/3,1)	0.38$\times10^{-5}$	0.22$\times10^{-5}$
$\frac{1}{\sqrt{X}}$	[1,4)	98.01$\times10^{-5}$	59.29$\times10^{-5}$
	[1,8/5)	73.58$\times10^{-5}$	54.88$\times10^{-5}$
	[8/5,11/5)	92.19$\times10^{-5}$	43.73$\times10^{-5}$
	[11/5,14/5)	150.00$\times10^{-5}$	19.94$\times10^{-5}$
	[14/5,17/5)	59.00$\times10^{-5}$	46.55$\times10^{-5}$
	[17/5,4)	110.00$\times10^{-5}$	66.41$\times10^{-5}$

Table I reports the main features in terms of hardware, power, and performance for both cases. The cost of one SP in the GPGPU was included as a reference for the comparison. In terms of performance and power consumption, some parameters, such as the operative voltage, frequency, and the optimization during the compilation generate the observed behavior. Finally, a complete synthesis of the GPGPU model integrating the SFUs was performed. The results show that the hardware cost of one SFU module varies from four to eight times the cost of one SP core: four (in the FPGA) or eight (in the ASIC) SPs occupies almost the same area that one SFU. Moreover, the insertion of SFUs in the GPGPU core represents an additional cost of 5.3% of cells and 7.5% of area.

A. Precision

Table II reports the average relative error produced by each one of the modules in the SFU. The results were obtained by performing simulation-based verification. As it can be observed, the average relative error and the standard deviation (σ) are moderate for most of the modules when comparing with results presented in [4]. However, the implementations algorithms differ and the relative error in SIN, COS, LOG2, and RSQRT depend on the evaluated range of input values.

In the CORDIC algorithm, the relative error increases rapidly when the input values are very close to 0 in the $\sin x$ function with an experimental maximum relative error of 0.4%. A similar situation is present in the $\cos x$ function when the input angle approaches $\frac{\pi}{2}$ with a maximum relative error of 0.8%. These errors are mainly caused by rounding issues related to the selection and fill with the LUTS for the single-precision representation. Moreover, the iterative implementation contributes to accumulating the rounding error on each intermediate result and is propagated to the final result. One possible solution to decrease the error would be the use of double-precision formats and applying a higher number

TABLE III

Coverage metrics in the SFU for unit and system verification

	Module	SFU	Interconections with GPU
Coverage(%)	Block	100.0	96.61
	Expression	68.0	66.67
	Toggle	97.0	93.75
	FSM	100.0	100
	Code	98.51	85.68
	Overall	84.61	92.84

of iterations allowing a fine-tune of the result. However, the performance could be compromised.

The relative error in the $\log_2 x$ is maximum when the input values are close to 1, and therefore the logarithm function tends to 0, so obtaining a maximum error value of 0.65%. The result deviation is caused by the quantization process, which determines the number of segments and the number of bits used to represent the coefficients, so directly affecting the final error. This error could potentially be decreased by employing a higher number of approximation segments as well as a more significant amount of bits to represent the coefficients.

Regarding the maximum relative error in $\frac{1}{\sqrt{x}}$ function, the obtained 0.18% is produced when the function is evaluated at values close to $1, 1.5, 2.5$ and 4. This behavior can be explained, considering that a single iteration of the Newton-Raphson method was used. A solution to reduce the error can be obtained by applying more iterations. Another approach could be based on the ALA algorithm combining the $log2$ and 2^x functions.

Finally, each function in the SFU was described as a software program for the GPGPU using Taylor series or the equivalent polynomial approximation. Then, Both versions of the functions (software and SFU) were compared, using the maximum number of threads in the SM, showing that the SFU functions can increase the performance of an application in the GPGPU by twice, for the iterative functions, to 22 times, for the non-iterative functions.

B. Verification

Table III reports the results obtained when computing the coverage metrics employing the *IMC* tool. In the results, we can observe that the procedures applied to verify the SFU at the unit level produced the 84.61% of overall coverage. The missing coverage is mainly caused by constant descriptions in the modules that are not feasible to stimulate. Regarding the coverage at the system level, the two programs reported an accumulated relatively high percentage of coverage (92.84%) when verifying the implemented instructions and the interconnections modules (controller, registers, and multiplexers) between the SFUs and the GPGPU.

The description and test-benches of the designed SFU are freely available[1]. Moreover, documentation [16] for standalone operation can also be accessed.

VI. Conclusions and Future Work

This paper outlined the design of a Special Function Unit (SFU) to be used in a GPGPU, its functional verification, and the integration with the open-source GPGPU (FlexGripPlus). The experimental results show that one SFU seems to have an equivalent hardware cost of four to eight execution units in the GPGPU, which contributes to explain the limited number of SFUs in a GPGPU core. Moreover, the proposed SFU increases the performance of an application in the order of two to 22 times depending on the architectural approach employed to describe the functions in the SFU. Similarly, the verification approaches were effective on targeting different objectives when verifying the functionality of the proposed SFU.

The modular scheme of the SFU allows the optimization of the implemented functions: as a future work, we will target the reduction of the relative error. Moreover, we plan to apply other methods of verification to further increase the confidence in the design. Additionally, we plan to select workloads using the SFU in GPGPUs and perform reliability analyses at the micro-architectural level of the GPGPU.

References

[1] W. Shi, M. B. Alawieh, X. Li, and H. Yu, "Algorithm and hardware implementation for visual perception system in autonomous vehicle: A survey," *Integration*, vol. 59, pp. 148 – 156, 2017.

[2] A. Li, S. L. Song, M. Wijtvliet, A. Kumar, and H. Corporaal, "Sfu-driven transparent approximation acceleration on gpus," in *Proceedings of the 2016 International Conference on Supercomputing*, ser. ICS '16. New York, NY, USA: Association for Computing Machinery, 2016.

[3] P. K. Meher, J. Valls, T. Juang, K. Sridharan, and K. Maharatna, "50 years of cordic: Algorithms, architectures, and applications," *IEEE Transactions on Circuits and Systems I: Regular Papers*, vol. 56, no. 9, pp. 1893–1907, 2009.

[4] S. F. Oberman and M. Y. Siu, "A high-performance area-efficient multifunction interpolator," in *17th IEEE Symposium on Computer Arithmetic (ARITH'05)*, 2005, pp. 272–279.

[5] S. Di Carlo, G. Gambardella, M. Indaco, I. Martella, P. Prinetto, D. Rolfo, and P. Trotta, "A software-based self test of cuda fermi gpus," in *2013 18th IEEE European Test Symposium (ETS)*, 2013, pp. 1–6.

[6] J. E. R. Condia, B. Du, M. Sonza Reorda, and L. Sterpone, "Flexgripplus: An improved gpgpu model to support reliability analysis," *Microelectronics Reliability*, vol. 109, p. 113660, 2020.

[7] E. Lindholm, J. Nickolls, S. Oberman, and J. Montrym, "Nvidia tesla: A unified graphics and computing architecture," *IEEE Micro*, vol. 28, no. 2, pp. 39–55, March 2008.

[8] K. Andryc, M. Merchant, and R. Tessier, "Flexgrip: A soft gpgpu for fpgas," in *2013 International Conference on Field-Programmable Technology (FPT)*, 2013, pp. 230–237.

[9] J. S. Walther, "A unified algorithm for elementary functions," in *Proceedings of the May 18-20, 1971, Spring Joint Computer Conference*, ser. AFIPS '71 (Spring). New York, NY, USA: Association for Computing Machinery, 1971, p. 379–385.

[10] D. Bariamis, D. Maroulis, and D. K. Iakovidis, "Adaptable, fast, area-efficient architecture for logarithm approximation with arbitrary accuracy on fpga," *Journal of Signal Processing Systems*, vol. 58, no. 3, pp. 301–310, 2010.

[11] J. N. Mitchell, "Computer multiplication and division using binary logarithms," *IEEE Transactions on Electronic Computers*, vol. EC-11, no. 4, pp. 512 – 517, 1962.

[12] M. Robertson, *A brief history of invsqrt*. Department of Computer Science & Applied Statistics, 2012.

[13] T. Bradshaw, "id-software/quake-iii-arena," Jan 2012. [Online]. Available: https://github.com/id-Software/Quake-III-Arena

[14] J. F. Blinn, "Floating-point tricks," *IEEE Comput. Graph. Appl.*, vol. 17, no. 4, p. 80–84, Jul. 1997.

[15] J. E. R. Condia, B. Du, G. Roascio, E. Scie, and J.-D. Guerrero-Balaguera, "Programmers manual flexgripplus sass sm 1.0," pp. 1–67, May 2020. [Online]. Available: https://doi.org/10.5281/zenodo.3819313

[16] J.-D. Guerrero-Balaguera, J. E. R. Condia, and C.-F. Moreno-Manrique, "Open source sfu user's manual," Jul. 2020. [Online]. Available: https://doi.org/10.5281/zenodo.3934441

[1] https://opencores.org/projects/special_functions_unit