SFU-Driven Transparent Approximation Acceleration on GPUs

Ang Li*, Shuaiwen Leon Song†, Mark Wijtvliet*, Akash Kumar§, and Henk Corporaal*

*Eindhoven University of Technology, The Netherlands. Email:{ang.li, m.wijtvliet, h.corporaal}@tue.nl

†Pacific Northwest National Laboratory, USA. Email:shuaiwen.song@pnnl.gov

§Technische Universität Dresden, Germany. Email:akash.kumar@tu-dresden.de

ABSTRACT

Approximate computing, the technique that sacrifices certain amount of accuracy in exchange for substantial performance boost or power reduction, is one of the most promising solutions to enable power control and performance scaling towards exascale. Although most existing approximation designs target the emerging data-intensive applications that are comparatively more error-tolerable, there is still high demand for the acceleration of traditional scientific applications (e.g., weather and nuclear simulation), which often comprise intensive transcendental function calls and are very sensitive to accuracy loss. To address this challenge, we focus on a very important but long ignored approximation unit on today's commercial GPUs — the special-function unit (SFU), and clarify its unique role in performance acceleration of accuracy-sensitive applications in the context of approximate computing. To better understand its features, we conduct a thorough empirical analysis on three generations of NVIDIA GPU architectures to evaluate all the single-precision and double-precision numeric transcendental functions that can be accelerated by SFUs, in terms of their performance, accuracy and power consumption. Based on the insights from the evaluation, we propose a transparent, tractable and portable design framework for SFU-driven approximate acceleration on GPUs. Our design is software-based and requires no hardware or application modifications. Experimental results on three NVIDIA GPU platforms demonstrate that our proposed framework can provide fine-grained tuning for performance and accuracy trade-offs, thus facilitating applications to achieve the maximum performance under certain accuracy constraints.

Keywords

Approximate Computing, GPU, Special-Function-Unit, Performance/Energy/Accuracy Trade-offs, Program Transformation

ICS '16, June 01-03, 2016, Istanbul, Turkey

DOI: http://dx.doi.org/10.1145/2925426.2926255

1. INTRODUCTION

Despite the conventional belief that being exact remains the default attribute for computing, for many promising applications, such as big data, machine learning and multimedia processing, extremely accurate compliance of the produced results is often not an essential requisite. This undoubtedly offers new opportunities for application speedup or the associated power reduction at the expense of modest precision loss [1]. Such precision loss is only acceptable when it is within the tolerance range of the user-defined quality-of-service (QoS) [2], which heavily depends on the specific application domain. Besides, many of these applications are data-parallelism intensive, making them well-suited candidates for the emerging general-purpose GPU computation (GPGPU) [3]. Concerning the above reasons, approximate computing has become an attractive research topic for GPUs [4, 5, 6, 7, 8, 9].

However, most existing GPU approximation designs are targeted for data-intensive applications [4, 5, 7, 9], which are comparatively more error-tolerable. Furthermore, they primarily rely on the spatial or temporal locality among the nearby-data or the consecutive functions so as to approximate the requested data/computation based on their neighboring [4, 5, 8, 9] or locally stored historical values [5, 6, 7, 9]. Such approaches, although quite efficient, may commit uneven errors across data elements or even catastrophic failures since the locality is not always held and the distortion to the final results could be considerable. Moreover, for the numerical-intensive scientific applications (e.g., various simulation and molecular dynamics) that are usually sensitive to accuracy loss, the current techniques are often not suitable. This is because even a relatively smaller error introduced in an intermediate result may potentially propagate and be significantly amplified when such applications are deployed in a supercomputer environment with thousands of working GPUs [10, 11]. Therefore, gaining performance while offering lower but still tractable assurance on accuracy loss becomes the major obstacle for applying approximation techniques to accuracy-sensitive applications on GPUs.

To address this challenge, we explore a very important but often ignored approximation unit on GPUs — the special-functional unit (SFU), and unveil its crucial role in performance acceleration for accuracy-sensitive scientific applications in the context of approximate computing. To better understand its approximation potentials, we first evaluate all the nine single-precision and four double-precision numeric

transcendental functions that could be accelerated by SFUs, in terms of performance, accuracy and power. Using the insights, we then leverage the GPU SIMT execution model to dynamically partition warps into executing two versions of the numerical computation: an accurate but slower version and a faster but approximate version (i.e., using SFUs), and then tune this partition ratio to control the trade-offs between the performance and accuracy, or power and accuracy. This software approach successfully introduces a relatively large, uniform and fine-grained tuning space. To accompany this design, we also propose an efficient heuristic searching method to quickly locate the optimal partition ratio that delivers the best performance under user-defined QoS. Finally, we compact the approach and its searching method into a transparent, tractable and portable SFU-centric approximate acceleration framework, which is then validated on multiple GPU architectures for its effectiveness. This paper thus makes the following contributions:

- This is the first work that specifically focuses on unleashing the approximation potentials of SFU on GPU. We explore its design, implementation, and fine-grained invocation methods. Also, we exhaustively evaluate the transcendental functions that can be accelerated by SFUs in terms of their latency, throughput, accuracy, resource cost, power, energy and the number of different operations contained.

- By leveraging the GPU SIMT execution model, we propose a runtime warp-partition method to introduce a fine-grained and nearly-linear tuning space for the performance-accuracy trade-offs on GPUs. This approach is well-suited for the scientific applications that enforce high accuracy constraints.

- Based on this approach, we propose a transparent, tractable and portable design framework to automatically tune the performance and accuracy of a GPU application and returns the best attainable performance subjecting to user-defined QoS. This framework can be integrated into the GPU compiler toolchain, hence bringing cheap, instant and significant performance gain with tractable assurance on accuracy loss.

- This is the first work to exploit hardware warp-slot id for fine-grained performance tuning and is the first to accelerate double-precision computation on GPUs via SFU-driven approximations.

2. BACKGROUND

SM Architecture: A GPU processor is composed of several *streaming-multiprocessors*[1] (SMs), sharing a L2 cache and DRAM controllers via a crossbar interconnection network. Shown in Figure 1, an SM features a number of *scalar processor* cores (SPUs[2]), each of which contains a single-precision floating-point unit and an integer arithmetic/logic unit — both units are fully pipelined. Additionally, SM

[1]In this paper, we adopt NVIDIA terminology for convenience. However, similar ideas and approaches can be applied to other GPUs as well.

[2]To distinguish scalar-processors from single-precision, we use SPU to denote scalar-processor-unit while SP to denote single-precision.

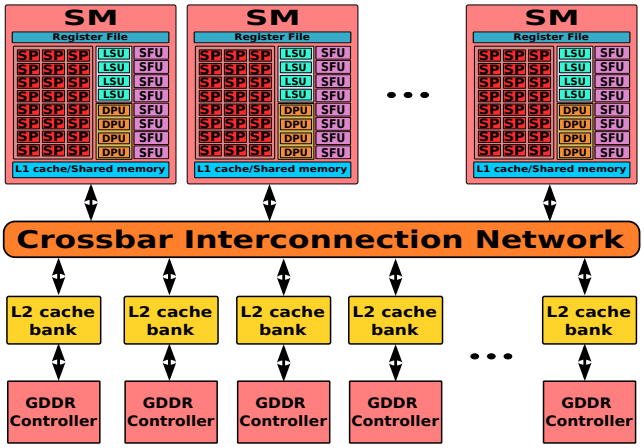

Figure 1: GPU Architecture

consists of two other function units, including the Double-Precision Units (DPUs) for double-precision (DP) floating-point calculation, and the Special-Function Units (SFUs) for processing transcendental functions and texture-fetching interpolations. Other components, such as the register files, load-store units (LSUs), scratchpad memory (i.e., shared memory), and various caches for on-chip data buffering also reside on the SM.

Execution Model: The execution model of GPU is evolved from the SIMD model, known as *single-instruction-multiple-threads* (SIMT) [12]. The function being executed on GPU is called a *kernel*, which initiates thousands of simultaneous lightweight GPU threads. These threads group into a number of blocks, called *Thread Blocks* or *Cooperative-Thread-Arrays* (CTAs). Threads within a CTA further form a batch of execution groups, called *warps*. A warp is the basic unit for instruction dispatching and issuing on an SM. Threads in a warp execute identical instruction stream upon different data in lockstep. Significant overhead may occur when warps are forced to diverge due to branching [13].

GPU supports multi-issuing and multi-dispatching. During execution, the dual- or quad-warp schedulers select two or four ready warps (with two independent instructions per warp [14]) to dispatch onto the different function units (e.g., SPUs and SFUs). Most instructions are accomplished by SPUs only. However, DPUs and SFUs can offer extra processing bandwidth, the ability to process special functions (e.g., transcendental functions), or additional precision (e.g., double precision) when instructions for different function units are independent. Although these features are useful, it is still challenging for users to utilize all three types of function units in a balanced way. This is the reason why multi-issuing/dispatching mixed instructions to these function units remains critical for GPU performance [15, 16].

3. SFU DESIGN AND IMPLEMENTATION

In this section, we zoom in on SFU and explore its design and operation. Based on the experiments on real hardware, we have observed interesting features of SFU implementation for approximating both SP and DP floating-point computation, which has not been covered by previous work.

Design: To accelerate the commonly-used transcendental functions in numeric routines as well as the texture-fetching interpolation operations from graphic applications, NVIDIA

Table 1: Invoking SP Transcendental Functions via CUDA and PTX APIs

Func.	CUDA API Intrinsics		PTX API Instructions	
	SP-Accurate Version	SFU-Approximate Version	SP-Accurate Version	SFU-Approximate Version
x/y	x/y	__fdividef(x,y) & –ftz=true	div.rn.f32 %f3,%f1,%f2;	div.approx.ftz.f32 %f3,%f1,%f2;
$1/x$	1/x	Non-Provided	rcp.rn.f32 %f2,%f1;	rcp.approx.ftz.f32 %f2,%f1;
$\sqrt{x}$	sqrtf(x)	Non-Provided	sqrt.rn.f32 %f2,%f1;	sqrt.approx.ftz.f32 %f2,%f1;
$1/\sqrt{x}$	1/sqrtf(x)	rsqrtf(x) & –ftz=true	sqrt.rn.f32 %f2,%f1; rcp.rn.f32 %f3,%f2;	rsqrt.approx.ftz.f32 %f2,%f1;
x^y	powf(x)	__powf(x) & –ftz=true	Very Complex	lg2.approx.ftz.f32 %f3,%f1; mul.ftz.f32 %f4,%f3,%f2; ex2.approx.ftz.f32 %f5,%f4;
e^x	expf(x)	__expf(x) & –ftz=true	Very Complex	mul.ftz.f32 %f2,%f1, 0f3FB8AA3B; ex2.approx.ftz.f32 %f3,%f2;
$\log(x)$	logf(x)	__logf(x) & –ftz=true	Very Complex	lg2.approx.ftz.f32 %f2,%f1; mul.ftz.f32 %f3,%f2, 0f3F317218;
$\sin(x)$	sinf(x)	__sinf(x) & –ftz=true	Very Complex	sin.approx.ftz.f32 %f2,%f1;
$\cos(x)$	cosf(x)	__cosf(x) & –ftz=true	Very Complex	cos.approx.ftz.f32 %f2,%f1;

Table 2: Invoking DP Transcendental Functions via CUDA and PTX APIs

Func.	CUDA API Intrinsics		PTX API Instructions	
	DPU-Accurate Version	SFU-Approximate Version	DPU-Accurate Version	SFU-Approximate Version
x/y	x/y	Non-Provided	div.rn.f64 %fd3,%fd1,%fd2;	rcp.approx.ftz.f64 %fd3,%fd2; mul.f64 %fd5,%fd3,%fd1;
$1/x$	1/x	Non-Provided	rcp.rn.f64 %fd2,%fd1;	rcp.approx.ftz.f64 %fd2,%fd1;
$\sqrt{x}$	x/y	Non-Provided	sqrt.rn.f64 %fd2,%fd1;	rsqrt.approx.ftz.f64 %fd2,%fd1; rcp.approx.ftz.f64 %fd3,%fd2;
$1/\sqrt{x}$	1/sqrt(x)	Non-Provided	sqrt.rn.f64 %fd2,%fd1; rcp.rn.f64 %fd3,%fd2;	rsqrt.approx.ftz.f64 %fd2,%fd1;

Table 3: Experiment Platforms. "Plat." stands for platform. "Dri./Rtm." stands for CUDA Driver/Runtime Version.

Plat.	GPU	Architecture	Code	CC.	Frequency	SMs	SPs	SFUs	Warp Slots	Memory Bandwidth	Dri./Rtm.
1	GTX-570	Fermi	GF-110	**2.0**	1464 MHz	15	32	4	48	152 GB/s	6.5/6.5
2	GTX-TitanZ	Kepler	GK-110	**3.5**	824 MHz	13	192	32	64	288 GB/s	7.5/6.5
3	GTX-750Ti	Maxwell	GM-107	**5.0**	1137 MHz	5	128	32	64	86.4 GB/s	7.5/6.5
4	Jetson TK1	Kepler	GK-20A	**3.2**	852 MHz	1	192	32	64	17 GB/s	7.0/7.0
5	Jetson TX1	Maxwell	GM-20B	**5.3**	998 MHz	2	128	32	64	25.6 GB/s	7.0/7.0

GPUs since Fermi begin to integrate an array of special hardware accelerators in the SMs, called Special-Functional Units (SFUs). The numeric transcendental functions include *sine, cosine, division, exponential, power, logarithm, reciprocal, square-root* and *reciprocal square-root* [16, 17]. Their implementations are based on the quadratic interpolation method through *enhanced-minmax-approximations* in the hardware design [18]. Such an approximation process is accomplished in three steps: (1) a **preprocessing** step to reduce the input argument into a dedicated range, (2) a **processing** step to perform quadratic polynomial approximation on the reduced argument via table look-up for the required coefficients, and (3) a **postprocessing** step to reconstruct, normalize and round the result to its original argument domain. Please refer to [18, 19] for more details.

Implementation: For **single-precision (SP)** floating-point computation, CUDA provides both an accurate implementation following IEEE-754 standard (labeled as **SPU version**) and an approximate implementation (labeled as **SFU version**) for the 9 transcendental functions, shown in Table.1. As can be seen, only 7 of the 9 transcendental functions have CUDA intrinsics. For the lower-level *Parallel-Thread-Execution* (PTX) assembly representation, we find that the SFU version for each transcendental function is comprised of a single or several SFU instructions, while the SP version is often a complex software-simulated procedure running on SPUs (or a procedure making modifications to the gross results obtained from the SFUs).

To initiate the SFU version, two most naive approaches are (1) invoking the corresponding CUDA intrinsics (e.g., *__sinf* [20] in Table 1) within the program, or (2) specifying the compiler option "*-use_fast_math*" to force the utilization of the SFU version in the generated *cubin* binary. However, using "*-use_fast_math*" applies to the entire program, which prevents the transcendental functions to benefit from fine-grained tuning. For instance, "*-use_fast_math*" option implies "*-ftz=true*", which will flash all the denormal values (i.e., floating-point numbers that are too small to be representable in the current precision[3]) in the program to zero. Although this will speedup the processing for transcendental functions on SFUs, it also increases the inaccuracy of the normal SP computation. If we make "*-ftz=false*", it will however, decrease the maximum speedup for SFUs. Thus, "*-use_fast_math*" is not suitable for fine-grained performance tuning. On the other hand, using CUDA API intrinsics to exploit SFU also has two problems: (1) Not all of them are supported, e.g., $1/x$ and $\sqrt{x}$; and (2) the flash-to-zero (*-ftz*) configuration cannot be set/unset by the CUDA intrinsics. Table.1 shows that only the PTX instructions can provide the full coverage for all the 9 transcendental functions, and the flexibility to enable/disable the *-ftz* without affecting other transcendental functions and regular computation. We will further discuss this matter in Section 5.1.

Regarding **double-precision (DP)** floating-point computation shown in Table.2, **no** CUDA intrinsics are offered for approximating the nine functions. However, at the PTX assembly level, we discover that *reciprocal* ($1/x$) and *reciprocal-square-root* ($1/\sqrt{x}$) can be approximated for acceleration via

[3]Also known as underflow, it is $\pm 2^{-126}$ for SP and $\pm 2^{-1022}$ for DP.

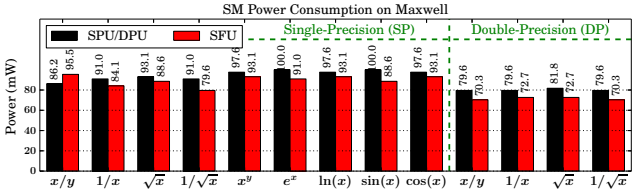

Figure 2: Power Consumption Measured on Jetson TX-1.

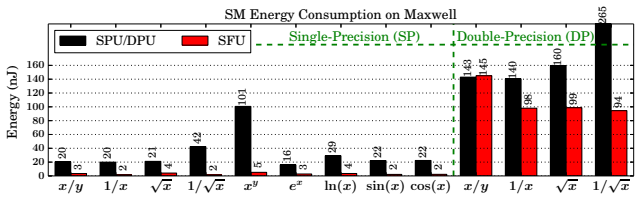

Figure 3: Energy Consumption for Jetson TX-1.

SFUs. This is confirmed by checking the usage of "MUFU" instructions in the generated *cubin* binary, which are the instructions specifically targeted for SFU usage. With $1/x$ and $1/\sqrt{x}$, two other functions *div* and *square-root* can also be implemented indirectly. Therefore, there are totally four transcendental functions that can be approximated by SFUs for DP computation. To the best of our knowledge, no existing literature or tutorial has discussed how to employ these four SFU-based approximations to accelerate DP-based applications, as there are no support from either CUDA intrinsics or compiler options. We will demonstrate that, if they are properly used, significant performance improvements can be achieved for applications with intensive DP computation (Section 5.3). Note that "*ftz*" is mandatory for these approximate functions in DP, i.e., the "*.ftz.*" suffix of the PTX instructions in Table 2. We label the DPU-based implementation as **DPU version**.

4. MEASUREMENT AND OBSERVATION: DETAILED EXPLORATION OF SP, DPU AND SFU

First, we would like to study the runtime characteristics of the GPU transcendental functions (have not been explored previously) before they can be properly deployed into the real applications. In this section, we design dedicated microbenchmarks to measure the *latency*, *relative-error*, *register usage*, *SPU/SFU/DPU operations contained*, *throughput per SM* as well as *power* and *energy cost* for the 9 SP and 4 DP transcendental functions. This information will serve as the motivation of our proposed design.

Our evaluation platforms are listed in Table 3. Three generations of NVIDIA GPUs (*Platform 1,2,3*) including Fermi, Kepler and Maxwell, are used for testing the *function latencies*. For *relative-error*, we perform both SPU/DPU and SFU-based transcendental calculation over 100,000 random data and compare their results to the versions offered by the host Intel CPU. The average difference over the elements is then used as the relative-error. *Register usage* is collected based on the statistics reported by the CUDA compiler. For the *operation throughput per SM*, sufficient transcendental function calls are initiated in the microbenchmark and all of them are completely independent with each other to fully exploit the instruction-level-parallelism (ILP) of the hardware. We observe the profiled curve until the values become stable, which are then used as the maximum sustainable throughput for that operation. These values are then divided by the SM number to get the per-SM throughput. All the above results are shown in Table 4 and Table 5 for SP and DP, respectively.

The existing approaches to obtain GPU **power consumption** are often based on either simulator approximation (e.g., *GPUWattch* [21]) or the power-draw value reported by *nvidia-smi* [22]. However, neither of them reports real GPU power consumption. In this work, we propose a new approach that is more accurate and reliable. It leverages the latest Maxwell-based NVIDIA Jetson TX-1 GPU (*Platform 5* in Table 3, which is mainly designed for embedded utilization) and measures the power of the board's computation module only (i.e., the quad-core CPU and dual-SM GPU). This is achieved by measuring the voltage alteration of the resistance $R264$, which is in series with the computation module when a GPU kernel is running, and then compare it with the baseline state when the compute module is idle. Inside the kernel, we use a loop to keep the transcendental functions repeatedly being executed until the average voltage of the resistance converges to a steady value. As the voltage change is quite small, we also design an amplifier circuit so that such small voltage change can be sensitively tracked by an oscilloscope[4]. The measured power results are shown in Figure 2.

We also tried to measure the power of the Kepler-based Jetson TK-1 board (*Platform 4* in Table 3). However, we found that there is no series resistance to the core module for this board. The only one that seems promising (i.e., resistance *R5C11*) is in series with the entire board (including GDDR, fan and other I/O modules), so the voltage is quite hard to stabilize. Thus, we do not show the TK-1 power results in this paper. With the measured power, we can calculate the energy consumption with the measured function latencies. The energy results are shown in Figure 3.

Table 4 and Table 5 show that SFU itself only injects small error in the individual function calculation. However, this small error can quickly propagate and get amplified across the program semantics, causing intolerable accuracy for some applications. Also, dramatic differences in latency and throughput have been observed between SPU and SFU versions on both Kepler and Maxwell platforms. Furthermore, we find that latency is not as good as throughput per SM (T/M) for indicating the real performance difference between the two versions. For example, $\mathbf{ln(x)}$'s throughput difference on Kepler is as high as 9.9x, while the latency difference is only 37%. This implies that the SFU appears to be a super-pipelined unit. For power and energy, Figure 2 and 3 show that (1) the power consumption using SPU/DPU is slightly higher than that using SFU, except for x/y in SP; and (2) due to the huge performance differences between the SP and DP versions on the Maxwell platform, the overall energy consumption of DP versions (including their SFU approximations) is significant higher than that of

[4]The resistance $R264$ is in series with the compute module. The voltage difference measured by the oscilloscope in a long steady state, after being divided by the amplification factor, is then divided by the resistance value $R264 = 0.005\Omega$ to obtain the electric current of the compute module. The current is then multiplied by the measured Vdd_mod to acquire the actual GPU power consumption.

Table 4: SPU Version vs. SFU Version Characterization for SP. Ver. stands for the version. Lat. is the measured latency in clock cycles. Rel-Err is the relative-error with respect to CPU results. Reg is the register consumption. F/P/D is the number of operations executed by SFU, SPU and DPU respectively in the function computation. T/M is the operation throughput per SM in the unit of Gop/s.

Func.	Arch.	Ver.	Lat.	Rel-Err.	Rg.	F/P/D	T/M	Func.	Arch.	Ver.	Lat.	Rela-Err.	Reg.	F/P/D	T/M
x/y	Fermi	SPU	2335	0	12	1/16/0	1.7	$1/x$	Fermi	SPU	1692	0	13	1/4/0	2.8
		SFU	2068	2.3433E-08	10	1/1/0	5.7			SFU	1651	1.1266E-08	8	1/0/0	5.7
	Kepler	SPU	1098	0	13	1/14/0	6.0		Kepler	SPU	715	0	14	1/4/0	7.9
		SFU	981	2.3433E-08	10	1/1/0	24.0			SFU	597	1.1266E-08	8	1/0/0	23.2
	Maxwell	SPU	236	0	14	1/14/0	4.1		Maxwell	SPU	219	0	14	1/4/0	4.7
		SFU	36	2.3433E-08	10	1/1/0	25.3			SFU	21	1.1266E-08	10	1/0/0	26.6
$\sqrt{x}$	Fermi	SPU	1708	0	10	1/6/0	2.6	$1/\sqrt{x}$	Fermi	SPU	1728	0	13	2/10/0	1.4
		SFU	1651	3.0763E-08	8	2/0/0	2.9			SFU	1651	2.7610E-08	8	1/0/0	5.7
	Kepler	SPU	711	0	10	1/6/0	6.4		Kepler	SPU	864	0	14	2/10/0	3.8
		SFU	613	3.0763E-08	8	2/0/0	12.9			SFU	597	2.7610E-08	8	1/0/0	23.2
	Maxwell	SPU	226	0	10	1/6/0	5.0		Maxwell	SPU	464	0	14	2/10/0	2.5
		SFU	47	3.0763E-08	10	2/0/0	14.8			SFU	21	2.7610E-08	10	1/0/0	27.1
x^y	Fermi	SPU	6073	3.0822E-08	14	3/59/0	8.0	e^x	Fermi	SPU	1681	2.3937E-08	10	2/7/0	1.9
		SFU	2110	8.0587E-08	10	2/1/0	43.1			SFU	1655	4.0603E-08	8	1/1/0	5.7
	Kepler	SPU	1496	3.0822E-08	15	3/60/0	9.1		Kepler	SPU	700	2.3937E-08	8	2/7/0	4.5
		SFU	997	8.0587E-08	10	2/1/0	156.7			SFU	612	4.0603E-08	8	1/1/0	23.4
	Maxwell	SPU	1029	3.0822E-08	16	3/60/0	3.8		Maxwell	SPU	160	2.3937E-08	8	2/7/0	4.7
		SFU	56	8.0587E-08	10	2/1/0	65.8			SFU	31	4.0603E-08	10	1/1/0	20.6
$\ln(x)$	Fermi	SPU	1779	4.6541E-09	11	1/19/0	1.2	$\sin(x)$	Fermi	SPU	1727	8.7079E-09	13	0/17/0	1.1
		SFU	1649	6.3260E-07	8	1/1/0	5.7			SFU	1660	9.6523E-07	8	1/0/0	5.7
	Kepler	SPU	834	4.6541E-09	11	1/19/0	2.1		Kepler	SPU	804	8.7079E-09	13	0/17/0	2.9
		SFU	608	6.3260E-07	8	1/1/0	22.9			SFU	602	9.6523E-07	8	1/0/0	25.0
	Maxwell	SPU	298	4.6541E-09	11	1/20/0	1.8		Maxwell	SPU	222	8.7079E-09	17	0/17/0	2.3
		SFU	38	6.3260E-07	10	1/1/0	26.3			SFU	25	9.6523E-07	10	1/0/0	22.5
$\cos(x)$	Fermi	SPU	1740	1.4455E-08	13	0/18/0	1.0								
		SFU	1646	1.1584E-06	8	1/0/0	5.7								
	Kepler	SPU	824	1.4455E-08	13	0/18/0	2.9								
		SFU	600	1.1584E-06	8	1/0/0	25.0								
	Maxwell	SPU	229	1.4455E-08	17	0/18/0	2.1								
		SFU	25	1.1584E-06	10	1/0/0	22.5								

Table 5: DPU Version vs. SFU Version Characterization for DP.

Func.	Arch.	Ver.	Lat.	Rel-Err.	Rg.	F/P/D	T/M	Func.	Arch.	Ver.	Lat.	Rela-Err.	Rg.	F/P/D	T/M
x/y	Fermi	DPU	1889	0	19	1/0/15	7.8	$1/x$	Fermi	DPU	2485	0	16	1/0/8	10.5
		SFU	1204	2.5561E-07	10	1/0/1	28.8			SFU	2166	2.5545E-07	8	1/0/0	42.3
	Kepler	DPU	1236	0	20	1/0/15	8.4		Kepler	DPU	774	0	14	1/0/10	13.0
		SFU	1104	2.5561E-07	10	1/0/1	30.4			SFU	902	2.5545E-07	8	1/0/0	44.9
	Maxwell	DPU	1793	0	20	1/0/15	2.2		Maxwell	DPU	1761	0	13	1/0/10	3.4
		SFU	2057	2.5561E-07	10	1/0/1	7.9			SFU	1346	2.5545E-07	9	1/0/0	11.7
$\sqrt{x}$	Fermi	DPU	2319	0	13	1/0/13	8.5	$1/\sqrt{x}$	Fermi	DPU	2551	0	16	2/0/21	5.3
		SFU	2171	2.8951E-07	10	2/0/0	42.1			SFU	2165	2.2110E-07	10	1/0/0	42.4
	Kepler	DPU	949	0	14	1/0/13	9.1		Kepler	DPU	1296	0	14	2/0/23	7.0
		SFU	921	2.8951E-07	8	2/0/0	44.3			SFU	897	2.2110E-07	8	1/0/0	44.9
	Maxwell	DPU	1947	0	14	1/0/13	2.4		Maxwell	DPU	3317	0	14	2/0/23	1.6
		SFU	1355	2.8951E-07	9	2/0/0	11.7			SFU	1340	2.2110E-07	9	1/0/0	11.7

the SP versions, in spite of their lower power. These observations motivate us to propose our design for tackling the performance-accuracy trade-offs using SFU approximation on GPUs, which will be discussed next.

5. SFU-DRIVEN APPROXIMATION ACCELERATION: A SOFTWARE APPROACH

From the experiments, we observe that SFUs can significantly boost the performance for transcendental-function intensive applications. But meanwhile their approximations also introduce errors that are sometimes too large to be accepted. Although Table 4 and 5 demonstrate that SFUs only introduce relatively small errors in each transcendental computation, the process about how these small errors propagate and eventually accumulate to intolerable results is often complicated. This is the reason why within a single thread context choosing the proper functions to approximate while keeping the overall error under control remains quite difficult [23, 24, 25]. Additionally, compared with the data-intensive applications, the numerically intensive applications are often much more sensitive to accuracy. Therefore, a fine-grained accuracy tuning scheme is in great need so that the most desirable performance can be achieved under more strict accuracy requirement. Ideally, such fine-grained tuning range should be within a small accuracy offset and comprises consecutive accuracy tuning points. In other words, applied techniques should be controlled to some extent and not cause *significant accuracy difference between two discrete tuning points* (e.g., techniques such as loop perforation [25] and specific optimization transformations [4] often cause large accuracy differences between tuning points).

GPU offers massive identical threads operating upon different data elements. If part of the threads on GPU could execute the approximate version while the remaining ones process the accurate version (such a design paradigm is labeled as **horizontal design**), it essentially opens the door to a new design direction that is perpendicular to the conventional ones, which seek to choose the appropriate functions for approximation in a single-thread context (labeled as **vertical design**). Comparatively, the horizontal design should have a much simpler and more tractable accuracy-performance trade-off relationship than the vertical one, as the error effects are similar from various threads but very different across functions. We will demonstrate our exploration

on the trade-off relation between performance and accuracy for the proposed horizontal design in Section 5.3. In fact, the horizontal design is one of the most highlighted features that differentiates a GPU from the CPU family, which can also be applied to resolve other design trade-offs, such as the one between thread volume and cache-performance [26].

Furthermore, the parallelism granularity is an important issue for enabling the horizontal design. Since warp divergence incurs significant overhead, instead of working at the fine-grained thread level, we focus on the medium-grained warp level to reduce the design space and eliminate the warp-divergence overhead. For the rest of this paper, we will demonstrate how to *practically and properly schedule the candidate warps between the accurate but slower SPU/DPU version and the approximate but faster SFU version.* More specifically, we will answer the following questions:

- How to implement the SPU/DPU and SFU versions of transcendental functions in a flexible way?
- How to control the approximation degree?
- How to decide the optimal warp scheduling so that the best performance can be achieved under a QoS constraint?

5.1 Flexible SPU/DPU/SFU APIs Invocation

There are three types of APIs that can be applied for approximating transcendental functions on GPU: *CUDA*, *PTX* and *SASS*. PTX routine is an intermediate machine-independent bytecode that is translated from CUDA program, and can be parsed and executed on-the-fly at runtime (or just-in-time). *Shader-Assembly* (SASS) code is generated by assembling PTX bytecode through *ptxas* assembler, which is the machine-dependent assembly code for GPU. Modifying SASS code requires enormous knowledge about the detailed hardware implementation, which is often concealed by the vendors. Migration is also very difficult for SASS code because it is hardware specific. Most importantly, there is no official SASS assembler. Therefore, SASS is excluded as an option to implement approximation.

On the other hand, PTX APIs are the specific PTX instructions, as listed in the right side of Table 1. As previously discussed, for the SFU version, all the 9 transcendental functions can be approximated via PTX APIs in the following format with less than three instructions:

```
function.approx.ftz.f32 %f3, %f1, %f2;
```

"*approx*" stands for the approximate version, "*ftz*" indicates that flashing-to-zero is true for denormal values, and "*f32*" is for SP. However, for the accurate SPU version, we discover that only *div, rcp, sqrt* and *rsqrt* can be expressed via 1 to 2 PTX instructions. The other five transcendental functions require complex representation if using PTX instructions. For instance, for *sin* and *cos*, the SPU-based implementations contain more than 140 lines of PTX code without counting the loops inside. Manipulating such a big block of PTX routines while keeping consistent with its upper and lower context (e.g., register naming, memory consistency, etc) remains very tedious and error-prone. Therefore, we cannot implement both accurate and approximate transcendental computation on GPU solely with PTX instructions.

As discussed in Section 3, all the SPU-based CUDA APIs have their original expressions, shown in the left side of Table 1. But for the SFU approximation, *reciprocal* and *square-root* do not have their CUDA intrinsics, unless recompiling the entire source file with "*-use_fast_math*". However, this

```
//CUDA API to implement accurate SPU version
float expRT = expf(-R*T);
//PTX API to implement approx SFU version with denormal
asm("mul.ftz.f32 %0, %1, 0f3FB8AA3B;":"=f"(tmp):"f"(-R*T));
asm("ex2.approx.ftz.f32 %0, %1;":"=f"(expRT):"f"(tmp));
```

Listing 1: CUDA-based SPU version vs. PTX-based SFU version.

is too coarse-grained and may affect other kernels unexpectedly. Moreover, one cannot flexibly control the denormal behavior for a single function by using CUDA intrinsics in the SFU approximation version. Specifying *-ftz=true/flase* would change all the kernels in the current source file.

To summarize, CUDA APIs cover all the accurate SPU versions and show the convenience for program transformation, while PTX APIs cover the entire SFU versions and offer the maximum flexibility for approximation. Therefore, our design combines the two via the embedded PTX [27]. Listing 1 shows the two versions of the *exp* function.

Note that there is another strong reason for implementing the SPU versions via PTX APIs. As shown in Table 2, there is no CUDA intrinsics offered at all for the DP approximation. This paper proposes the first SFU-driven approximation approach for DP computation via PTX APIs on GPU.

5.2 Control Approximate Degree Horizontally

Now we need a way to control the approximation degree so that the trade-offs between performance and accuracy are subject to QoS. Ideally, to allow fine-grained tuning, the approximation degree range should be relatively large (within in a certain accuracy expectation though) while the gap between discrete degrees remains small. In our horizontal design, this is achieved by *tuning the partition of the homogeneous warps between the SPUs/DPUs and the SFUs.*

Our basic approach is that we set a threshold for the approximate degree (labeled as λ) at the beginning of the kernel. During the consequent execution, in case a transcendental function is invoked,

- for warps with hardware index less than the threshold ($warp\_id < \lambda$), they perform the SFU version via embedded PTX instructions.
- for warps with hardware index larger than or equal to the threshold ($warp\_id \geq \lambda$), they perform the SPU/DPU version via CUDA APIs.

The warp index used here is not the common software warp id in the programming context calculated by dividing the thread id with the warp size, but essentially the hardware warp-slot id of a GPU SM, which can be acquired by fetching from the special register – "*%warpid*" via PTX instructions. There are three reasons for using the hardware warp id in our design: (1) The hardware warp ids contain a larger tuning range, since its corresponding warp-slots are for an entire SM while the software warp ids are only for a CTA. More specifically, an SM usually accommodates multiple CTAs (up to 16 for Kepler and Maxwell), so tuning according to hardware warp-slots is more fine-grained. For example, assume a SM has 16 CTAs and each contains 4 warps. Therefore, all the warp-slots of the SM are occupied and the occupancy is 1. If software warp id is used to partition the warps, the tuning range is from 0 to 4. However, if hardware warp slot id is applied, the tuning range becomes from 0 to 64 (48 for Fermi, see Table 3). (2) Using hardware warp slot ids can achieve better load-balancing. Unlike

```
#define PI 3.14159265358979f
__device__ inline void BoxMuller(float& u1,float& u2){
  float r=sqrtf(-2.0*logf(u1)); float phi=2*PI*u2;
  u1=r*cosf(phi); u2=r*sinf(phi);
}
__global__ void BoxMullerGPU(float *d_Random,int nPerRng){
  const int tid=blockDim.x*blockIdx.x+threadIdx.x;
  for (int iOut=0;iOut<nPerRng;iOut+=2)
    BoxMuller(d_Random[tid+(iOut+0)*MT_RNG_COUNT],
              d_Random[tid+(iOut+1)*MT_RNG_COUNT]);
}
```

Listing 2: The Original Mersenne Kernel.

```
__device__ inline void BoxMuller_sfu(float& u1,float& u2){
  float r, t1, t2; float phi=2*PI*u2;
  asm("lg2.approx.ftz.f32 %0, %1;":"=f"(t1):"f"(u1));
  asm("mul.ftz.f32 %0, %1, 0f3F317218;":"=f"(t2):"f"(t1));
  asm("sqrt.approx.ftz.f32 %0, %1;":"=f"(r):"f"(-2.0*t2));
  asm("cos.approx.ftz.f32 %0, %1;":"=f"(u1):"f"(phi));
  asm("sin.approx.ftz.f32 %0, %1;":"=f"(u2):"f"(phi));
  u2=u2*r; u1=u1*r;
}
__global__ void BoxMullerGPU(float *d_Random,int nPerRng){
  const int tid=blockDim.x*blockIdx.x+threadIdx.x;
  unsigned warpid;
  //const bool flag=(threadIdx.x>>5)<Lambda;//soft_id
  asm("mov.u32 %0, %%warpid;":"=r"(warpid));//hard_id
  const bool flag=(warpid<Lambda);//approx degree
  if(flag){//SFU approximate version
    for(int iOut=0;iOut<nPerRng;iOut+=2)
      BoxMuller_sfu(d_Random[tid+(iOut+0)*MT_RNG_COUNT],
                    d_Random[tid+(iOut+1)*MT_RNG_COUNT]);
  }else{//SPU accurate version
    for(int iOut=0;iOut<nPerRng;iOut+=2)
      BoxMuller(d_Random[tid+(iOut+0)*MT_RNG_COUNT],
                d_Random[tid+(iOut+1)*MT_RNG_COUNT]);
  }}
```

Listing 3: Transformed Mersenne Kernel.

using software warp ids, warps are dynamically binded to the hardware warp-slots at runtime. This will average out the scenarios where some warps are always scheduled and consequently finished earlier than other warps in a CTA (i.e., the starvation problem). For example, specifying "*if* $warp\_id < 8$" using hardware warp id has almost the same performance as the scenarios such as *if* $warp\_id \geq 56$ and *if* $warp\_id < 4$ *or* ≥ 60. (3) The approximate degree is 1 warp among two consecutive tuning steps for using hardware warp-slot id, but num_CTA per SM for using software warp id. (4) Obtaining the hardware warp id can be completed in a single register-read operation. However, it requires an additional integer division (or right-shifting) instruction to gain software warp id.

Additionally, when transcendental functions are invoked inside a loop, to reduce the branching overhead (though there is no warp-divergence), we put the warp partition process outside the loop to reduce the overhead.

We demonstrate this process using an example. Listing 2 shows the the BoxMullerGPU kernel from Mersenne [28], in which *log, sqrt, sin* and *cos* functions are invoked repeatedly in a "*for*" loop. Listing 3 shows the modified SFU-driven approximate tuning kernel. As can be seen, a new approximate device function "`BoxMuller_sfu`" is generated using embedded PTX for the SFU version. Then by specifying the "`Lambda`" variable either statically at compile-time or dynamically at runtime, we are able to change the partition of warps between SFUs and SPUs, which serves as the approximate degree for fine-tuning the trade-offs between performance and accuracy.

The overhead of the proposed design is very small. Since we work at the medium-grained warp level, warp-divergence is avoided. In terms of spatial overhead, only the flag variable has a lifetime across the kernel and costs a 1-bit predicate register per thread. Furthermore, as observed in Table 4 and 5, the SFU versions always consume less registers than the SPU versions. Therefore, adding a branch should not incur additional registers (in this way the occupancy keeps unchanged). Also, because the predicate-register checking is internally supported by the GPU hardware as one stage of the pipeline, the only overhead is the issuing delay for this extra branching. Such branching overhead can be significantly mitigated by being moved outside the loop, as shown in Listing 3. Other overheads such as the delay for fetching the hardware warp id, comparing with the threshold and setting the flag (i.e., the predicate register [29]) are negligible.

5.3 Exploring Performance-Accuracy Trade-off

In this subsection, we attempt to explore the exact trade-off relationship between performance and accuracy on a wide range of scientific applications using the approach discussed previously. By doing so, we can build an strategy to answer how to decide the optimal approximate degree to achieve the best performance under certain QoS. We select applications that contain transcendental numeric functions in their kernels from Rodinia [32], Parboil [31], SDK [28], Polybench [33] and Shoc [34] benchmark suites, as listed in Table 6. We apply the program transformation discussed and plot the curves of normalized application execution time and relative-errors[5] (against the SPU/DPU version) with respect to the variation of approximate degree λ on *Platform-1,2,3* in Table 3. The figures for the 20 single-precision applications on Maxwell are shown in Figure 4. We also plot the figures for the 4 applications that contain double-precision computation in Figure 5. Since the shapes on Fermi and Kepler are similar, they are omitted here due to page limitation. From the figures, we have the following observations:

(1) Without considering the accuracy loss, our SFU-driven method demonstrates very significant performance speedup on the commodity GPU hardware (e.g., up to 5.1x for SP on Maxwell). We want to particularly highlight the DP scenarios (e.g., `CFD`, `S3D` and `COR`), as conventional wisdom believes SFU is specific for SP acceleration on GPUs. Based on our finding, other than directly programming in embedded PTX, there is currently no other software-level approach that can easily achieve such kind of DP acceleration.

(2) Although the performance gains from using SFU versions are impressive, they do incur accuracy losses. For some cases, such losses are intolerable for scientific applications (e.g., `BLA`, `CUT`, `NB`, `GUS`, `MEN`, `CFD`, `MRQ`) because the SP/DP version on GPU is already not as accurate as the CPU counterpart (see Table 4). Note that these applications are only small benchmarks or proxy applications on a single GPU that are available to us. In future, when large-scale numeric applications containing hundreds of these proxy kernels run on thousands of GPU nodes in a

[5]How to calculate the QoS for applications from various domains still misses a unified approach [35]. Here we use mean-relative-error as an example. However, other metrics can be applied to our design as well via the replacement of the error-calculation method.

Table 6: Benchmark Characteristics

Application	Description	abbr.	Domain	Hotspot Kernel	Transcendental Functions	Ref
BlackScholes	Black-scholes option pricing	BLA	Compute Finance	BlackScholesGPU	*sqrt,div,log,exp,rcp*	[28]
single	Monte Carlo single Asian option	SIN	Compute Finance	generatePaths	*sqrt,exp*	[28]
MonteCarlo	Monte-Carlo option pricing	MCO	Compute Finance	MonteCarloKernel	*exp*	[28]
cp	Coulombic potential	COP	Molecular dynamics	cenergy	*rsqrt*	[30]
cutcp	Distance-cutoff coulombic potential	CUT	Molecular dynamics	lattice6overlap	*rsqrt*	[31]
lavaMD	Particle potential and relocation	LAV	Molecular dynamics	kernel_gpu_cuda	*exp*	[32]
nbody	Fast n-body simulation	NBO	Molecular dynamics	integrateBodies	*rsqrt*	[28]
oceanFFT	FFT-based ocean simulation	OCN	Molecular dynamics	generateSpectrum	*rcp,sqrt,sin,cos*	[32]
backprop	Back propagation	BKP	Machine Learning	layerforward	*pow,log*	[32]
nn	K-nearest neighbors	KNN	Machine Learning	euclid	*sqrt*	[32]
corr	Correlation computation	COR	Linear algebra	reduce_kernel	*div,sqrt*	[33]
gaussian	Gaussian elimination solver	GUS	Linear algebra	Fan1	*div*	[32]
mersenne	Mersenne-twister random generator	MEN	Simulation	BoxMullerGPU	*log,sqrt,sin,cos*	[28]
cfd	Redundant flux computation	CFD	Simulation	comp_step_factor	*sqrt,rcp,div*	[32]
s3d	Combustion process simulation	S3D	Simulation	ratt2_kernel	*div*	[34]
mri-q	Q matrix for MRI reconstruction	MRQ	Image processing	ComputeQ_GPU	*sin,cos*	[31]
bilateralFilter	Bilateral smoothing filter	BIF	Image processing	d_bilateral_filter	*div,exp*	[28]
srad	Speckle reducing anisotropic diffusion	SRD	Image Processing	srad	*rcp,div*	[32]
grabcutNPP	GrabCut with NPP	NPP	Image Processing	GMMDataTerm	*log,exp*	[28]
imageDenoising	Image Denosing	IMD	Image Processing	KNN	*exp,rcp*	[28]

supercomputer, a relatively small distortion to a result (e.g., COP on SP and COR on DP) can result in a significantly erroneous outcome. Thus, there is a clear trade-off between performance gain and accuracy loss.

(3) Differ from our expectation that the point for best performance might be located in the middle of the curve where SFUs and SPUs are exploited simultaneously, the results show that using our approach the best performance is **almost always** achieved when all the warps are executed in SFUs while the worst when all of them are executed in SPUs/DPUs [6]. Correspondingly, the least accuracy loss occurs for pure SPUs/DPUs while the most for pure SFUs.

(4) More importantly, the results show that the **trade-off relationship between performance and accuracy with respect to approximate degree is nearly-linear**. There are five obvious exceptions here: OCN, BKP, SRD, NPP and IMD. All of them represent the scenario where kernels use SP floating-point as the basic data-type during initial computation, and then convert them to integers for the finial results of the applications. This actually matches their domains, which are image processing and machine learning.

(5) For some figures, there appears a flat region at the end of the curve where the performance and accuracy become constant (i.e., beyond the green dot line). This is because for some applications, not all the hardware warp-slots are fully occupied due to the low occupancy (e.g., cases with $ocp < 1$ in Figure 4 and 5). For example, the performance and accuracy when setting $\lambda = 49 \sim 64$ are essentially the same as those under $\lambda = 48$, if only 48 hardware warps slots are filled (i.e., $ocp = 0.75$). Therefore, the tuning space may be reduced by skipping these redundant tuning points.

5.4 Finding the Optimal Approximate Degree

In this subsection, we attempt to find the optimal approximate degree concerning the user-defined QoS. Assume the execution time function with respect to approximate degree λ is $T(\lambda)$ (e.g., the black curves in Figure 4) while the error function is $E(\lambda)$ (e.g., the red curves in Figure 4). Then the searching problem can be formalized as:

$$\min(T(\lambda)) \mid E(\lambda) \leq QoS$$

This problem is difficult to solve if $T(\lambda)$ and $E(\lambda)$ are general functions. However, as $T(\lambda)$ is negatively correlated to $E(\lambda)$ and from Figure 4 we observe that $T(\lambda)$ is monotonically decreasing with λ, the problem thus can be reformulated as

$$\max(\lambda) \mid E(\lambda) \leq QoS$$

or simply finding the root of equation $E(\lambda) = QoS$ provided that $E(\lambda)$ is continuous. However, as λ here is discrete, it is essentially the last point before the root of $E(\lambda) = QoS$.

A naive approach to find the optimal λ is to start searching from the pure-SFU version with $\lambda = 64$ or 48, and evaluate all the points along the reduction of λ until $E(\lambda) \leq QoS$. This simple approach is labeled as **SMP**.

To accelerate the searching process, based on the nearly-linear observations about $E(\lambda)$, we further propose a linear-approaching method motivated from *Newton's Method*. We use the *cutcp* application as an example. As illustrated in Figure.6, assume the QoS of this case is $0.85E-05$. To start, we first run the transformed kernel with $\lambda = 0$, which corresponds to the pure SPU/DPU version and dump the results. The performance $T(\lambda = 0)$ can also be measured if we want to calculate the speedup later. Next, we execute the kernel with $\lambda = 64$ (48 for Fermi) which corresponds to the SFU version. Similarly, we measure $T(\lambda = 64/48)$ and dump the results. Additionally, we measure the occupancy of the SFU version to reduce the search space (discussed in Section 5.3). For *cutcp*, the occupancy of the SFU version is 0.75, which indicates that the searching space is from 0 to 48. Then, by calculating the relative-error of the SFU version, we locate the position of $P0$ in Figure 6. Based on the nearly-linear observation about $E(\lambda)$, we draw a line from $P0$ to the origin and intersects it with the QoS level (the magenta line). The intersection is denoted as $V1$, where $\lambda = 30$. We run the kernel again with $\lambda = 30$ and calculate the relative-error $E(30)$, which locates $P1$. If $P1$ is less than QoS, it is the new lower-bound and we move the origin to $P1$; if $P1$ equals to the QoS, we return $P1$; if $P1$ is larger than QoS, it is the new upper-bound and we set $P1$ as the updated terminal point, as shown in Figure 6. We then connect $P1$ to the origin to form a new straight line, which intersects QoS at $V2$ where $\lambda = 26$. We run the kernel again

[6] We have observed an exception here for SIN on Fermi, in which the optimal performance point locates in the middle. This explains why later in Figure 8, SIN's SFU bar is lower.

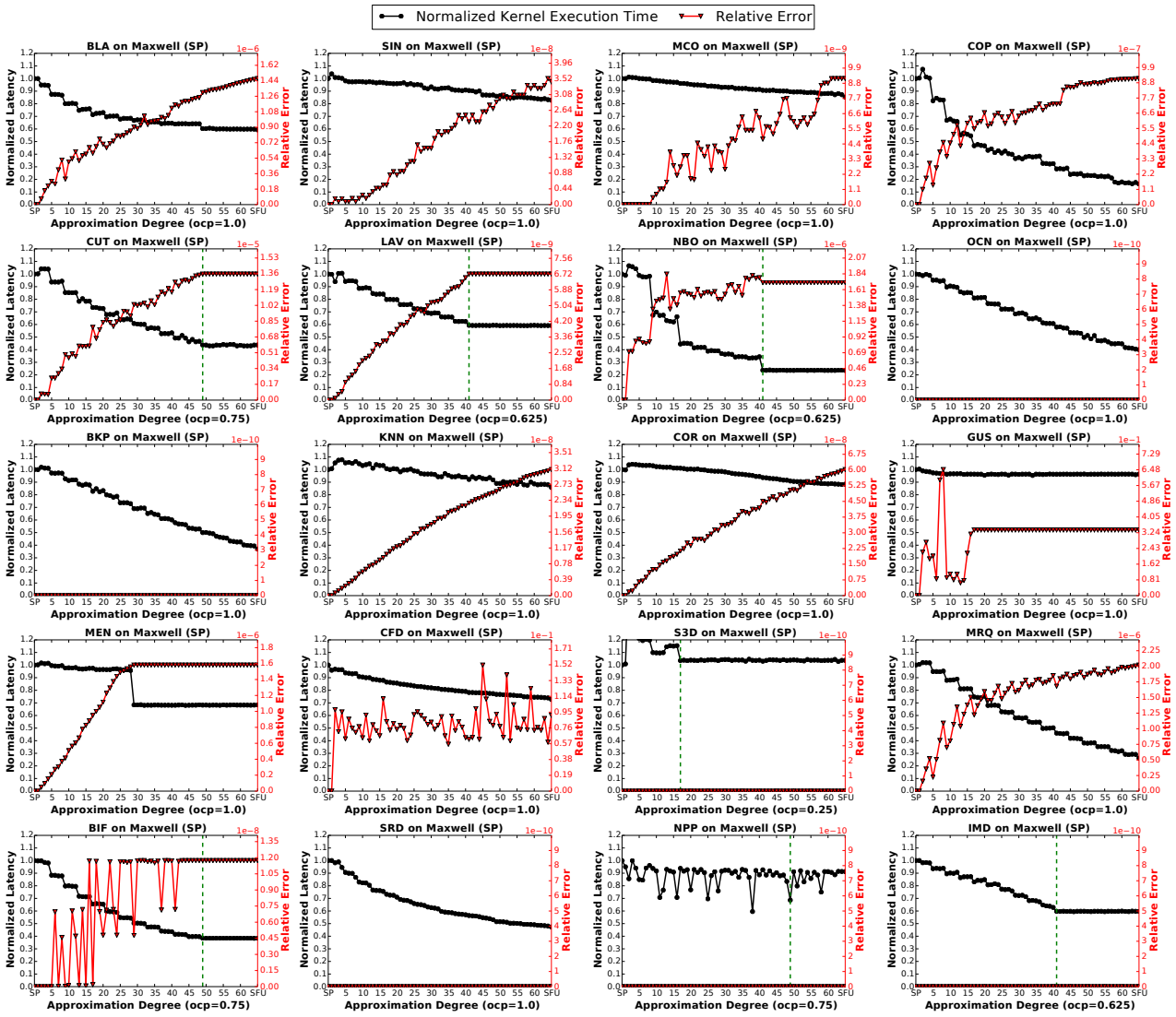

Figure 4: Performance-Accuracy Trade-offs for SP Applications on Maxwell GPU. The green dot line is based on the occupancy (i.e., *ocp* in the x-label). It indicates the border of the tuning space beyond which both the time and error curves keep steady. The relative-error is referring to the pure SPU version.

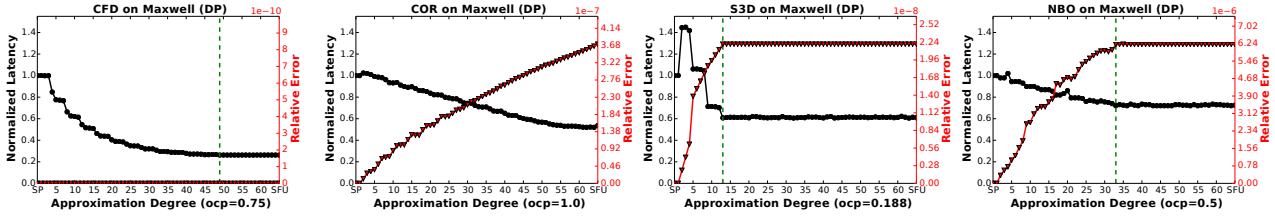

Figure 5: Performance-Accuracy Trade-offs for DP Applications on Maxwell GPU.

with $\lambda = 26$ and find that $E(26)$ at V2 happens to be the same as the QoS. Therefore, the search process terminates and returns $\lambda = 26$. Otherwise, it repeats such a process until $E(\lambda)$ is finally equal to QoS. We label this heuristic method as **HEU**. Note that this linear-approaching method converges only when $E(\lambda)$ is roughly smooth. However, this is not always the case (e.g., `NBO`, `CFD`, `BIF` in Figure.4). In these scenarios, **HEU** may get trapped in a local optimal value. Therefore, in order to ensure $E(\lambda^*) < QoS$, when it is not satisfied, we add an extra phase to assess the points along the reduction of λ from the local optimal, all the way untill $E(\lambda^*) < QoS$.

Compared to the naive *SMP* approach and the exhaustive search that traverses the entire λ searching space (labeled as **OMG**), our proposed *HEU* method can be much more efficient (will be validated in Section 7). The *HEU* method is also integrated into our SFU-driven approximation framework, which will be discussed next.

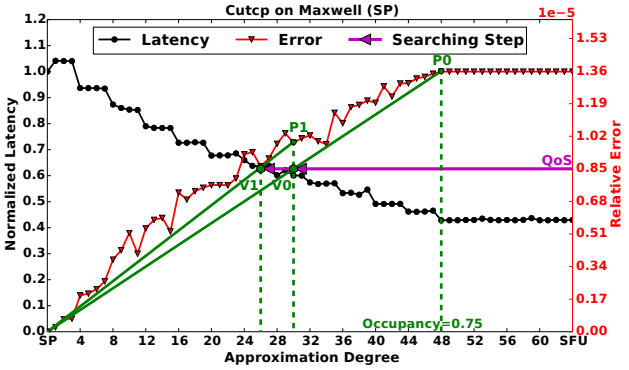

Figure 6: The proposed linear-approaching method (HEU) to locate the optimal λ for *cutcp* on a Maxwell GPU. The searching process terminates after two steps when QoS is satisfied.

Algorithm 1 Kernel Transformation (Refer to Listing 3)

```
1: Inline all device functions into the global kernel
2: Insert the flag assignments at the beginning of the kernel
3: for each transcendental function call do
4:     if the call is inside a loop then
5:         Replicate the loop
6:     else
7:         Replicate the call
8:     end if
9:     Convert the call to its SFU-based version (Table 1,2)
10:    Compact the calls into the if-else branching
11: end for
```

6. THE OVERALL FRAMEWORK

In this section, we describe the overall framework for our SFU-driven approximation acceleration design. As shown in Figure 7, when the application kernel is given, the framework first checks if it invokes any transcendental functions (SP or DP), especially the ones within a loop or nested loops. If so, it performs the program transformation discussed in Section 5.2. Such a transformation can be fully automatic as the mapping between the embedded PTX and the corresponding transcendental functions are fixed. The automatic transformation procedure is depicted in Algorithm 1. Then the framework will perform the heuristic method discussed in Section 5.4 to find the optimal λ for achieving the best performance under certain QoS. The only difference is that if the relative-error of the SFU version is less than QoS (e.g., `OCN`, `BKP`, `SRD`, `NPP` and `IMD`), it is returned immediately. Note that the "SFU/SPU result" indicated in Figure 7 is for the entire application instead of a single kernel. During the search, one can also profile the number of SPU/DPU/SFU operations performed in each step, and then combine the power/energy information in Figure 2 and 3 to calculate the power/energy consumption.

Our design is highlighted for its *transparency, tractability* and *portability*. It is **transparent** because it is a pure-software design that converts the code at compile time and runtime, so that it requires no extra efforts from both application developers and hardware designers. It also brings significant, instant and cheap speedup with guaranteed accuracy. Meanwhile, it is **tractable** because it is simple to understand and can be fully automatic (i.e., integrated into the CUDA toolchain). In addition, the horizontal approach it adopts introduces the nearly-linear performance-accuracy trade-off curves with a relatively large, uniform and fine-grained tuning space. Finally, regarding **portability**, our design works for all the current generations of GPUs with SFUs equipped, and it does not rely on architecture-related properties except for the limitation of the hardware warp-slots (Table 3).

7. VALIDATION

In this section, we validate our SFU-driven approximate acceleration design in the overall framework. We test 20 SP and 4 DP applications shown in Table 6 on the Fermi, Kepler and Maxwell platforms (*Platform 1,2,3* in Table 3). To be convenient, here we define **QoS_ratio** as the ratio of QoS with respect to the error-rate of the SFU version, which is supposed to be the highest based on the observations in Section 5.3. Note that QoS_ratio is not QoS. For example, if the QoS of the pure SFU version regarding an application is 0.7, which means the error-rate of the SFU version is 1-0.7=0.3; then a QoS_ratio of 0.8 equals to a QoS of 1-0.3*0.8=0.76. We use QoS_ratio because the QoS values for the SFU-versions of different applications are distinct. The QoS_ratio offers a unified assessment criteria for comparison among applications. We also implement the naive (*SMP*), the heuristic (*HEU*) and the exhaustive search (*OMG*) methods described in Section 5.4 for searching efficiency comparison. Figure 8, 9 and 10 illustrate the results for applying our framework to locate the optimal approximate degree of the 20 SP applications on the three GPU platforms with the *QoS_ratio*[7] *=0.8*, respectively. Figure 11 shows the results for the 4 DP applications. In these four figures, **SPU/DPU** is the baseline with no approximation. **SFU** is the maximum attainable speedup via the proposed approach when all the transcendental functions are calculated by the SFUs. The green numbers marked on top of the bars indicate the total search rounds or steps, as described in Section 5.4. Such numbers indicate the numbers of executions during the search, or the searching overhead. We also show the geometric-mean of the performance speedup across the 20 SP and 4 DP applications to provide a general sense of acceleration under our framework. These figures demonstrate that given a specified QoS, *HEU* can achieve close to the best attainable performance with smaller searching iterations, compared to *SMP* and *OMG*.

Figure 12, 13 and 14 illustrate that the normalized power and energy reduction for SP and DP on the Maxwell Jetson-TX1 GPU (*Platform 5* in Table 3) for calculating the transcendental functions in the 20 SP and 4 DP applications via the proposed methods (*SMP, HEU* and *OMG*, which is the most optimal can be achieved at that QoS level) under the QoS_ratio=0.8. As can be seen, although the power reduction does not seem to be tremendous (around 5% for SP and 10% for DP), the energy reduction is quite significant – more than 75% and 25% for SP and DP respectively, which implies that our approximate method can also be quite effective for addressing power/energy constraining problems on GPUs.

8. RELATED WORK

Approximate computing, which broadly refers to technique that harvests substantial performance/energy benefits

[7]We choose QoS=0.8 as an example for demonstration purposes. Users should determine the proper QoS metric and level for their individual application.

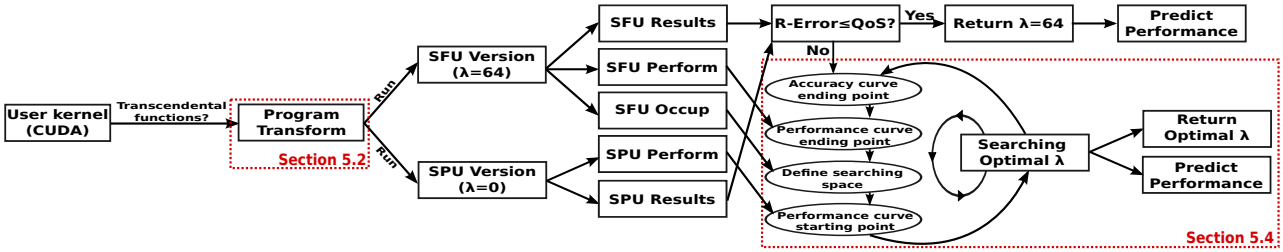

Figure 7: SFU-Driven Transparent Approximate Acceleration Framework.

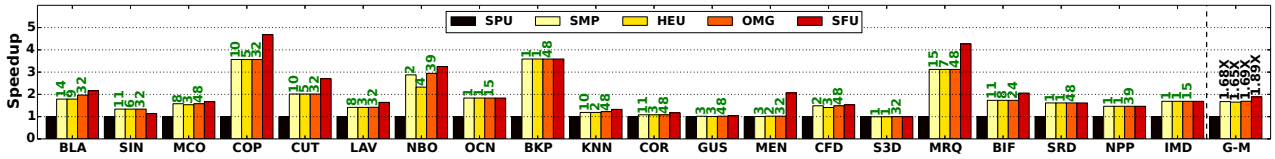

Figure 8: Performance speedup with QoS_ratio=0.8 on Fermi GPU in SP.

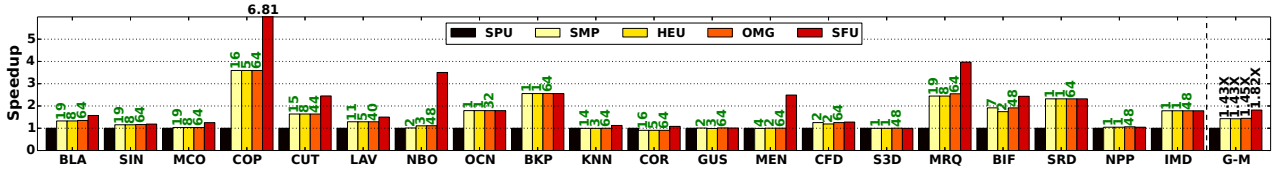

Figure 9: Performance speedup QoS_ratio=0.8 on Kepler GPU in SP.

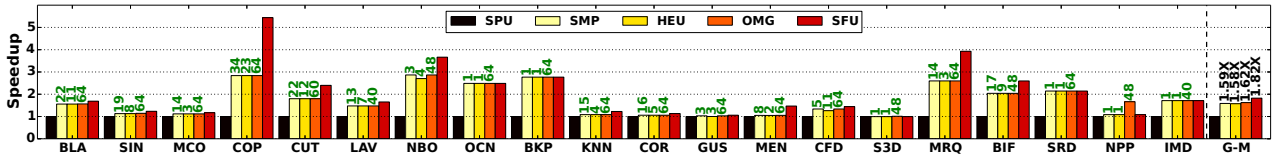

Figure 10: Performance speedup with QoS_ratio=0.8 on Maxwell GPU in SP.

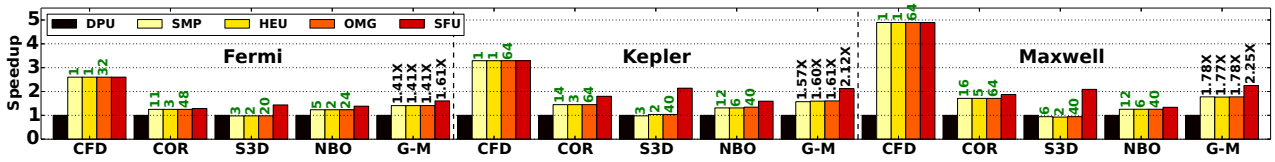

Figure 11: Performance speedup QoS_ratio=0.8 on Fermi, Kepler and Maxwell GPUs in DP.

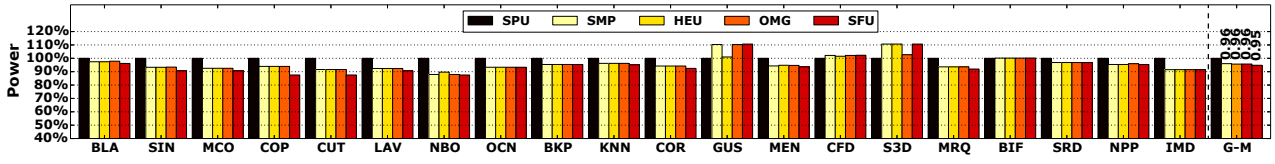

Figure 12: Normalized power reduction with QoS_ratio=0.8 on Maxwell Jetson-TX1 in SP.

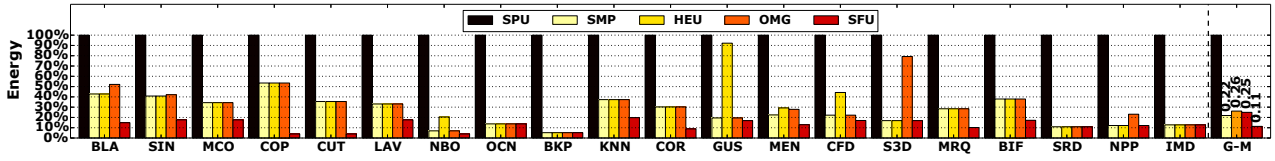

Figure 13: Normalized energy reduction with QoS_ratio=0.8 on Maxwell Jetson-TX1 in SP.

at the expense of modest accuracy loss, has prevailing at all levels of hardware and software designs. On one hand, the emerging big-data, multimedia and machine learning applications are much more insensitive to the computation accuracy. On the other hand, the low level hardware design faces ever-growing concerns on energy, resilience and sustainable scaling of performance. The majority of the existing research has been related to some traditional topics at both hardware

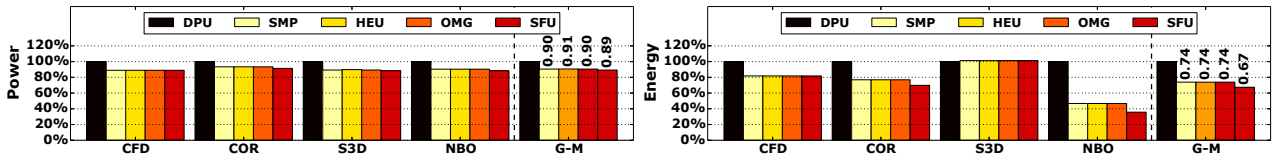

Figure 14: Normalized power and energy reduction with QoS_ratio=0.8 on Maxwell Jetson-TX1 in DP.

level (e.g., fault-allowable storage [36], voltage overscaling [37], DRAM refresh [38], analog circuits [39], neural acceleration [40], descent fault recovery [41], remote memory data prediction [42], function memorization [5, 7], control/memory divergence [6]) and software level (e.g., loop perforation [25], task skipping [43], loop early termination [4, 44], program transformation [23], compilation [24], bitwdith reduction [38]). However, it is often not suitable to deploy the current approximate techniques directly to the scientific applications (e.g., weather simulation and molecular dynamics), which are usually numerically intensive and very sensitive to accuracy loss. This is especially true when future large-scale scientific applications are executed on thousands of heterogeneous HPC nodes (e.g., CPUs+GPUs) and a small inaccurate intermediate result can accumulate or propagate quickly to become significant [10, 11].

Recently, trading the accuracy of the results for better performance has been studied on GPUs [4, 5, 6, 7, 8, 9], as they become the essential computation units in both data centers and HPC systems. Samadi et al. [4] proposed three optimization techniques to automatically generate a series of GPU kernels with different aggressiveness of approximations. They also adopt an iterative sampling-calibration runtime tuning system to select the kernel in the series that is the most aggressive but complying to the specified QoS, provided that the same kernel is invoked repeatedly. Later, they found that for data-parallel applications, six common-used algorithm patterns could be approximated based on their specific properties [5]. Arnau et al. [7] proposed a look-up-table based task-level memorization approach to remove the redundant fragment computation when processing graphical applications in low-power mobile GPUs. Sartori and Kumar [6] applied the approximate concept to address the control and memory divergence on GPUs. They claimed that, for some error-tolerated applications, if the lockstep execution and memory coalescing are strictly enforced by approximating divergent paths to regular/coalesced paths, significant performance can be achieved with limited output quality degradation. Yazdanbakhsh et al. [8] focused on the long memory latency and limited memory bandwidth of GPUs, and predict the requested memory value without actually fetching it from the off-chip memory. Finally, Sutherland et al. [9] predicted the requested memory values using the GPU texture fetch units based on a thread's local history. However, the work above primarily exploits the spatial and/or temporal locality — the similarity among memory elements, computation lanes, historical memory loads, etc. They use hardware (e.g., look-up-table) or software (e.g., program transformation) approaches to approximate some of the requested data or computation with the predicted value based on locality. They often cannot provide accuracy assurance as locality is not always held, and if the crucial elements are approximated significantly inaccurate, catastrophic failures may occur. That is why most of the work above focused on applications that inherently have high tolerance for errors (e.g., machine learning or image applications), e.g., $\geq 10\%$ inaccuracy for approximation. Furthermore, the exact trade-off trends between the performance and accuracy are mostly nonlinear, sometimes even unknown beforehand. This is also why many of them require a profiling phase to test the kernel versions or train the look-up table. In addition, the performance-accuracy tuning space is relatively small and coarse-grained for most of the work above. In contrast, our SFU-centric approximation approach introduces nearly-linear performance-accuracy trade-off curves with a relatively large and fine-grained tuning space, for accuracy-sensitive scientific applications.

9. LIMITATIONS AND FUTURE WORKS

Limited by the situation that only 9 single-precision and 4 double-precision approximate numeric functions are implemented in the SFUs, the propsed design can only accelerate applications that contain these functions. Furthermore, limited by the fixed accuracy of the current SFU design (with error less than 1E-06, see Table 4), we are unable to trade more accuracy with additional performance/energy gains. With regard to the future work, from hardware perspective, we can either design special-function accelerators that are faster but with higher error tolerance, or create accelerators that are more general-purpose such as the neural accelerator for GPUs [45]. From the software perspective, application developers can provide alternative approximate kernel implementations. For instance, in the *leukocyte* application from Rodinia benchmark [32], the *heaviside()* kernel has another "*simpler and faster*" approximate implementation which targets *actanf()*. Using a similar idea proposed in this work, we can co-schedule this user-defined approximate version with the accurate version without hardware involvement. [46] actually offers some software-based approximate functions, such as *sin, cos, exp* and *rcp*. Finally, it is also possible to apply the co-scheduling approach to approximate/accurate memory access of GPUs, such as guessing the data value when it is missed in the cache [8], or approximating a value based on the surrounding elements via interpolation in the texture cache [9].

10. CONCLUSION

In this paper, we focus on a crucial GPU component which however, has long been ignored — the Special Function Units (SFUs), and show its outstanding role in performance acceleration and approximate computing for GPU applications. We exhaustively evaluated the 9 single-precision and 4 double-precision numeric transcendental functions that are accelerated by SFUs in terms of their latency, accuracy, power, energy, throughput, resource cost, etc. Based on these information, we proposed a transparent, tractable and portable design framework for SFU-driven approximate acceleration on GPUs. It leverages the SIMT execution model of GPU to partition the initiated warps into a SPU/DPU-based slower but accurate path, and a SFU-based faster but approximated path, and then tune the relative parti-

tion ratio among the two to control the trade-offs between the performance and accuracy of the kernels. In this way, a fine-grained and almost linear tuning space for the trade-off between performance and accuracy can be created for a scientific application with approximate acceleration. With the linear tuning curve, we propose a simple yet effective heuristic method to search the optimal approximate degree that delivers the best performance subjecting to a user-predefined QoS level. The entire tuning process can be encapsulated as an automatic pure-software approximate-optimization framework, which is demonstrated to be effective for delivering immediate and substantial performance gains over a series of commodity GPU platforms.

11. ACKNOWLEDGMENTS

We thank all the anonymous reviewers for their insightful comments. This research is supported by the U.S. Department of Energy's (DOE) Office of Science, Office of Advanced Scientific Computing Research, under award 66150: "CENATE - Center for Advanced Architecture Evaluation". The Pacific Northwest National Laboratory is operated by Battelle for the U.S. Department of Energy under contract DE-AC05-76RL01830. It is also supported by the German Research Foundation (DFG) within the Cluster of Excellence Center for Advancing Electronics Dresden (cfaed).

12. REFERENCES

[1] Swagath Venkataramani, Srimat T. Chakradhar, Kaushik Roy, and Anand Raghunathan. Approximate Computing and the Quest for Computing Efficiency. In *Proceedings of the 52nd Annual Design Automation Conference*, DAC'15, pages 120:1–120:6. ACM, 2015.

[2] Sasa Misailovic, Stelios Sidiroglou, Henry Hoffmann, and Martin Rinard. Quality of Service Profiling. In *Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering - Volume 1*, ICSE'10, pages 25–34. ACM, 2010.

[3] Hwu Wen-Mei. *GPU Computing Gems Emerald Edition*. Elsevier, 2011.

[4] Mehrzad Samadi, Janghaeng Lee, D Anoushe Jamshidi, Amir Hormati, and Scott Mahlke. Sage: Self-tuning approximation for graphics engines. In *Proceedings of the 46th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pages 13–24. ACM, 2013.

[5] Mehrzad Samadi, Davoud Anoushe Jamshidi, Janghaeng Lee, and Scott Mahlke. Paraprox: Pattern-based Approximation for Data Parallel Applications. In *Proceedings of the 19th International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS'14, pages 35–50. ACM, 2014.

[6] John Sartori and Ravindra Kumar. Branch and data herding: Reducing control and memory divergence for error-tolerant GPU applications. *IEEE Transactions on Multimedia*, 15(2):279–290, 2013.

[7] Jose-Maria Arnau, Joan-Manuel Parcerisa, and Polychronis Xekalakis. Eliminating redundant fragment shader executions on a mobile GPU via hardware memoization. In *Proceedings of the 41st ACM/IEEE International Symposium on Computer Architecture (ISCA)*, pages 529–540. IEEE, 2014.

[8] Amir Yazdanbakhsh, Gennady Pekhimenko, Bradley Thwaites, Hadi Esmaeilzadeh, Taesoo Kim, Onur Mutlu, and Todd C Mowry. RFVP: Rollback-Free Value Prediction with Safe-to-Approximate Loads. In *Proceedings of the 11th International Conference on High Performance and Embedded Architectures and Compilers (HiPEAC)*. ACM, 2016.

[9] Mark Sutherland, Joshua San Miguel, and Natalie Enright Jerger. Texture Cache Approximation on GPUs. 2015.

[10] David Fiala, Frank Mueller, Christian Engelmann, Rolf Riesen, Kurt Ferreira, and Ron Brightwell. Detection and Correction of Silent Data Corruption for Large-scale High-performance Computing. In *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*, SC'12, pages 78:1–78:12. IEEE Computer Society Press, 2012.

[11] Rizwan A. Ashraf, Roberto Gioiosa, Gokcen Kestor, Ronald F. DeMara, Chen-Yong Cher, and Pradip Bose. Understanding the Propagation of Transient Errors in HPC Applications. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, SC'15, pages 72:1–72:12. ACM, 2015.

[12] Erik Lindholm, John Nickolls, Stuart Oberman, and John Montrym. NVIDIA Tesla: A unified graphics and computing architecture. *IEEE Micro*, (2):39–55, 2008.

[13] Wilson W. L. Fung, Ivan Sham, George Yuan, and Tor M. Aamodt. Dynamic Warp Formation and Scheduling for Efficient GPU Control Flow. In *Proceedings of the 40th Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO-40, pages 407–420. IEEE Computer Society, 2007.

[14] NVIDIA. NVIDIA's Next Generation CUDA Compute Architecture: Kepler GK110, 2015.

[15] Vasily Volkov. Better performance at lower occupancy. In *Proceedings of the GPU Technology Conference (GTC)*, volume 10. San Jose, CA, 2010.

[16] Nicholas Wilt. *The CUDA handbook: A comprehensive guide to GPU programming*. Pearson Education, 2013.

[17] David A Patterson and John L Hennessy. *Computer organization and design: the hardware/software interface*. Newnes, 2013.

[18] Stuart F Oberman and Michael Y Siu. A high-performance area-efficient multifunction interpolator. In *Proceedings of the 17th IEEE Symposium on Computer Arithmetic (ARITH)*, pages 272–279. IEEE, 2005.

[19] Davide De Caro, Nicola Petra, and Antonio GM Strollo. High-performance special function unit for programmable 3-D graphics processors. *IEEE Transactions on Circuits and Systems I: Regular Papers (TCAS-I)*, 56(9):1968–1978, 2009.

[20] NVIDIA. CUDA Math API, 2015.

[21] Jingwen Leng, Tayler Hetherington, Ahmed ElTantawy, Syed Gilani, Nam Sung Kim, Tor M. Aamodt, and Vijay Janapa Reddi. GPUWattch: Enabling Energy Optimizations in GPGPUs. In *Proceedings of the 40th Annual International*

Symposium on Computer Architecture (ISCA), pages 487–498. ACM, 2013.

[22] NVIDIA. NVIDIA system management interface, 2015.

[23] Zeyuan Allen Zhu, Sasa Misailovic, Jonathan A Kelner, and Martin Rinard. Randomized accuracy-aware program transformations for efficient approximate computations. In *ACM SIGPLAN Notices*, volume 47, pages 441–454. ACM, 2012.

[24] Pooja Roy, Jianxing Wang, and Weng Fai Wong. PAC: program analysis for approximation-aware compilation. In *Proceedings of the 2015 International Conference on Compilers, Architecture and Synthesis for Embedded Systems*, pages 69–78. IEEE Press, 2015.

[25] Stelios Sidiroglou-Douskos, Sasa Misailovic, Henry Hoffmann, and Martin Rinard. Managing performance vs. accuracy trade-offs with loop perforation. In *Proceedings of the 19th ACM SIGSOFT symposium and the 13th European conference on Foundations of software engineering*, pages 124–134. ACM, 2011.

[26] Ang Li, Gert-Jan van den Braak, Akash Kumar, and Henk Corporaal. Adaptive and Transparent Cache Bypassing for GPUs. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, SC'15, pages 17:1–17:12. ACM, 2015.

[27] NVIDIA. Inline PTX Assembly in CUDA, 2015.

[28] NVIDIA. CUDA SDK Code Samples, 2015.

[29] Ang Li, Gert-Jan van den Braak, Henk Corporaal, and Akash Kumar. Fine-Grained Synchronizations and Dataflow Programming on GPUs. In *Proceedings of the 29th ACM on International Conference on Supercomputing (ICS)*, pages 109–118. ACM, 2015.

[30] Ali Bakhoda, George L Yuan, Wilson WL Fung, Henry Wong, and Tor M Aamodt. Analyzing CUDA workloads using a detailed GPU simulator. In *Proceedings of the IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*, pages 163–174. IEEE, 2009.

[31] John A Stratton, Christopher Rodrigues, I-Jui Sung, Nady Obeid, Li-Wen Chang, Nasser Anssari, Geng Daniel Liu, and W-M Hwu. Parboil: A revised benchmark suite for scientific and commercial throughput computing. *Center for Reliable and High-Performance Computing*, 2012.

[32] Shuai Che, Michael Boyer, Jiayuan Meng, David Tarjan, Jeremy W Sheaffer, Sang-Ha Lee, and Kevin Skadron. Rodinia: A benchmark suite for heterogeneous computing. In *Proceedings of the IEEE International Symposium on Workload Characterization (IISWC)*. IEEE, 2009.

[33] Scott Grauer-Gray, Lifan Xu, Robert Searles, Sudhee Ayalasomayajula, and John Cavazos. Auto-tuning a high-level language targeted to GPU codes. In *Innovative Parallel Computing (InPar)*. IEEE, 2012.

[34] Anthony Danalis, Gabriel Marin, Collin McCurdy, Jeremy S Meredith, Philip C Roth, Kyle Spafford, Vinod Tipparaju, and Jeffrey S Vetter. The scalable heterogeneous computing (SHOC) benchmark suite. In *Proceedings of the 3rd Workshop on General-Purpose Computation on Graphics Processing Units (GPGPU)*, pages 63–74. ACM, 2010.

[35] Ismail Akturk, Karen Khatamifard, and Ulya R Karpuzcu. On quantification of accuracy loss in approximate computing. In *Workshop on Duplicating, Deconstructing and Debunking (WDDD)*, page 15, 2015.

[36] Adrian Sampson, Jacob Nelson, Karin Strauss, and Luis Ceze. Approximate storage in solid-state memories. *ACM Transactions on Computer Systems (TOCS)*, 32(3):9, 2014.

[37] Hyungmin Cho, Larkhoon Leem, and Subhasish Mitra. ERSA: Error resilient system architecture for probabilistic applications. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 31(4):546–558, 2012.

[38] Hadi Esmaeilzadeh, Adrian Sampson, Luis Ceze, and Doug Burger. Architecture Support for Disciplined Approximate Programming. In *Proceedings of the Seventeenth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS XVII, pages 301–312. ACM, 2012.

[39] Renée St. Amant, Amir Yazdanbakhsh, Jongse Park, Bradley Thwaites, Hadi Esmaeilzadeh, Arjang Hassibi, Luis Ceze, and Doug Burger. General-purpose Code Acceleration with Limited-precision Analog Computation. In *Proceeding of the 41st Annual International Symposium on Computer Architecuture*, ISCA'14, pages 505–516. IEEE Press, 2014.

[40] Hadi Esmaeilzadeh, Adrian Sampson, Luis Ceze, and Doug Burger. Neural Acceleration for General-Purpose Approximate Programs. In *Proceedings of the 2012 45th Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO-45, pages 449–460. IEEE Computer Society, 2012.

[41] Marc de Kruijf, Shuou Nomura, and Karthikeyan Sankaralingam. Relax: An Architectural Framework for Software Recovery of Hardware Faults. In *Proceedings of the 37th Annual International Symposium on Computer Architecture*, ISCA'10, pages 497–508. ACM, 2010.

[42] Joshua San Miguel, Mario Badr, and Natalie Enright Jerger. Load Value Approximation. In *Proceedings of the 47th Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO-47, pages 127–139. IEEE Computer Society, 2014.

[43] Martin Rinard. Probabilistic Accuracy Bounds for Fault-tolerant Computations That Discard Tasks. In *Proceedings of the 20th Annual International Conference on Supercomputing*, ICS'06, pages 324–334. ACM, 2006.

[44] Woongki Baek and Trishul M Chilimbi. Green: a framework for supporting energy-conscious programming using controlled approximation. In *ACM Sigplan Notices*, volume 45, pages 198–209. ACM, 2010.

[45] Amir Yazdanbakhsh, Jongse Park, Hardik Sharma, Pejman Lotfi-Kerman, and Hadi Esmaeilzadeh. Neural Acceleration for GPU Throughput Processors. 2015.

[46] Nicolas Brisebarre, Jean-Michel Muller, and Arnaud Tisserand. Sparse-coefficient polynomial approximations for hardware implementations. In *Signals, Systems and Computers, 2004. Conference Record of the Thirty-Eighth Asilomar Conference on*, volume 1, pages 532–535. IEEE, 2004.