ST² GPU: An Energy-Efficient GPU Design with Spatio-Temporal Shared-Thread Speculative Adders

Vijay Kandiah
ECE, Northwestern University
vijayk@u.northwestern.edu

Ali Murat Gök
Argonne National Labs
agok@anl.gov

Georgios Tziantzioulis
EE, Princeton University
gt5@princeton.edu

Nikos Hardavellas
CS & ECE, Northwestern University
nikos@northwestern.edu

***Abstract*—Modern GPUs employ thousands of cores, yielding higher performance but also higher power consumption. To meet performance targets while staying within a reasonable power budget, designers have to make these execution cores increasingly more power efficient. One way to increase their power efficiency is to employ power-efficient adders. In this paper, we observe that consecutive arithmetic computations from the same code location are highly correlated and propose *ST² GPU*, a GPU architecture that uses history-based speculative adders that produce guaranteed correct results while saving 70% of the nominal adder power. We estimate that *ST² GPU* saves 21% of the GPU chip energy with practically no performance and area overheads.**

I. Introduction

Graphics Processing Units (GPUs) are becoming increasingly popular for accelerating both general-purpose and high-performance computing applications. Currently, there are 147 GPU-accelerated systems in the TOP500 HPC list [1] and 70% of the top-50 HPC applications are GPU-accelerated [2]. As the appeal of GPUs grows, so does the demand for higher performance. To meet the ever-increasing performance targets, designers cram increasingly more cores per GPU chip, leading to a commensurate rise in power consumption. However, the power budget of modern GPUs is already reaching the limits of practical cooling technology. For example, both NVIDIA's Volta GV100 architecture [3] and the previous-generation Pascal GP100 are limited by the same 250 W thermal design power, even though GV100 contains 43% more CUDA cores. In order to continue increasing the core count at a constant power budget, the cores must become more energy efficient.

Owing to their sheer number on a chip, add/subtract execution units such as integer arithmetic and logic units (ALUs) and floating-point units (FPUs) are collectively among the most power-hungry hardware components. Often, ALUs and FPUs are exercised intensely by workloads: 21 out of 23 kernels from Rodinia [4], NVIDIA CUDA Samples [5], and Parboil [6] running on an NVIDIA TITAN V Volta exhibit high arithmetic intensity, i.e., more than 20% of the executed dynamic instructions are ALU and FPU instructions (Figure 1). In this paper, we directly target a reduction in ALU and FPU power consumption by introducing a new power-efficient adder design, and an associated GPU architecture.

Our work is inspired by the observation that real-world GPU applications exhibit an important but overlooked behavior: the computed values of consecutive operations from the same line of code are often highly correlated. (i.e., the values computed by the same instruction as it repeatedly executes, tend to be of similar magnitude). We capitalize on this observation and propose Spatio-Temporal Shared-Thread (*ST²*) adders, a power-efficient speculative adder design that utilizes the spatio-temporal history of arithmetic operations in a GPU kernel to perform additions. While the adder executes speculatively, mispredictions are immediately detected upon the nominal end of the adder's execution and corrected in subsequent cycles. Thus, ST² adders guarantee correctness. At the same time, they save 70% of the adder power and achieve 27% higher prediction accuracy over the current state-of-the-art VaLHALLA [7] design.

This work was partially funded by NSF awards CCF-1453853, CNS-1763743.

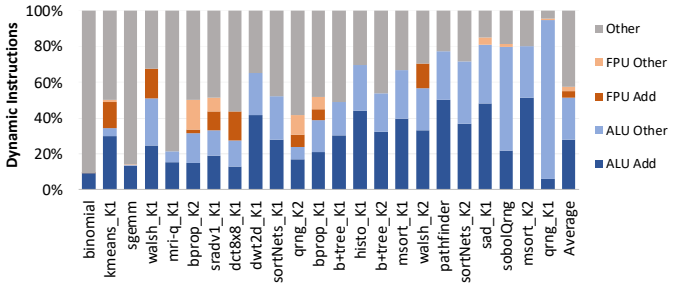

Fig. 1. **ALU and FPU operations are prevalent in GPU kernels.**

We incorporate ST² adders into a new GPU architecture, ST² GPU. ST² GPU modifies the pipeline of an NVIDIA Volta GV100 to accommodate the variable-delay adders, and facilitates access to history tables by piggy-backing on the GPU's operand collector. The ST² GPU design achieves a 21% chip energy reduction across 23 kernels from Rodinia [4], NVIDIA CUDA Samples [5], and Parboil [6], with practically no performance and area overheads.

This work makes the following contributions:

- We observe, explain and quantify spatio-temporal value correlation on real-world GPU applications.
- We propose ST² adders, a speculative adder design that exploits spatio-temporal value correlation to perform carry speculation and reduce power consumption. ST² adders guarantee correctness and outperform state-of-the-art designs.
- We perform a design-space exploration of carry speculation units on GPUs along the spatial axis (PC correlation), temporal axis (history depth), and history sharing among threads, and arrive at a practical, high-performance carry speculation unit for GPUs.
- We propose ST² GPU, an architecture that integrates ST² adders and carry speculation units into the warp pipeline, and show it achieves significant power savings with negligible overheads.

II. Background

A. Volta Architecture and Execution Model

Our GPU architecture model is inspired by the NVIDIA Volta GV100 GPU architecture [3], and particularly the TITAN V Volta. The TITAN V Volta has 80 Streaming Multiprocessors (SMs) each with 64 32-bit integer units (ALUs), 64 32-bit floating-point units (FPUs), 32 64-bit double-precision units (DPUs), 4 special function units (SFUs) for complex operations (e.g., log, square root), and 8 tensor cores for matrix arithmetic. Our design targets the adders within the ALUs, FPUs and DPUs.

GPUs execute programs known as "kernels", which typically comprise thousands of threads. Upon launching a kernel, each thread gets its own GPU-wide unique *global* thread ID. Threads do not execute instructions independently; rather, sets of 32 threads (warps)

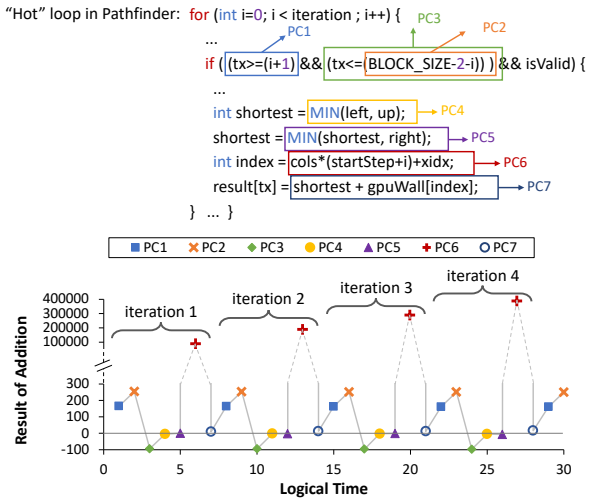

Fig. 2. **Value evolution of addition results from the Pathfinder kernel.**

execute the same instruction on different data. Each thread in a warp is identified by its *local* thread ID, i.e., a number between 0–31. In the rest of the paper, we refer to these *global* and *local* thread IDs.

GPU kernels are offloaded to the the GPU device through the use of CUDA, a parallel computing platform and application programming interface model. The CUDA programming environment provides a parallel thread execution (PTX) [8] instruction set architecture (ISA), which is an intermediate ISA that exposes the GPU as a data-parallel computing device. PTX programs are translated at install time to the target hardware ISA that executes natively on the GPU.

B. Speculative Adders

To reduce power consumption, speculative adders divide a regular adder's full bit range into smaller bit ranges ("slices") and run them in parallel. As slices are smaller, they can execute at a fraction of the nominal clock period. Speculative adders exploit the unused clock period to scale down each slice's supply voltage to the lowest setting that allows the slice to still fit within the same cycle time [9], gaining quadratic power savings. However, running the slices in parallel breaks the carry-propagation chain. Speculative adders overcome this obstacle by speculating on the carry-in of each slice.

Approximate speculative adders [10]–[13] do not possess error correction mechanisms and wrong results are supplied whenever a carry-in is mispredicted. In contrast, variable latency speculative adders [7], [14] detect mispredictions at the end of the nominal execution and occupy additional execution cycles to recompute with the corrected carry-in if an error occurred. Thus, they always provide the correct result, but incur an overhead whenever a misprediction occurs. Our design is inspired by VaLHALLA [7], a recently-proposed variable-latency adder that is shown to outperform prior speculative adder designs. VaLHALLA provides a static prediction for all slices' carry-ins based on the correlation between the length of the carry propagation chain and the input operands.

III. Spatio-Temporal Value Correlation in GPUs

A characteristic of applications is that code execution repeats, both within threads of computation (e.g., in loops) and across threads (e.g., the same kernel running on separate threads). Thus, the same instructions, at the same PC, repeat, one iteration after another and one kernel thread after another. As these "hot" instructions operate in succession they transform data. While the data values produced by different instructions often bear limited correlation with each other, instructions at the same PC often operate on arguments of

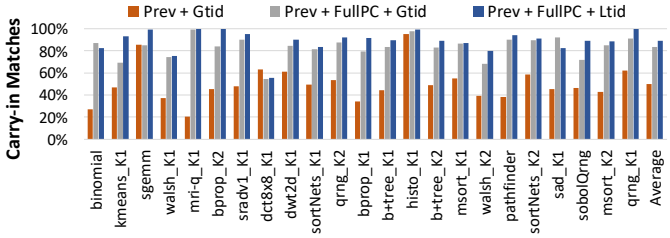

Fig. 3. **8-bit slice carry-in correlation across the temporal & spatial axes.**

similar magnitude and produce values similar to the ones the same instruction produced in the previous invocation. For example, the same instruction that increments the iterator of a loop will keep executing, repeatedly producing a sequence of nearby values (e.g., 1, 2, 3). Another instruction in the body of the loop may operate on other data and produce new values. As that instruction repeats, it produces values that tend to be within similar magnitudes across a short window of time, gradually evolving rather than wildly fluctuating across the integer or floating-point range. In short, code repetition gives rise to value correlation.

Figure 2 shows a real-world example of value correlation. The code snippet on Figure 2 is the hot loop of the kernel in *pathfinder* from the Rodinia benchmark suite (Section V-A). We highlight the loop's addition operations and mark them with their logical PC, ranging from PC1 to PC7. As these additions execute and operate on the application's data, they produce new values. Figure 2 (bottom) shows the evolution of these values in logical time (i.e., in the order of instruction execution). Control and data dependencies force the instructions at PC1, PC2, PC3, PC4, PC5 and PC7 to execute in this exact order, while PC6 is ordered between PC3 and PC7.

As Figure 2 shows, when observed as a whole the values generated by these additions as they execute in order vary greatly. The exist values in the 100s (PC1, PC2), around 0 (PC4, PC5, PC7), and even tens of thousands (PC6) or negative (PC3). While there is some correlation in the magnitude of the results from different instructions that execute consecutively (e.g., both PC4 and PC5 produce values close to zero), this correlation is weak and is often broken by instructions producing wildly different results. However, the values produced by the *same* instruction (i.e., at the same PC) across iterations are of *similar magnitude* and strongly correlated.

This value correlation translates to correlation in the carry chains. Operations on small positive numbers yield short carry chains, (e.g., PC1, PC2, and PC7 which produce carry chains that do not propagate beyond the first 8 bits), while instructions producing larger values produce longer carry chains (e.g., PC6's results may produce a carry that propagates through the first 16 bits). Additions producing negative results (e.g., PC3) may produce carry chains that propagate all the way to bit 63. We observe that while the carry chain length is weakly correlated across different instructions, it is strongly correlated across subsequent executions (temporal correlation) of the same instruction (spatial correlation).

We quantify this spatio-temporal value correlation in our workload suite in Figure 3. We envision additions performed not in a monolithic 64-bit adder, but rather by stringing together 8-bit adder slices, each fed with the carry out of the previous 8-bit slice. We compare the carry-outs/carry-ins between adder slices as instructions execute. When we compare the carry-ins between consecutive additions executed within each thread (same global threadID), regardless of the PC, only 50% match on average (Prev+Gtid). Thus, there is practically no correlation along the temporal axis alone. However, when we compare separately the slice carry-ins from consecutive executions of

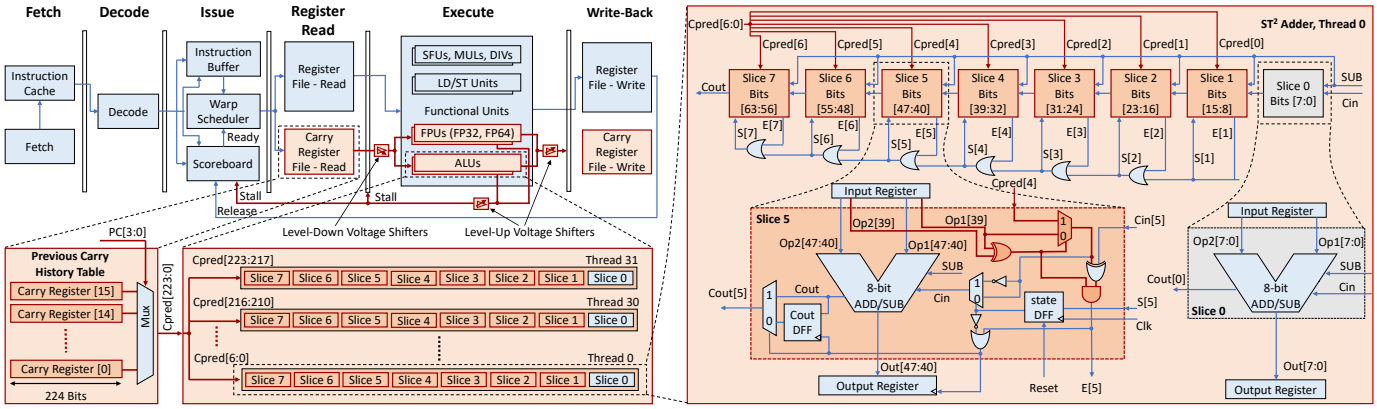

Fig. 4. **Adder slice design. Slices 1-7 are similar. Changes over VaLHALLA are highlighted in red.**

the *same* PC within each thread, we find matches in 83% of the cases on average (Prev+FullPC+Gtid). Thus, while temporal correlation alone is limited, the spatio-temporal correlation is strong. In addition, as all threads in a GPU kernel's block execute the same program, they can learn from each other. When we compare not against the previous execution of an instruction at the same PC by the same thread (same global threadID), but across all threads in the same warp lane (between 0 and 31), then matches are found in 89% of the cases (Prev+FullPC+Ltid), showing that sharing history among threads can enhance the speed of finding correlations. In the following section we capitalize on these observations to design the ST^2 adders.

IV. ST^2 Design and Space Exploration

A. ST^2 Adder Slice Design

The ST^2 adder is inspired by VaLHALLA [7] and significantly improves upon it. Figure 4 depicts the design of the ST^2 adder, with slices 0 and 5 shown in detail. While The NVIDIA TITAN V Volt GPU has only 32-bit adders, here we show the design of ST^2 for the general case of a 64-bit adder. For simplicity of explanation, let's assume that the nominal latency of an ADD operation is 1 cycle.

At the beginning of an ADD operation, ST^2 makes a prediction of the carry-in for each slice and performs the ADD computation. The prediction is communicated to the adder through signals $Cpred[0] - Cpred[6]$ from the Carry Register File to each adder slice in Figure 4 (Section IV-B explains how ST^2 makes these predictions). At the end of the nominal execution cycle, each slice compares the prediction it received ($Cpred[4]$ for slice 5) with the carry-out generated by the previous slice ($Cin[5]$ for slice 5). If they do not match then a misprediction has occurred. In that case the slice consumes an additional cycle to re-compute the ADD operation with the inverse carry-in of the previous cycle ($\neg Cpred[4]$ for slice 5). Thus, the execution of an ADD may take one or two cycles, depending on whether there was a misprediction.

If slice i mispredicts, then all carry-outs generated by slices $i+1, ..., 7$ are suspect of being incorrect. Thus, upon a misprediction, an error signal is generated ($E[5]$ for slice 5) that propagates to all higher-order slices (signals $S[i]$) and informs them that they may have received an erroneous carry from their previous slices. Each of the affected slices will then proceed with a second cycle of computation, using the inverse carry than the one assumed in the previous cycle. A 1-bit State DFF register keeps track of whether the slice is performing the first or the second cycle of computation. At the beginning of an ADD operation, all State DFFs are reset to 0. At the end of the first cycle, each slice's State DFF is updated by OR-ing the error signals of the current and all previous slices, and then stays at that value until a new operation is assigned to the adder. Thus, the State DFF remembers whether the predicted carry is to be trusted or not. At the end of this second cycle all correct carry-ins are known, and each ST^2 slice decides to either keep the results already in its output register (i.e., the first cycle's computation was the correct one) or overwrite them with the result of the second cycle. This operation is similar to a Carry Select Adder (CSLA) [15]. Unlike CSLA, though, which always performs computations with both carry-ins for all slices, ST^2 performs additional slice computations only when a misprediction occurs, and only on the subset of slices that cannot trust their prediction. Thus, ST^2 avoids unnecessary computations and exhibits significant power savings over CSLA.

B. ST^2 Carry Speculation Mechanism and Comparison to VaLHALLA

ST^2 improves upon VaLHALLA by offering improved carry speculation. Specifically, it employs speculation only when necessary, provides per-thread history-based predictions, promotes thread-history sharing, and is adapted to GPU pipelines. We arrive at the ST^2 architecture by performing a design space exploration, shown in Figure 5. As the figure shows, static carry prediction (e.g., always predict 0—*staticZero*) suffers from high error rates (*staticOne* is even worse). VaLHALLA reduces the misprediction rate through dynamic speculation. However, dynamic speculation is not always necessary. If the most significant bits (MSbs) of the two input operands of the previous slice (Op1[39] and Op2[39] for slice 5) are both zeros, then the carry-in will surely be zero; if they are both ones, then the carry-in will surely be one. Such static predictions are guaranteed to be correct. ST^2 capitalizes on this observation by having each slice *peek* at the MSbs of the previous slice to make a static prediction, and relies on dynamic speculation (and risks errors) only when static predictions are not possible. VaLHALLA always performs dynamic speculation even in these cases. Retrofitting VaLHALLA with *Peek* further reduces its misprediction rate by 18%.

The second limitation of VaLHALLA is that it predicts a single 1-bit carry for the entire adder, which is broadcasted to all slices. Providing the same prediction to all slices increases the misprediction rate and causes additional execution cycles. Instead, ST^2 makes separate carry-in predictions for each slice by observing that consecutive arithmetic operations are correlated (Section III). Correlation increases the likelihood that carry chains have similar lengths among operations executed close in time. Thus, ST^2 remembers, in a *previous carry* history table, the carry-outs produced by each slice for an ADD at time i, and uses them as per-slice predictions for the carry-ins at time $i+1$. The corresponding *Prev+Peek* design puts everything together and achieves a 26% reduction in miss rate over VaLHALLA.

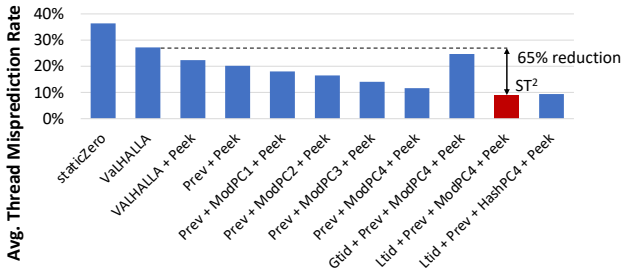

Fig. 5. **Design space exploration for ST² carry speculation mechanism.**

The achieved 20% misprediction rate may still be relatively high, though, as each misprediction means the ALU will consume additional cycles, raising the probability of structural hazards. *Prev+Peek* fails to achieve very low misprediction rates because it allows all instructions to alias with each other. While consecutive executions of the same instruction may be highly correlated and produce similar output values (Section III), executions of different instructions are less likely to correlate. To disambiguate predictions, ST² employs a number of PC bits as part of the index into the previous carry history table by using the lowest k bits of the PC as index to the previous carry history table (*ModPCk*). Figure 5 shows that as k increases, the misprediction rate falls. At 4 PC bits the misprediction rate is just 12%, a full 57% lower than VaLHALLA. Increasing k further provides diminishing returns.

We speculate that the main limitation of the latest design we arrived at is thread aliasing. While the history from different instructions is disambiguated, all threads in *Prev+ModPC4+Peek* share the same history, which may cause undesirable destructive interference. Moreover, there may be significant thread contention, as all threads that execute the same (or aliasing) instructions may simultaneously attempt to update the same entry in the carry history table. To address these problems, we add to *Prev+ModPC4+Peek* the ability to disambiguate threads by adding the global thread ID as part of the carry history table index. The resulting *Gtid+Prev+ModPC4+Peek* design completely disambiguates threads. As we see in Figure 5, however, this design fares significantly worse than most other designs. Thus, sharing history among threads may be beneficial, indicating that interference may be constructive as well. By having threads operate on similar data, they can act as "prefetchers" of the correct carry-ins into the history table. Our data indirectly support the hypothesis that constructive interference among threads may be more prevalent.

Armed with this new intuition, we modify the design to incorporate the local thread ID instead of the global ID in the carry speculation table index, thereby allowing threads to share predictions across warps. The resulting design, *Ltid+Prev+ModPC4+Peek*, achieves a small 9% misprediction error (Figure 5), which is 65% lower than VaLHALLA's. Volta GV100 supports up to 2048 threads/SM [3], thus *Gtid+Prev+ModPC4+Peek* may require a 15-bit history table index (11 global thread ID bits + 4 PC bits) and a commensurately large history table. In contrast, *Ltid+Prev+ModPC4+Peek* exhibits minimal contention that can be practically addressed with random arbitration, as only the few warps executing the register write-back pipeline stage at the exact same cycle in the same SM's computational cluster may conflict with each other, and only when threads within these warps mispredict. The speculative adder design we pick for ST² is *Ltid+Prev+ModPC4+Peek*. More complex PC-based indexing (e.g., XOR-hash of 4-bit PC chunks) provide no additional benefits.

The final ST² design is in stark contrast to VaLHALLA. VaLHALLA predicts the same carry-in for all slices, while ST² predicts an independent carry-in per slice based on instruction history (the *Prev* mechanism). VaLHALLA performs a prediction on every ADD, while ST² predicts only when necessary (the *Peek* mechanism). VaLHALLA predictions are performed for each adder, while ST² allows history sharing across threads. These improvements result in significantly higher prediction accuracy for ST² over VaLHALLA.

It is important to note that configurations to the left of *Ltid+Prev+ModPC4+Peek* in Figure 5 would be unimplementable due to numerous hardware threads requiring simultaneous read/write accesses to the same carry history table entry. This design space exploration shows that our ST² adder design which uses *Ltid+Prev+ModPC4+Peek* exhibits lower misprediction rates than even these optimistic approaches (including VaLHALLA) shown in Figure 5 which ignore contention in accesses to the same history table entry by multiple hardware threads in an SM.

C. ST² GPU Microarchitecture

Figure 4 shows a model of a modern GPU warp pipeline with the proposed modifications to support ST². As ST² operates at lower-than-nominal voltage, level shifters are required when crossing voltage domains. A Carry Register File (CRF), placed next to the regular register file, holds the per-slice carry-outs produced by previous add operations in a history table. The CRF is read along with operands from the register file in the register read pipeline stage. The speculated carry-ins from CRF are sent to the warp's Functional Units (FUs)—i.e., adder or FMA units in ALUs, FPUs or DPUs, depending on the operation—during the execute stage together with the operands, and are utilized by ST² adders to perform the computation. The CRF is structured as a 16×224-bit register file. A CRF read uses PC[3:0] as an index to retrieve 224 bits. These correspond to 7 carry bit predictions (one for each of $slice_1, ..., slice_7$) of each of the warp's 32 threads. Upon completion of the operation, threads with mispredictions update their corresponding bits in the CRF with the new carry-outs, to be used as predictions in subsequent operations. The CRF is updated at the write-back pipeline stage along with the register write-back, similarly to a register file update.

When an FU detects a misprediction, the operation is repeated with the inverse carry-ins to recover from the error (Section IV-B). Upon a misprediction, the FU generates a stall signal that propagates to the scoreboard to prevent the issue of another instruction on the still-occupied functional unit, and stalls the pipeline register to prevent instructions already scheduled from getting to the execute stage.

ST² GPU employs ST² adders not only in integer ALUs, but in FPUs and DPUs as well when performing mantissa operations. Mantissas are 23 or 52 bits for FP32 and FP64, so these units use 3 or 7 slices, respectively. We refrain from employing speculative adders for exponent operations (exponents are only 8-11 bits wide and speculation does not provide any benefit) or in other complex units such as multipliers.

V. ST² GPU Evaluation Methodology

We use GPGPU-Sim 3.x [16] in PTX simulation mode, which is calibrated against an NVIDIA TITAN V Volta and shown to have high correlation with hardware performance measurements [17]. We use this version as the baseline and modify it to incorporate the ST² GPU architecture. Our simulator models ALUs, FPUs and DPUs as seperate components that perform adds, subtracts, and a host of other simpler operations. Multipliers are modeled as separate units.

A. Workloads

Our evaluation suite consists of 23 kernels selected from 18 workloads from NVIDIA CUDA Samples [5] (cudaTensorCoreGemm,

BinomialOptions, fastWalshTransform, dct8x8, sortingNetworks, quasirandomGenerator, histogram, mergesort, and SobolQRNG), Rodinia [4] (kmeans, backprop, sradv1, dwt2d, b+tree, and pathfinder), and Parboil [6] (sgemm, mri-q, and sad). All workloads were compiled with NVCC V9.1.85 with support for Volta using the *arch=sm_70* compiler option. We excluded workloads that could not compile for GPGPU-Sim (e.g., due to unimplemented instructions like *warp.sync*) or were impractical to simulate ($>$ 2 days per run). Additionally, we excluded a few short-running kernels for which we were unable to collect reliable hardware power measurements from our power modeling workflow (it probes the hardware at 50–100 Hz, and as a consequence we could not validate our power model for these workloads against hardware measurements to ensure its accuracy). We use the largest available input configuration for all workload runs.

B. Circuit Design

We model all adder designs in Verilog. We synthesize all designs with the same optimization parameters using Synopsys Design Compiler (H-2013.03-SP5-4) and Synopsys IC Compiler (vI-2013.12-SP5), using the Synopsys SAED 90 nm library. We simulate the netlists using Synopsys VCS-MX (I-2014.03-2) in analog mode and Synopsys HSpice (K-2015.06-1) to analyze their energy and delay characteristics. The reference adder is the default adder synthesized by Synopsys Design Compiler. It is a state-of-the-art, industrial-strength design directly imported from the Synopsys DesignWare Library [18], and synthesized using the recommended default optimization settings to obtain an overall balanced design. We determine the minimum execution delay of the reference adder when nominal voltage is supplied, and use it to define the nominal clock period. Then, we identify the voltage at which we can scale the slices while still fitting within the nominal clock period. From this characterization we extract the reference adder and slice power consumption we use in our modeling. While the circuit modeling is performed with a 90 nm cell library, we estimate that the relative energy differences across adder designs will persist when we scale the designs to the 12 nm FinFET process that NVIDIA Titan V Volta uses.

We perform a design space exploration to identify the optimal ST^2 slice bitwidth. We synthesize sub-adders of different bitwidths, feed them with random vectors as inputs, and evaluate their power consumption on the same random input sequence. We identify 8-bit slices as the best design option for ST^2, as they allow the supply voltage to scale to 60% of the reference voltage, leading to 75–87% potential energy savings per adder. We model the energy and area footprint of ST^2's carry speculation unit independently.

C. Power Modeling

To evaluate power consumption, we develop a power model and extensively validate it before collecting power results for ST^2 GPU over the baseline. We use an internal version of GPUWattch [19] that was calibrated using a set of micro-benchmarks and an NVIDIA TITAN V Volta GPU. Following the methodology of GPUWattch, we develop a suite of 123 micro-benchmarks that isolate and stress specific GPU hardware components. We run these kernels on silicon to collect hardware power measurements using the NVIDIA Management Library at 50–100 Hz. We then use a least-square-error solver to calibrate the GPUWattch power scaling factors per component.

At a high level, our power model is represented by:

$$P_{\text{total}} = P_{\text{const}} + (N_{\text{idleSM}} \times P_{\text{idleSM}}) + \sum_{i=1}^{N} (P_{\text{i}} \times Scale_{\text{i}}) \quad (1)$$

The constant power P_{const} includes power from components such as GPU board fans, power regulators, peripheral circuitry, and leakage.

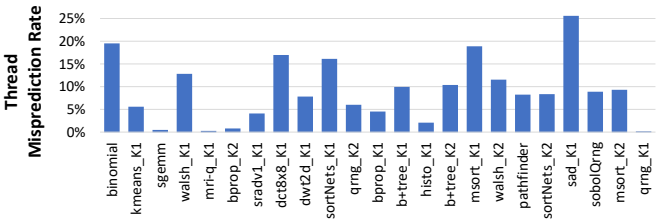

Fig. 6. **Thread misprediction rate for ST^2 adders.**

P_{idleSM} models static power per idle SM, and is multiplied by the number of idle SMs, N_{idleSM}. We model the dynamic power of each component i by multiplying the component's scaling factor $Scale_i$ estimated by the solver, with the component power we obtain from our GPUWattch simulations, P_i. To estimate the total modeled system power, we sum the dynamic power of each component with the chip's constant power and the total idle SM power. P_{const} and P_{idleSM} are also estimated by our solver across all microbenchmarks.

The NVIDIA Volta GV100 does not have divider units. Instead, divisions are performed in hardware as an algorithm that uses other instructions (e.g., FMAs, shifts, etc). However, the PTX ISA includes division instructions, which subsequently are modeled by GPGPU-Sim. Thus, we also separately model the power consumption of division operations; this power does not correspond to a single hardware component, but rather to the collective instructions that execute on hardware to calculate the result of a division operation.

We evaluate the accuracy of our power model by applying it on our benchmark suite of 23 kernels. The model was trained on the microbenchmark stressors only, thus our benchmark suite constitutes a proper validation set. We find that our power model attains an average absolute relative error of $10.5\% \pm 3.8\%$ (95% confidence interval), giving it a strong Pearson r coefficient of 0.8.

VI. Evaluation

We evaluate our design by running the kernels in our benchmark suite on a simulated ST^2 GPU architecture using our in-house versions of GPGPU-Sim and GPUWattch that incorporate our adder and warp pipeline designs and the power model. ST^2 GPU attains a 9% average thread misprediction rate across all kernels (Figure 6). We calculate that on average across all kernels, a single thread misprediction causes 1.94 slices to re-compute their results (up to 2.73). Thus, mispredictions do not cause an excessive energy penalty.

Figure 7 shows the energy breakdown in our suite when running on a simulated TITAN V Volta baseline and on ST^2 GPU. On average the baseline spends 27% of the total system energy on ALUs/FPUs, which corresponds to 30% of the chip energy (excluding DRAM). Kernels such as *qrng_K1* spend as much as 57% of the system energy on ALUs/FPUs. We observe that several of our kernels have high arithmetic intensity, with 14 out of 23 workloads each spending more than 20% of the system energy on ALU/FPU units. These workloads spend on average 31% of the total system energy on ALUs/FPUs, corresponding to an average of 40% of the total chip energy. Thus, minimizing the ALU/FPU energy consumption holds significant promise in increasing the energy efficiency of GPUs.

Figure 7 shows that across our kernel suite, ST^2 saves on average 19% of the system energy, which corresponds to 21% average chip energy savings (excluding DRAM). The energy savings for kernels with high arithmetic intensity are even higher: ST^2 GPU saves on average 26% of the system energy (up to 40% for *msort_K2*), which corresponds to 28% chip energy savings (up to 42%).

These energy savings come with practically no performance overhead. A mispredicted carry-in for even one adder slice in a single

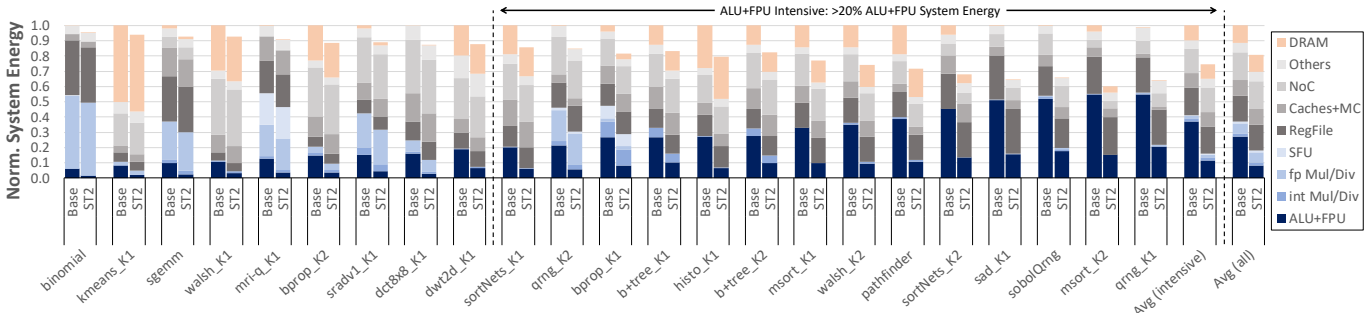

Fig. 7. **Normalized system energy for the baseline and ST^2 GPU architectures.**

thread in a warp would stall the entire warp until the correct output is obtained. While the penalty may seem high, the ST^2 speculation mechanism exhibits very low misprediction rates (Section IV-B), and GPUs are tolerant of such additional latencies. Our experiments show that ST^2 GPU practically provides the same performance as the baseline: the execution time is within 0.36% of the baseline on average. The worst performance impact among our 23 kernels is suffered by *dwt2d_K1*, which shows a still small 3.5% slowdown.

The voltage level shifters required for ST^2 adders to interface the new power domain have negligible area and power overheads. Level shifters in a 45 nm technology can be made at 2.8 μm^2 [20]. We estimate that, using level shifters for each adder's input operands and outputs on the chip, even without scaling from 45 nm to 12 nm FinFETs, these level shifters in total occupy less than 5.5 mm^2, which for a NVIDIA Titan V Volta is 0.68% of the 815 mm^2 chip area. [21] shows that level shifters for 16 nm FinFETs consume 1.38 fJ per transition and 307 nW in static power. Assuming these level shifters in our design, their total static power consumption for a Titan V Volta chip without any scaling to 12 nm FinFETs is only 0.6 W. Under the worst case estimation that every single bit of every instruction that goes through an adder unit flips, thus consuming the maximum amount of energy in the level shifters, the total dynamic power consumption of the level shifters, averaged across all kernels in our suite, is a mere 470 μW. This overestimated level shifter overhead amounts to just a 0.5% penalty in our average system energy savings bringing it down to 18.5%. Finally, the worst-case delay per falling or rising transition for a 500 mV to 790 mV crossing is only 20.8 ps; in our analysis we also consider the additional delay imposed by the level shifters at the inputs and outputs of our ST^2 adders.

Finally, the ST^2 GPU area overhead is negligible, as we illustrate by considering a hypothetical NVIDIA TITAN V Volta with ST^2 GPU. Every SM has a 448-byte CRF (16×224 bits), thus the entire chip requires just 35 kB of total additional area. Moreover, each slice (except 0) has 2 bits for the state and Cout DFFs (Figure 4). Thus, each ALU adder requires an additional 14 bits, and FP32 and FP64 adders need 4 and 12 bits, respectively, for the mantissa adders. Overall, the space requirements for the DFFs amount to an additional 15 kB per chip. This brings the ST^2 overhead to a total of 50 kB per chip, which is a mere 0.09% of the on-chip caches and register files.

VII. Related Work

Approximate speculative adders [10]–[13] split execution into multiple slices that run in parallel with predicted carry-ins. However, they do not employ error correction and supply wrong results whenever a carry-in is mispredicted. VLSA [14] speculates on carry-ins, but detects mispredictions and occupies additional cycles to recompute with the corrected carry-in if an error occurs. CASA [13] provides a static prediction for all slices' carry-ins based on the correlation between the input operands. VaLHALLA [7] extends CASA to a variable-latency speculative adder that speculates carry-ins for all operations. In contrast, ST^2 employs speculation only when needed, and introduces novel concepts such as per-thread history-based predictions and thread-history sharing, and is adapted to GPUs.

VIII. Conclusions

Just like most modern chips, GPU scaling is hampered by power limitations. We address this problem with *ST^2 GPU*, a GPU architecture that employs history-based speculative adders that produce guaranteed correct results while saving power. We explore the design space of the speculative mechanisms and arrive at an adder design that shows high accuracy (91% on average) and high power savings (70% of the nominal adder power). Overall, ST^2 GPU reduces the energy consumed by a GPU chip by 21% (and chip+DRAM by 19%), with minimal area overhead and practically no performance impact.

References

[1] "Top500 list," November 2020.
[2] A. Snell and L. Segervall, "HPC application support for GPU computing," 2017.
[3] NVIDIA, "Whitepaper: NVIDIA Tesla V100 GPU architecture," 2017.
[4] S. Che, M. Boyer, J. Meng, D. Tarjan, J. W. Sheaffer, S. Lee, and K. Skadron, "Rodinia: A benchmark suite for heterogeneous computing," in *IISWC*, 2009.
[5] NVIDIA, "CUDA-9.1_Samples."
[6] J. A. Stratton, C. Rodrigues, I.-J. Sung, N. Obeid, L.-W. Chang, N. Anssari, G. D. Liu, and W.-M. Hwu, "Parboil: A revised benchmark suite for scientific and commercial throughput computing," in *IMPACT Technical Report, IMPACT-12-01, UIUC*, 2012.
[7] A. M. Gok and N. Hardavellas, "VaLHALLA: Variable latency history aware local-carry lazy adder," in *GLSVLSI*, 2017.
[8] NVIDIA, "Parallel thread execution ISA version 7.0," 2020.
[9] J. M. Rabaey, *Digital Integrated Circuits: A Design Perspective*. Prentice-Hall, 1996.
[10] A. B. Kahng and S. Kang, "Accuracy-configurable adder for approximate arithmetic designs," in *DAC*, 2012.
[11] J. Hu and W. Qian, "A new approximate adder with low relative error and correct sign calculation," in *DATE*, 2015.
[12] X. Chen, A. M. Eltawil, and F. J. Kurdahi, "Low latency approximate adder for highly correlated input streams," in *ICCD*, 2017.
[13] G. Liu, Y. Tao, M. Tan, and Z. Zhang, "CASA: Correlation-aware speculative adders," in *ISLPED*, 2014.
[14] A. K. Verma, P. Brisk, and P. Ienne, "Variable latency speculative addition: A new paradigm for arithmetic circuit design," in *DATE*, 2008.
[15] O. J. Bedrij, "Carry-select adder," *IRE Trans. Electr. Computers*, 1962.
[16] A. Bakhoda, G. Yuan, W. Fung, H. Wong, and T. M. Aamodt, "Analyzing CUDA workloads using a detailed GPU simulator," in *ISPASS*, 2009.
[17] M. Khairy, A. Jain, T. M. Aamodt, and T. G. Rogers, "A detailed model for contemporary GPU memory systems," in *ISPASS*, 2019.
[18] Synopsys, "Synopsys DesignWare library."
[19] J. Leng, T. Hetherington, A. ElTantawy, S. Gilani, N. S. Kim, T. M. Aamodt, and V. J. Reddi, "GPUWattch: Enabling energy optimizations in GPGPUs," in *ISCA*, 2013.
[20] W. Liu, E. Salman, C. Sitik, and B. Taskin, "Enhanced level shifter for multi-voltage operation," in *ISCAS*, 2015.
[21] A. Shapiro and E. G. Friedman, "Power efficient level shifter for 16 nm FinFET near threshold circuits," *IEEE Trans. VLSI Systems*, 2016.