iGPU: Enabling GPUs for Intermittent AI on Batteryless Systems

Byounguk Min[1], Yilun Wu[2], Yuchen Zhou[1], Kasim Sinan Yildirim[3], Khakim Akhunov[4],
Jianping Zeng[5], Jongouk Choi[6], Wenjie Xiong[7], Dongyoon Lee[2], Changhee Jung[1]
[1]Purdue University, West Lafayette, IN, USA [2]Stony Brook University, Stony Brook, NY, USA
[3]University of Trento, Trento, Italy [4]imec, Leuven, Belgium [5]Arizona State University, Tempe, AZ, USA
[6]University of Central Florida, Orlando, FL, USA [7]Virginia Tech, Blacksburg, VA, USA
{min87, zhou1166, chjung}@purdue.edu {yiluwu, dongyoon}@cs.stonybrook.edu kasimsinan.yildirim@unitn.it
khakim.akhunov@imec.be jpzeng@asu.edu jongouk.choi@ucf.edu wenjiex@vt.edu

***Abstract*— This paper proposes iGPU, the first GPU proposal for energy harvesting systems (EHSs), enabling *intermittent AI* workloads where power failure is the norm. The key advantage of iGPU is that it completely eliminates the need for costly just-in-time (JIT) checkpointing of the GPU's huge register file (RF). Instead, iGPU leverages compiler support to ensure correct recovery from power failure. The compiler divides the program into idempotent regions, each persisting only a small set of live-out register updates, allowing the vast RF to remain volatile. However, applying idempotent recovery to GPU's SIMT architecture poses a unique challenge because the standard post-dominator (PDOM) stack inherently discards the state of completed branch paths before divergent threads re-converge; if power failure occurs in between, the interrupted region may fail to re-execute correctly. To address this, iGPU lets the PDOM stack retain recovery metadata for all divergent subgroups until re-convergence, while aligning the idempotent region boundaries with divergence points. This hardware/software co-design keeps the PDOM hardware simple—still small enough for JIT checkpointing—while ensuring correct recovery even when power failure occurs with a warp diverged. As a result, iGPU makes GPUs a viable and high-performance option for EHSs for the first time. Experimental results show that across 21 applications, iGPU delivers an average speedup of 2.38–2.71× over jGPU, which adopts JIT checkpointing, 7.76–8.85× over a CPU-based EHS, and 1.56–1.69× over MOUSE, an intermittent compute-in-memory accelerator.**

I. Introduction

Batteryless energy harvesting systems (EHSs) [1]–[4] are widely used for long-lived sensing in challenging deployments, including wildlife tracking [5]–[7], marine monitoring [8]–[10], and smart farming [11]–[13]. These systems increasingly interpret sensor streams *on-device* to achieve low latency while minimizing communication [14], [15], which motivates TinyML workloads—primarily lightweight deep neural network (DNN) inference [16]–[19].

Such devices rarely have a stable power supply; they scavenge energy from ambient sources, *e.g.,* solar, thermal gradients, radio frequency, and ocean waves [6], [8], [20]–[22]. These sources are weak and unpredictable, causing available energy to fluctuate with environmental conditions and resulting in frequent power outages. Thus, forward progress is dictated by how well harvested energy—buffered in a small capacitor—aligns with application demand [23]. This paradigm has ushered in *Intermittent AI* [14], [24]–[26], where energy efficiency determines how much inference progress EHSs can make before the next inevitable outage.

Their intermittent nature fragments program execution into short power cycles, making state preservation essential for correct operation against repeated power outages. To operate under intermittent power, EHSs commonly employ non-volatile processors (NVPs) that just-in-time (JIT) checkpoint registers to non-volatile flip-flops (NVFFs) upon power failure [27]–[33], with non-volatile memory (NVM) as main memory to preserve program state across outages. Despite the success of NVP-based CPU-only EHSs, they are insufficient for bringing intermittent AI to practice, as modern DNN inference is dominated by data-parallel operators, *e.g.,* dot products, matrix multiplications, and convolutions. The crux of the problem is that CPU-only EHSs lack the parallel compute resources needed to execute these operators efficiently within a limited energy budget, though prior work partially addresses this by using the low-energy accelerator (LEA) [34] of MSP430 [35] microcontrollers for intermittent DNN inference [36]–[38].

A different direction is to leverage compute-in-memory (CIM) [39]–[44] that can accelerate inference operations directly within or near NVM (STT-MRAM) arrays; each computational RAM (CRAM) tile contains an array of magnetic tunnel junctions. By exploiting tile-level parallelism and reducing data movement, CIM can improve forward progress for intermittent inference with minimal backup and recovery overhead owing to the non-volatility of STT-MRAM. However, end-to-end DNN support in CIM remains limited by poor programmability and high integration costs, making it difficult to accommodate rapidly evolving inference models on EHSs.

A seemingly attractive alternative is to keep the batteryless sensor node simple by offloading computation over a wireless link to an edge server backed with GPUs and stable power. Although this avoids backup and recovery overhead while exploiting the mature CUDA ecosystem, it is often infeasible or inefficient for EHS deployments. In settings such as space systems, wide-area farms, and wildlife-tracking forests, building and maintaining connectivity infrastructure can be prohibitively expensive in practice. Even where connectivity is available, communication latency and energy consumption

often outweigh the cost of local inference. A more logical solution is therefore to integrate GPUs directly into EHSs, enabling on-device intermittent AI. This, however, requires GPUs to execute correctly under intermittent power, creating a compelling need for GPU-enabled *intermittent inference*.

Nevertheless, equipping EHSs with a GPU presents a daunting challenge, since the GPU register file (RF) is orders of magnitude larger than its CPU counterpart yet must be preserved across power outages. Naively applying an NVP-style JIT checkpointing mechanism to the GPU RF would incur prohibitive energy costs (39.8× those of NVP's JIT checkpointing), necessitating impractically large capacitors and long charging times [45] (∼149 s with a 1 mF capacitor at 0.5 mW). Employing NVFFs as checkpoint storage for such a vast RF is likewise infeasible because of the enormous area overhead and associated manufacturing challenges.

To this end, this paper proposes iGPU, the first GPU design for EHSs, enabling energy-efficient intermittent AI workloads. iGPU avoids JIT checkpointing the GPU's massive RF, thus significantly reducing hardware complexity. For this purpose, iGPU makes use of *roll-back recovery* [46]–[55] to resume power-interrupted program, tailoring the recovery mechanism to GPU's *warp-level* execution under intermittent energy. When power fails, all active warps are interrupted, potentially at different program points. Upon power restoration, iGPU recovers interrupted warps on a per-warp basis; once an interrupted warp is scheduled again, it is redirected to its own safe recovery point and resumes execution there.

To realize this warp-level recovery, the iGPU compiler leverages idempotent processing [56]–[63]. Specifically, the compiler partitions the input program into a sequence of idempotent regions that contain no memory write-after-read (WAR) dependence, enabling side-effect-free re-execution [64]. In addition, the iGPU compiler inserts stores that persist each region's live-out registers to the persistent domain; to illustrate, these register-checkpoint stores may write to the L1 cache, which is assumed to be JIT checkpointed to NVM upon power loss. At reboot time, when an interrupted warp is rescheduled, iGPU restarts that warp from the beginning of its interrupted region—after reloading the region's live-in registers from NVM. Consequently, iGPU achieves per-warp recovery through region-level re-execution while providing crash consistency and persistence guarantees. Nonetheless, realizing per-warp recovery correctly on GPUs is fundamentally more challenging than on prior CPU-based EHSs.

A key difference from idempotent recovery in CPU-based EHSs is that power failure may occur while a warp is executing under control divergence, invalidating conventional idempotent recovery; different thread subgroups within the warp may then reside in different idempotent regions along distinct control-flow paths. Moreover, GPU's standard post-dominator (PDOM) stack inherently discards the state of completed branch paths before divergent threads re-converge; if power failure occurs in between, naively rolling back the warp to the most recent region boundary can cause some threads to miss their required idempotent recovery—due to their discarded state—or to restart from an incorrect point.

To address this GPU-specific recovery challenge, iGPU introduces divergence-aware region partitioning and extends the PDOM stack [65], [66] to preserve, via JIT checkpointing upon power loss, the control-flow metadata needed for recovery. Based on this support, iGPU enables a *divergence-aware* recovery mechanism that redirects divergent threads to the correct recovery point while maintaining a consistent control-flow state, even when outages occur during divergence.

Another challenge for iGPU is that some idempotent regions may be too large, particularly on GPUs, where re-executing long regions is costly due to the large number of threads per warp; repeated outages during such re-execution can waste harvested energy and even lead to *stagnation* [52] that prevents forward progress. For this reason, iGPU applies *capacitance-aware region partitioning* [52] to further split overly long regions so that each region's energy consumption fits within the energy stored in a fully charged capacitor—which commodity EHSs typically have available at reboot time. Unlike prior region partitioning for simple CPUs such as MSP430, iGPU performs this partitioning using an energy estimate that takes into account the pipeline behavior of a streaming multiprocessor (SM). In particular, iGPU eliminates stagnation not only during normal execution but also during recovery, guaranteeing forward progress across power outages, and avoiding the waste of hard-earned energy.

Taken together, these techniques enable iGPU to support energy-efficient intermittent inference on GPUs. Experimental results demonstrate that across 21 applications, iGPU achieves an average speedup of 2.38–2.71× over jGPU, which relies on NVP-style JIT checkpointing originally developed for energy-harvesting CPUs, and 7.76–8.85× over the CPU-based EHS baseline. iGPU also outperforms MOUSE, a CRAM-based inference platform for EHSs, by 1.56–1.69× on the ML workloads relative to MOUSE's best-performing 16-tile configuration, thus remaining competitive with the state of the art in intermittent inference as well.

It is important to note that iGPU achieves these substantial performance gains with minimal hardware modifications. In particular, iGPU extends only the PDOM stack to enable divergence-aware recovery, incurring a small per-SM hardware overhead (15 times smaller than jGPU's hardware cost) while leaving the critical path unchanged. In summary, the principal contributions of iGPU are as follows:

- To the best of our knowledge, iGPU is the first to enable GPUs to operate on EHSs, paving the way for the democratization of intermittent AI.
- iGPU guarantees correct power failure recovery even in the presence of control divergence, with lightweight hardware support.
- iGPU delivers high-performance intermittent computation, achieving nearly an order-of-magnitude speedup over CPU-based EHSs.
- Across a range of capacitor sizes and energy-harvesting conditions, iGPU outperforms its competing approaches in most cases by a significant margin.

II. Background And Motivation

A. Architecture of CPU-Based Energy Harvesting Systems

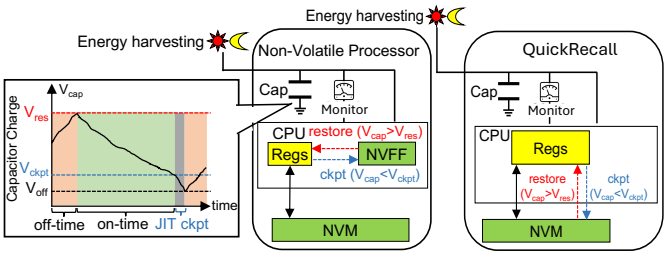


Fig. 1: JIT checkpointing (left) and two typical CPU-based EHSs that employ it: NVP (middle) and QuickRecall (right)

EHSs must tolerate frequent power outages while preserving sufficient program state to resume execution correctly after each outage. Figure 1 shows the execution model and JIT checkpointing behavior of typical NVP-based designs as the capacitor voltage varies. NVPs execute only within a power cycle, *i.e.,* from when the capacitor charges to V_{res}, triggering a reboot, until the voltage drops below a predefined checkpoint threshold V_{ckpt}, which triggers a JIT checkpoint.

The fundamental problem is that program state, including the RF, remains volatile and disappears on power failure. To preserve the state, prior work has widely adopted *roll-forward* recovery for its simplicity and ability to resume execution from the power-interruption point. It is typically implemented through JIT checkpointing [67]–[70]; when the voltage monitor senses imminent power failure (at V_{ckpt}), the processor freezes the instruction pipeline and saves volatile register state either to NVM, as in QuickRecall [71], or to non-volatile flip-flops (NVFFs) [72], [73], as in NVPs, before entering hibernation to recharge the capacitor. Once the resulting voltage rises above the restoration threshold V_{res}, the checkpointed state is restored and execution resumes from the power-interruption point.

B. GPU Architecture and Control Divergence Handling

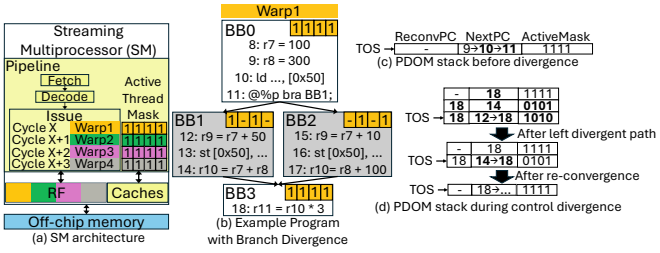


Fig. 2: Control divergence handling via PDOM stack

GPUs exploit massive thread-level parallelism (TLP) via the single instruction, multiple threads (SIMT) model. The SM fetches and decodes instructions sequentially for a warp, then executes each issued instruction in lockstep across the warp's active threads, as indicated by the active thread mask. Figure 2(a) illustrates this at the SM level using a 4-thread warp, where a newly issued instruction starts with all threads active (mask `1111`), while Figure 2(b)–(d) show how the PDOM stack manages the thread masks. When threads in a warp take different paths at a conditional branch, the active mask is updated so that only the threads on the currently executed path remain active, whereas the others are disabled; the masks are updated again when paths re-converge. The PDOM stack manages this by tracking per-subgroup state in each entry, which holds `ReconvPC` (the re-convergence PC), `NextPC` (the next PC for that subgroup), and the `ActiveMask`.

Figure 2(b) shows example code in which *Warp1* diverges at the conditional branch of *BB0*. Until the warp reaches the divergence point, the PDOM stack holds a single entry, and only its `NextPC` advances, as shown in Figure 2(c). Upon branch divergence, the stack is updated as shown at the top of Figure 2(d). The original entry is retained with its `NextPC` set to the re-convergence point, while two additional entries are pushed for the two divergent paths. These path entries share the same `ReconvPC` (line 18) but have different `ActiveMask` values. Note that the entry pointed to by the top-of-stack (TOS) pointer is executed first, corresponding to the left path in this example. Once its `NextPC` reaches the re-convergence point, the entry is popped and TOS advances to the other path entry to execute the remaining divergent threads, as shown in the middle of Figure 2(d). After all path entries are popped, the stack returns to the original entry as shown at the bottom of Figure 2(d), indicating the fully re-converged warp.

C. jGPU: A JIT-Checkpointing GPU for EHSs

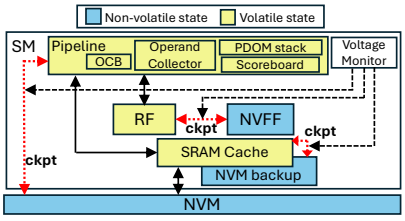


Fig. 3: jGPU for JIT checkpointing the entire volatile SM state

Unlike a typical CPU, a GPU replicates core components such as the instruction pipeline, RF, and caches within each SM. Consequently, designing GPUs for EHSs requires SM-level hardware support to ensure recoverability across power failure. The most straightforward approach is to adopt roll-forward recovery using NVP-style JIT checkpointing, resulting in what we refer to as a JIT-checkpointing GPU (jGPU). However, this approach requires extensive microarchitectural modifications to checkpoint the entire volatile state.

Specifically, once its voltage monitor detects an imminent power outage, jGPU must preserve the volatile SM state (yellow in Figure 3). That is, the large RF is checkpointed to NVFFs, with backup performed in *batches* to amortize the checkpoint overhead, while the dirty blocks of the SRAM cache are checkpointed to its NVM counterpart as in NVSRAM [74]–[76]. Similarly, the pipeline components, including the operand collectors, the scoreboard, and the PDOM stack,

are checkpointed to NVM for recovery of in-flight execution from power failure. Since some instructions may have been issued from the operand collectors but not yet completed at the failure, jGPU additionally places an Operand Collector Buffer (OCB) between the operand collectors and the execute stage, whose contents are JIT checkpointed as well.

TABLE I: V_{Backup} of jGPU and iGPU

Design	Capacitor Size			
	50 µF	100 µF	500 µF	1000 µF
jGPU	4.2521 V	3.2650 V	2.1735 V	1.9955 V
iGPU	1.8018 V	1.8009 V	1.8001 V	1.8000 V

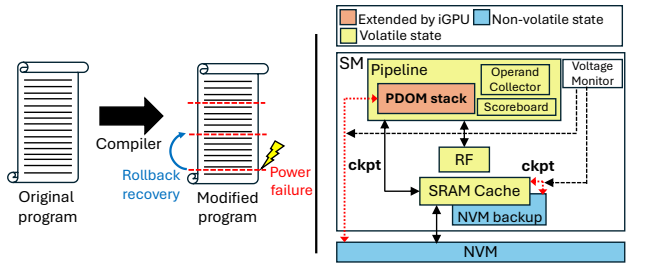
The fundamental downside of jGPU is its prohibitive backup energy overhead. To complete JIT checkpointing safely without losing state, the capacitor must retain sufficient energy, denoted E_{Backup}, to save the entire architectural state before the system reaches its minimum operating voltage V_{min} and shuts down. This requirement in turn determines the backup voltage threshold V_{Backup} that triggers checkpointing. Given a capacitor of size C (capacitance) and the GPU's V_{min}, E_{Backup} and V_{Backup} are computed as follows.

$$E_{\text{Backup}} = \frac{1}{2} * C(V_{\text{Backup}}^2 - V_{\text{min}}^2) * \text{Number of SMs} \quad (1)$$

$$V_{\text{Backup}} = \sqrt{\frac{2 * E_{\text{Backup}}}{C * \text{Number of SMs}} + V_{\text{min}}^2} \quad (2)$$

Assuming a minimum operating voltage (V_{min} = 1.8 V) and using the state-of-the-art voltage monitor [77], Table I reports the V_{Backup} values derived from Equation 2 across different capacitor sizes. The high energy demand for GPU state preservation forces V_{Backup} to impractical levels, especially for a small capacitor (50 µF). This creates two major drawbacks: (1) additional hardware support, such as NVFFs, is required for RF preservation, significantly increasing area overhead given that the RF is even larger than the caches, and (2) the recharge time before reboot becomes much longer due to the elevated V_{Backup}. These drawbacks render roll-forward recovery approaches based on full JIT checkpointing—exemplified by jGPU—unsustainable for EHSs.

III. iGPU Approach



(a) iGPU Compiler support (Software) (b) iGPU Architectural support (Hardware)

Fig. 4: Overview of iGPU and its software/hardware co-design

The goal of iGPU is to offer a practical yet performant GPU solution for EHSs. To this end, as shown in Figure 4(a), iGPU leverages compiler-directed roll-back recovery (Section IV)—instead of costly roll-forward recovery that requires jGPU's full-state JIT checkpointing. However, compiler support alone is insufficient for GPUs due to their unique control-divergence challenges detailed in Sections IV-D1 through IV-D3. To overcome these challenges, iGPU takes a software/hardware co-design approach. Figure 4(b) shows that this approach makes minimal architectural modifications confined to the PDOM stack (Section V-A). At run time, iGPU updates the stack (Section V-B) to ensure correct, lightweight GPU recovery (Section V-C).

IV. iGPU Compiler

A. iGPU Compiler Workflow

As shown in Figure 5, the iGPU compiler first (1) divides the program into a series of *warp-level* idempotent regions, each free of memory WAR dependence and thus re-executable to recover from power failure. However, due to control divergence in GPUs, conventional idempotent recovery used in CPUs is insufficient to recover the registers of all warps. In response, the iGPU compiler applies (2) control-divergence-aware region partitioning, inserting region boundaries at divergence and re-convergence points to simplify the recovery of all warps' registers. Then, the compiler conducts (3) capacitance-aware region partitioning to split overly long regions—whose estimated energy consumption exceeds a fully charged capacitor's energy—into shorter regions, thereby reducing re-execution overhead across outages and preventing stagnation. Both steps also split regions wherever internal memory WAR dependence would otherwise arise.

For each region, the iGPU compiler identifies its live-out registers, inserts their checkpoint stores, and generates its *recovery block*, which restores the registers and restarts the region from the beginning. In this way, any power-interrupted region can correctly recover by invoking its recovery block when power comes back. In particular, when a region's live-in registers are also live-out, creating WAR dependence on them, re-executing the region risks overwriting their checkpoints. To prevent this, the iGPU compiler performs (4) *register renaming* [64] by default, falling back to *checkpoint coloring* [78] if the renaming results in additional spilling. The compiler then (5) allocates checkpoint storage for these live-out registers in either NVSRAM shared memory or NVM local memory (accessed via cache), prioritizing shared memory based on program usage; checkpoint data exceeding the shared memory capacity is placed in local memory. Finally, the transformed PTX undergoes (6) register allocation via CRAT [79], producing allocated PTX for final code generation by the GPU backend. The rest of this section details each compilation phase; for clarity, the presentation order deviates slightly from the workflow in Figure 5.

B. Capacitance Aware Region Partitioning

Forming idempotent regions—with control-divergence-aware region boundary placement (Section IV-D2)—and checkpointing their live-out registers allows iGPU to achieve

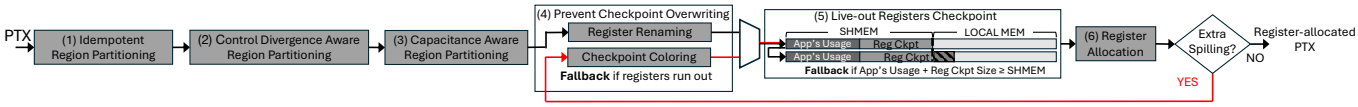


Fig. 5: Workflow of iGPU compiler

correct power failure recovery. However, correct recovery alone does not guarantee forward progress under frequent power failure. If a region is excessively long, failure near its end still forces recovery to re-execute the entire region from its beginning, consuming a substantial fraction of the hard-won energy. Even worse, the re-execution may itself be interrupted by power failure again—potentially multiple times—before the region ends. In the worst case, the region suffers stagnation [52], [80], [81], rendering forward progress impossible.

To ensure forward progress regardless of power failure frequency, the iGPU compiler performs capacitance-aware region partitioning inspired by Safe Active Time (SAT) [52] yet adapted to the GPU execution model. SAT is the execution time that an energy harvesting system is guaranteed to sustain on a fully charged capacitor, and is calculated using a capacitor leakage model [52] under worst-case power consumption—obtained via the Jetson Nano GPU power model (Section VI-B). In iGPU, the compiler estimates each region's worst-case execution time (WCET) and partitions any overly long region into smaller regions so that, on a best-effort basis, each satisfies SAT and can thus complete upon reboot without encountering power failure.

To infer each region's WCET, iGPU calculates the WCET of each PTX instruction, considering the worst-case latency of the Jetson Nano GPU SM pipeline and memory hierarchy. Compute instructions use worst-case pipeline latencies, while memory instructions, except shared memory accesses, assume the worst access path to global memory (ReRAM)—via a cache miss—with the NVM latency modeled using NVSim [82]. The iGPU compiler analysis then accumulates these per-instruction WCETs under a pipeline-aware estimate and inserts a new region boundary whenever the accumulated WCET plus the recovery code execution time exceeds the SAT for a given capacitor size, as follows:

$$\sum_{i \in Inst(R_{\text{curr}})} WCET(i) + L_{\text{NVM}} \cdot \left| LiveIn(R_{\text{curr}}) \right| \tag{3}$$

where R_{curr} is the current region starting from the prior boundary, $Inst(R_{\text{curr}})$ gives the set of instructions in the current region, $LiveIn(R_{\text{curr}})$ denotes the set of live-in registers in the region, and L_{NVM} is the NVM load access latency.

In Equation 3, the first term captures the WCET estimate of the current region to be ended with the new boundary during the compiler analysis, while the second term accounts for the recovery code execution time required at reboot to reload the region's live-in registers from their checkpoints in NVM, *i.e.,* ReRAM. In practice, iGPU can effectively avoid stagnation at both run time and recovery time, ensuring forward progress across power outages.

Note that unlike prior GPU WCET analysis [83], [84], iGPU's compiler analysis is much simpler; it ignores warp scheduling delay, which is negligible compared to NVM latency. Specifically, the scheduling delay becomes significant only when a large number of concurrently executing warps contend for memory; however, power failure typically occurs before such contention develops, resetting the number of scheduled warps. This is confirmed by the number of warps executed during each power-on period, which is only around 11, compared with 31 on average in the absence of power failure. Furthermore, there is no need to model the control divergence for WCET analysis, because iGPU places region boundaries at both divergence and re-convergence points—to minimize hardware cost and simplify the recovery protocol (Section V-B–V-C).

In the rare case that the WCET estimate still leads to stagnation (never observed in our experiments), iGPU can rely on a simple yet effective fallback mechanism for potentially stagnating regions. iGPU can set up a volatile watchdog timer at the beginning of such a region and, upon timer expiration, checkpoint the warp registers along with cache and PDOM data, allowing the warp to resume from that point by restoring the checkpointed state. Even if such a region is power-interrupted, it could thus gradually make forward progress until reaching the next region. To determine the watchdog timer value, iGPU could use either the SAT or profiling similar to prior work [49].

C. Region-Level Persistence

Once the regions are formed, iGPU must ensure that their live-in registers—required for idempotent re-execution—persist across power failure. This requirement, known as *region-level persistence* [51], [85], [86], dictates that all stores, including the checkpointed live-out registers of the preceding region—which can be the current region's live-in registers—must be persisted before execution enters the current region.

Without this guarantee, warp-level idempotent recovery could not succeed, since some live-in registers of an interrupted region may have been lost if they were not persisted before the failure. To achieve region-level persistence, iGPU takes advantage of NVSRAM for all on-chip caches—L1-instruction and L1-data caches, shared memory, constant cache, and texture cache—as well as NVM for global memory. All in-region stores are thus guaranteed to persist by the end of the region; their results are written to the data cache or shared memory immediately upon instruction commit, where they are retained across power failure and can be loaded for recovery.

Specifically, Figure 6 shows how iGPU ensures region-level persistence. At time T_1, when region boundary $RB2$ is reached, the live-out register checkpoints of the prior region $R1$, which appear as two ▲ in the figure, are already treated as persisted in the data cache or shared memory—destined for JIT checkpointing on an outage—following the commit of instructions at lines $1C$ and $2C$. Suppose that Mem(T_n) represents the states of checkpointed registers in caches, shared memory, and global memory at T_n. Mem(T_1) thus includes the two ▲, while Mem(T_2) includes the checkpointed registers at lines $6C$ and $7C$ in region $R2$, *i.e.,* ■ and ✖. If region $R2$ encounters power failure at T_2, with the checkpoint stores at lines $6C$ and $7C$ already persisted, iGPU initiates recovery from region boundary $RB2$ at T_3. Note that Mem(T_2) and Mem(T_3) remain identical across power failure, *i.e.,* Mem(T_1) $\neq$ Mem(T_2) $=$ Mem(T_3). Since the live-in registers of the interrupted region $R2$ are available in Mem(T_3), *i.e.,* the two ▲, iGPU loads them before restarting $R2$.

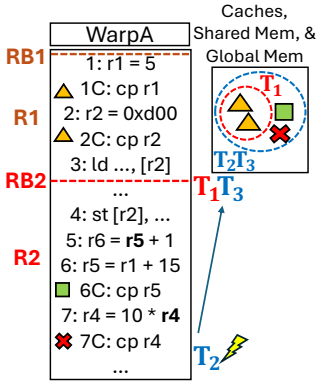


Fig. 6: Region-level persistence

A potential concern is that iGPU's checkpoint stores for $R2$'s live-out registers at lines $6C$ and $7C$ (the ■ and the ✖) overwrite the checkpoints of $R2$'s own live-in registers `r5` and `r4`. Recovery from $RB2$ would then load the overwritten values, breaking idempotent re-execution. To prevent this checkpoint-overwriting problem, the iGPU compiler leverages *register renaming*; if a checkpointed live-in register is overwritten within a region, the compiler renames its later definition to a new register, preserving the checkpointed value for recovery. However, the renaming can increase register pressure and lead to more register spills. When further renaming is infeasible due to the lack of available registers, iGPU falls back to *checkpoint coloring*; refer to prior work [78] for details.

D. Control Divergence Challenges for Idempotent Recovery

Although the iGPU compiler provides the essential mechanisms (Sections IV-B and IV-C) for power failure recovery, these alone are insufficient to recover register state across all threads due to GPU's unique challenges of control divergence as discussed in the remainder of this section.

1) Challenge 1: Loss of Divergence State on Power Failure: The first challenge arises when power failure occurs after a warp has diverged but before its threads re-converge, causing the warp's divergence state to be lost. In the wake of the failure, the system can determine neither which threads took each divergent path nor where within each path their execution must resume, rendering correct recovery impossible. To illustrate, consider the execution of *WarpA* in Figure 7, starting with the non-divergent case. If power failure occurs before

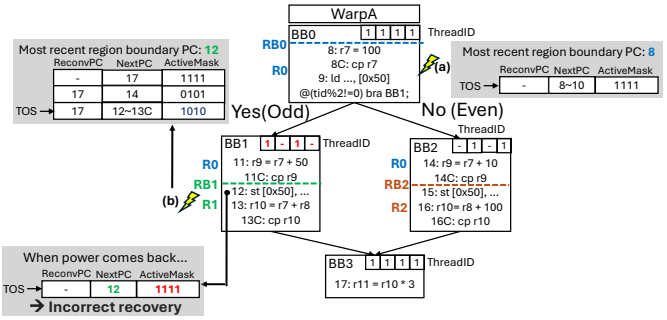


Fig. 7: Divergence state lost on power failure

the divergent branch at point (a) between lines 8 and 10, the PDOM stack holds only the single entry for the entire warp, whose `NextPC` alone is updated while its `ActiveMask` remains `1111` as all threads are active. Here, idempotent recovery is identical to that of CPUs, requiring only a restart from region boundary $RB0$ (line 8).

However, if power failure occurs in the left path after divergence (*e.g.,* between lines 12 and $13C$), re-executing from the most recent region boundary $RB1$ is incorrect. This is because, with the PDOM stack lost upon the failure, the interrupted path's `ActiveMask` is unknown, so blind re-execution defaults to the incorrect mask (*e.g.,* `1111`), restarting region $R1$ on all threads and potentially corrupting their registers.

Solution: To address this, iGPU JIT checkpoints the PDOM stack right before power failure, restoring the divergence state required to track each thread subgroup's `ActiveMask`.

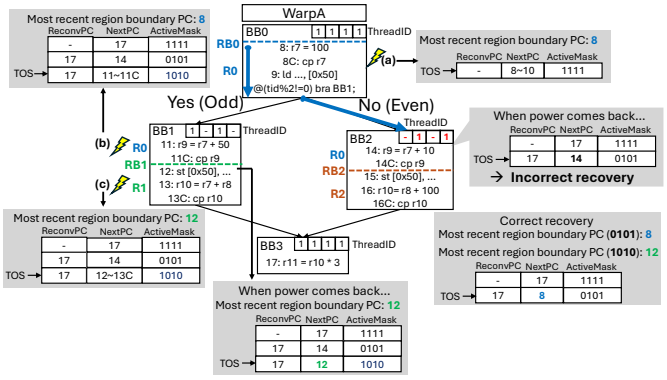


Fig. 8: Multiple idempotent recoveries required upon reboot

2) Challenge 2: Misalignment between Divergence Points and Region Boundaries: While JIT checkpointing the per-warp PDOM stacks recovers a warp's control-flow state, a unique challenge remains for GPU idempotent recovery. Specifically, compiler-defined idempotent region boundaries often do not align with hardware-managed control divergence points, as illustrated in Figure 8. If a region starting in the middle of a branch path (*divergent region*, *e.g.,* $R1$ and $R2$ in the figure) encounters power failure, simply restarting the interrupted region is insufficient, as it covers only that path's thread subgroup rather than all threads in the warp. Instead,

recovery must be initiated separately for each subgroup from the beginning of its own divergent region. Thus, a single power outage may require multiple idempotent recoveries, rendering recovery far more complex than in CPUs.

Figure 8 shows this complexity using *WarpA*. If power failure occurs at point (a) or (b) before the divergent regions are entered at $RB1$/$RB2$, *WarpA* is recovered by re-executing from $RB0$. In contrast, if $R1$ suffers the failure at point (c) between lines 12 and 13C on the left path, the odd-numbered threads must first be recovered by re-executing from $RB1$. Once these threads end the left path during recovery, the warp must resume from $RB0$ to follow the blue path, unlike standard PDOM stack semantics that would set `NextPC` to the beginning of basic block $BB2$ (line 14), assuming the flow is $BB0 \rightarrow BB1 \rightarrow BB2 \rightarrow BB3$. The reason is that even-numbered threads have not been recovered yet; another idempotent recovery must be initiated from $RB0$ for them. Only after the even-numbered threads re-execute $BB0 \rightarrow BB2$ can the warp reach the re-convergence point (line 17).

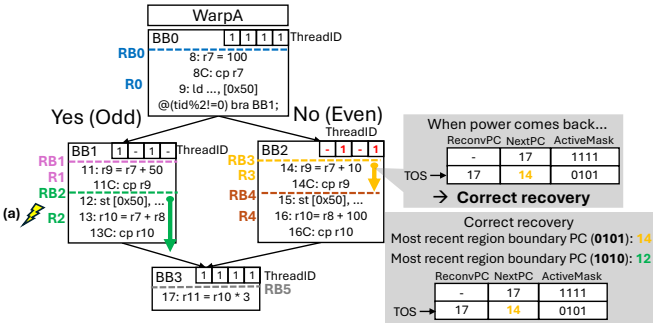


Fig. 9: Control divergence aware region partitioning

Solution: To minimize modifications to standard PDOM stack semantics, the iGPU compiler aligns region boundaries with control divergence. As shown in Figure 9, the compiler lets a new region start at the entry of each divergent path ($RB1$ and $RB3$) and at the re-convergence point ($RB5$, whose role is described in Section V-B). Since every divergent region now begins exactly where its path begins, the interrupted thread subgroup restarts from a boundary on that path, *e.g.,* along the green path from $RB2$ for the failure at point (a)—rather than from $RB0$ before the branch. The remaining subgroups are then recovered along the yellow path via standard PDOM stack handling, which already advances TOS to the next path entry, without needing to manually overwrite `NextPC` to re-execute from $RB0$ and repeat the divergence. In this way, control divergence aware region partitioning integrates the recovery of divergent warps into the existing re-convergence logic. Despite this, the compiler support is still insufficient to fully recover all threads' registers, as the standard PDOM stack discards the entries of completed divergent paths (Section IV-D3).

3) Challenge 3: Losing the Stack Entry of Completed Paths: Although the above approach (Figure 9) appears straightforward, it relies on a hidden assumption that contradicts the standard PDOM stack. That is, it assumes that entries for all

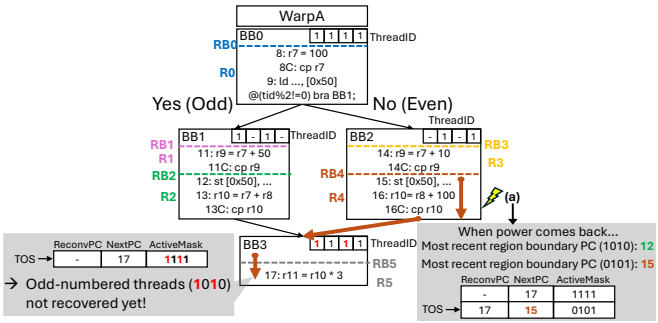


Fig. 10: Wrong recovery of a completed path at re-convergence

divergent branches remain accessible even after those branches complete. In hardware, however, an entry is popped as soon as its divergent branch completes, as shown in Figure 10, creating a critical barrier for idempotent recovery. If power failure occurs at point (a) of $BB2$, the odd-numbered threads have already finished the left path, again assuming $BB0 \rightarrow BB1 \rightarrow BB2 \rightarrow BB3$. JIT checkpointing here captures only the entries for the current divergent branch and the re-convergence point, so recovery can proceed only along the brown path from $RB4$ in that the entry for the left path has already been popped. The problem is that executing the warp instruction at the re-convergence point (line 17) requires restoring the state of all threads, including the odd-numbered ones—whose entry is no longer available. Standard PDOM stack semantics are thus incompatible with the GPU's idempotent recovery. To this end, iGPU devises a simple yet effective hardware solution below.

V. iGPU Architecture

A. Extending the PDOM Stack

iGPU extends each PDOM stack entry with two fields for recovery: (1) `RecoveryPC`, which holds the PC of the current idempotent region's recovery block, and (2) `RecoveredBit`, a volatile flag distinguishing recovered thread subgroups from unrecovered ones to prevent duplicate recovery. With these fields, iGPU manages the recovery state of a warp's divergent thread subgroups inside the PDOM stack itself. iGPU also adds a new pointer, `Top of Recovery` (TOR), which is updated independently of TOS. While TOS advances to execute each divergent path, TOR stays behind at the completed one, so the entries of completed paths are preserved between TOR and TOS. When the warp reaches the region boundary at the re-convergence point, TOR is updated to match TOS, permanently discarding all preserved entries at once.

B. iGPU Runtime: Updating the Extended PDOM Stack

Figure 11 summarizes how iGPU updates the extended PDOM stack during a normal power-on period, so that the correct state is available for JIT checkpointing and the subsequent recovery protocol (Section V-C). Figure 11(a) shows an example with *WarpB*, whose basic blocks ($BB0$–$BB3$) are partitioned to idempotent regions ($R1$–$R6$). For each region, the compiler generates a *recovery block* ($Rec1$–$Rec6$) that

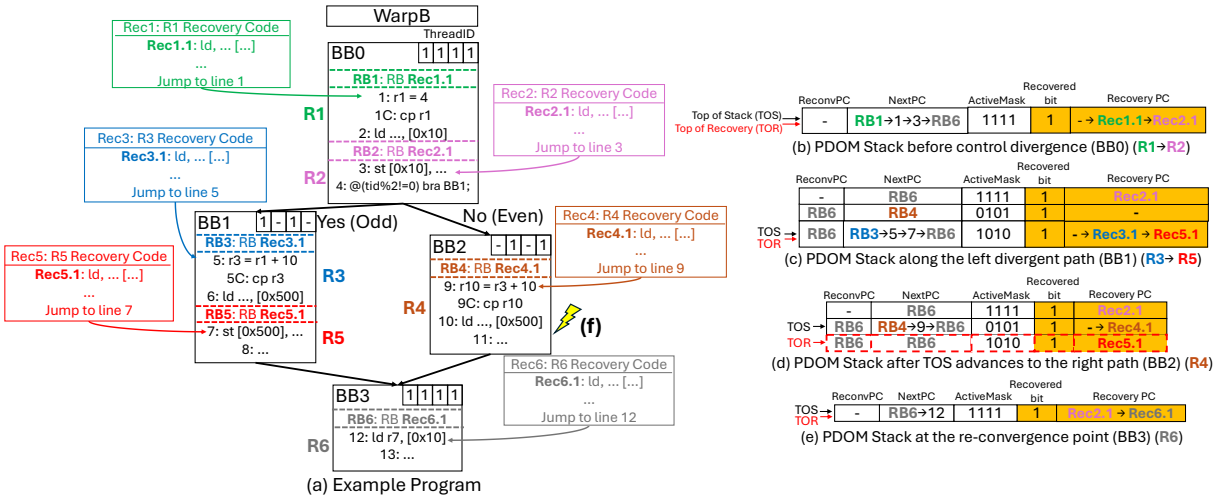


Fig. 11: Run-time operation of the extended PDOM stack in iGPU

loads the live-in registers of the region from their checkpoints and jumps to its first instruction. To associate each region with its recovery block, the compiler then inserts a special region boundary (RB) instruction ($RB1$–$RB6$) as the region's first instruction. Each RB instruction carries the PC of the corresponding recovery block as an operand. A warp executing an RB instruction writes this operand into the `RecoveryPC` field of its TOS entry, which thus always records the recovery block of the region the warp currently executes.

Figure 11(b)–(e) shows the PDOM stack at the corresponding points of *WarpB*'s execution, starting with how the TOS entry changes through $BB0$ in Figure 11(b). Right before the first instruction, `NextPC` holds $RB1$, and `RecoveryPC` is empty. Executing $RB1$ advances `NextPC` to line 1 and writes $Rec1.1$ into `RecoveryPC`. As a result, the warp enters region $R1$ with its recovery block already recorded—unlike prior work Penny [78], which records the address of $RB1$ and needs a separate map to translate it to $Rec1$. Within $R1$, only `NextPC` is updated; once $RB2$ is executed, `NextPC` advances to line 3 while `RecoveryPC` is updated from $Rec1.1$ to $Rec2.1$. At the end of $BB0$, `NextPC` points to $RB6$ (the re-convergence point's first instruction) according to the original PDOM stack semantics, whereas `RecoveryPC` still holds $Rec2.1$.

When control divergence occurs at line 4, a new entry is pushed to the PDOM stack for each divergent path, as shown in Figure 11(c). Both entries take $RB6$ as their `ReconvPC`, and each `ActiveMask` covers the corresponding subgroup of threads (`1010` for the left path and `0101` for the right). Their `NextPC` fields are set to $RB3$ and $RB4$, the region boundaries at the head of the two paths, while `RecoveryPC` is left empty on both until execution of each path begins.

Upon pushing these entries, both TOS and TOR point to the left-path entry, so *WarpB* executes the left divergent path first. Executing $RB3$ advances `NextPC` to line 5 and fills the empty `RecoveryPC` with $Rec3.1$. When the thread subgroup reaches $RB5$, `NextPC` advances to line 7 and the TOS entry's `RecoveryPC` is updated from $Rec3.1$ to $Rec5.1$. At the end of the left path (line 8), `NextPC` becomes $RB6$, which now matches the entry's `ReconvPC`. At this point, the standard PDOM stack would permanently pop the entry. Instead, iGPU defers the deallocation by advancing TOS to the right path while TOR stays at the completed left-path entry, as shown in Figure 11(d). The retained entry keeps $Rec5.1$ available until re-convergence, so the odd-numbered threads can still be recovered if power fails while the right path is executing.

With TOS now targeting the right-path entry, the even-numbered threads begin executing region $R4$, as shown in Figure 11(d). Executing $RB4$ advances `NextPC` to line 9 and populates `RecoveryPC` with $Rec4.1$. Because the right path consists of the single region $R4$, `RecoveryPC` stays at $Rec4.1$ while only `NextPC` advances. At the end of the right path (line 11), `NextPC` reaches $RB6$, matching the entry's `ReconvPC`. Since both paths are now complete, TOS moves down to the re-converged entry, which reflects its state at the end of $BB0$—all threads active (`1111`) and `RecoveryPC` still holding $Rec2.1$. When *WarpB* executes $RB6$ and enters $R6$, `NextPC` advances to line 12 and `RecoveryPC` is updated from $Rec2.1$ to $Rec6.1$. TOR then moves to match TOS, discarding the two preserved path entries at once, as shown in Figure 11(e). Consequently, placing a region boundary at the re-convergence point bounds the lifetime of these preserved entries.

C. iGPU Recovery Protocol

Upon detecting an imminent power outage, iGPU JIT checkpoints each warp's PDOM stack, excluding the volatile `RecoveredBit`. Suppose power failure occurs when program control is at point (f) in Figure 11 where (d) shows the PDOM stack state at the moment of the failure. In the wake of the failure, the recovery protocol is then triggered by the warp scheduler. When *WarpB* is scheduled (first by default), iGPU restores the entire stack and sets TOS to TOR, so recovery

TABLE II: Simulation configuration under a 100 µF capacitor and a solar power trace

	iGPU	jGPU	NVP-Inorder	NVP-OOO	CIM
Perf./Power Model	Jetson Nano GPU (Maxwell)		ARM A7 core	ARM A15 core	MOUSE [42]
#SMs (warp)/#cores/#tiles	1/2/4 SMs (32 threads per warp)		1 core	1 core	1/16/32 tiles
Clock, L1I/L1D/Const/Tex/Shmem	200 MHz, 32/32/8/32/48 KB (NVSRAMCache [87])				30.3 MHz, –
PDOM Stack/Warp Sched.	56 KB/GTO	40 KB/GTO	-		
Off-chip/Main Memory	ReRAM-based NVM (tCK=0.94 ns, tBURST=7.5 ns, tRCD=18.0 ns, tCL=15.0 ns, tWTR=7.5 ns, tWR=150 ns, tXAW=30 ns)				MTJ-based CRAM (P/AP resistance = 3.15/7.34 kΩ), Switching Current=40 µA, Time=3 ns [88]

starts from the bottom entry (left path) whose `ActiveMask` is `1010` as shown in Figure 11(d). The recovery runtime then inspects the `RecoveredBit` in the TOS entry, which is initially unset because this bit was not checkpointed, triggering warp-level idempotent recovery by overwriting `NextPC` with the stored `RecoveryPC` (*Rec*5.1). Once those threads re-execute from *Rec*5.1 through the end of the left path and reach the re-convergence point, the `RecoveredBit` is set so that rescheduling the warp does not trigger this recovery again for the already recovered thread subgroup. Next, TOS moves to the second entry in (d). Its unset `RecoveredBit` triggers recovery of the other subgroup for the right path from that entry's `RecoveryPC` (*Rec*4.1). As on the left path, these threads execute the recovery block (*R*4), re-executing the right path up to the re-convergence point at which the `RecoveredBit` is set.

VI. Evaluation

A. Experimental Setting

We implement both iGPU and jGPU within the Accel-Sim GPU simulator [89]. Since Accel-Sim targets desktop GPUs, we adapt it to model the NVIDIA Jetson Nano [90], a single-SM, 20 nm Maxwell embedded GPU, and configure both designs accordingly. To evaluate performance under frequent power outages, we conduct power modeling for the Jetson Nano using AccelWattch [91]. Because iGPU utilizes NVM for global memory, we replace Accel-Sim's memory parameters with ReRAM-based NVM parameters and, correspondingly, substitute the DRAM power components in AccelWattch with NVM power data from NVSim [82]. Unless otherwise specified, the power failure evaluations in Sections VI-C2 through VI-C4 use the default EHS configuration shown in Table II, *i.e.,* a 100 µF capacitor under a solar power trace.

The storage overhead of iGPU's extended PDOM stack and jGPU's standard PDOM stack is calculated for the worst-case scenario to ensure that JIT checkpointing of the stack remains possible regardless of control divergence depth. We model a maximum of 64 warps per SM, each provisioned with a 32-entry PDOM stack, which suffices even when all 32 threads in a warp follow a unique, nested divergence path, *i.e.,* nested control divergence happens 32 times.

As shown in Table II, iGPU and jGPU use the default Greedy-Then-Oldest (GTO) warp scheduling policy in Accel-Sim. Since iGPU's compiler transformations are decoupled from scheduling, the warp scheduler selects among ready warps regardless of their idempotent regions, even after power restoration. When a warp is first selected, iGPU redirects it to the recovery code of its current region unless it has already been recovered. Thus, some warps may still be recovering while others make forward progress.

For comparison, we establish CPU baselines using the gem5 [92] simulator to model both inorder and out-of-order NVPs. These baselines employ a prior power-recovery mechanism for pipelined EHS cores [93], NVSRAMCaches [87], and ReRAM-based NVM modeled via NVSim. Their power is modeled using existing CPU power models [94], [95]. We additionally include MOUSE [42] as a CIM baseline. Following the original MOUSE configuration in Table II, we adopt modern STT-MRAM parameters [88] and a 30.3 MHz operating frequency. The machine-learning (ML) benchmarks are evaluated on MOUSE using an in-house simulator, with each tile comprising a 1024 × 1024 array (128 KB).

B. Jetson Nano Power Modeling for iGPU

We adapt AccelWattch [91] to model the Jetson Nano GPU's power consumption. In AccelWattch, GPU power is decomposed into constant (P_{const}), static (P_{static}), and dynamic (P_{dynamic}) components as functions of active SMs and active lanes. For the Jetson Nano, we use tegrastats [96] to collect activity statistics and measure power in milliwatts on the GPU power rail, VDD_GPU as NVML [97] and nvidia-smi [98]—which AccelWattch relies on—are unavailable on this platform. Also, AccelWattch measures GPU power at 65 °C, whereas we measure the Jetson Nano's GPU power at 50 °C due to the board's more limited thermal capability [99].

From the AccelWattch framework, we derive a Jetson-specific power model by retaining only the terms relevant to the VDD_GPU rail. Because tegrastats reports only VDD_GPU, which captures the Jetson Nano GPU's P_{static} and P_{dynamic}, the board-level constant term P_{const} falls outside the measurement target and is therefore excluded. Moreover, since the Jetson Nano has only one SM, there is no separate idle-SM power contribution. For the dynamic power term, we adapt AccelWattch's component-level model to the Maxwell architecture by retaining only the units present on the Jetson Nano GPU (*e.g.,* excluding Tensor Core) and replacing the DRAM term with NVM dynamic power estimated via NVSim [82]. The resulting model is:

$$P_{\text{est},\, y\text{Lanes}} = \left(\sum_{i=1}^{N} \frac{a_i \cdot \hat{E}_i \cdot x_i}{T_{\text{elapsedTime}}} + P_{\text{dyn,NVSim}} \right) + P_{\text{static},\, y\text{Lanes}} \tag{4}$$

where the term in parentheses represents the dynamic power; $P_{\text{static},\, y\text{Lanes}}$ denotes the static power for a warp with y active lanes; and for the i-th modeled component, a_i, $\hat{E}_i$, and x_i denote its activity count, uncalibrated base dynamic energy,

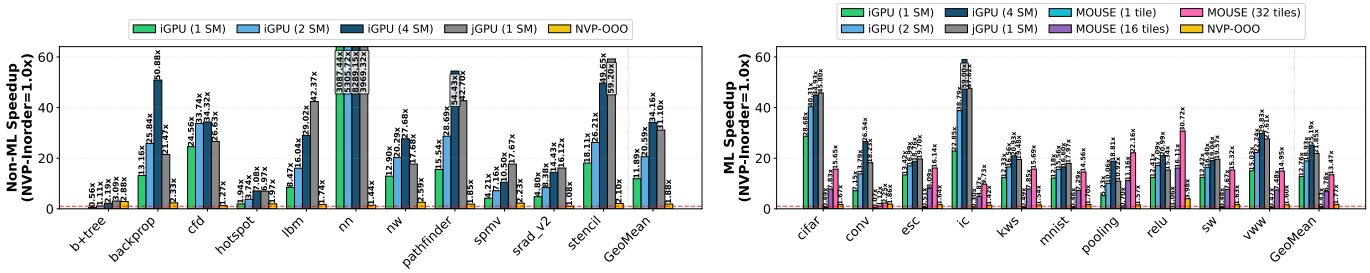


Fig. 12: Speedup of out-of-order NVP, iGPU, jGPU, and MOUSE over the inorder NVP baseline without power failure

and calibration factor, respectively. We calibrate each x_i on 66 AccelWattch microbenchmarks using the Sequential Least Squares Quadratic Programming (SLSQP) solver [100]. The final model achieves $6.8\% \pm 2.4\%$ mean absolute percentage error (MAPE) against measured VDD_GPU power.

C. Performance Analysis

1) Performance Analysis without Power Failure: Before evaluating power failure scenarios, we first examine all the schemes in the absence of power outages to characterize their baseline performance free of backup and recovery overhead. In this experiment, jGPU denotes the Jetson Nano GPU modeled in Accel-Sim [89] running the unmodified binary, *i.e.,* containing no register-checkpoint instructions for idempotent recovery. iGPU uses the same SM configuration and differs only in that it runs the binary modified by the iGPU compiler. jGPU is evaluated with the single SM of the Jetson Nano, whereas iGPU is evaluated with 1, 2, and 4 SMs, each additional SM replicating that same SM. This keeps jGPU consistent with its evaluation under power failure in Section VI-C3, where we explain why it is limited to a single SM. Likewise, the CPU baselines incur no backup/recovery overhead. For the CIM baseline, we evaluate MOUSE while varying its tile-level parallelism (1, 16, and 32 active tiles).

Because MOUSE is applicable only to ML workloads, we divide the benchmarks into non-ML and ML sets. The non-ML set consists of representative Rodinia [101] and Parboil [102] workloads, evaluated on both the GPU and CPU baselines. The ML set includes microbenchmarks whose operations are commonly used in DNN inference (convolution (conv), ReLU, and pooling), together with MLPerf [103] inference benchmarks. For fairness, we align the workload sizes across the CPU, CIM, and GPU implementations of these ML benchmarks, and measure equivalent execution windows rather than end-to-end application time, *i.e.,* the GPU kernel execution for iGPU and jGPU, the CIM kernel execution for MOUSE, and the equivalent code section for the CPU baselines.

Figure 12 shows that both GPUs deliver the strongest performance among the general-purpose architectures, whereas the OOO provides only modest improvement over the inorder baseline. MOUSE is competitive only for the ML workloads it targets, with performance scaling with the number of active tiles. Comparing the two GPUs at the same single-SM configuration, jGPU outperforms iGPU by 161.6% on the non-ML workloads and by 71.2% on the ML workloads in geometric mean. The reason is that iGPU must additionally execute the checkpoint instructions inserted by its compiler. The gap is large enough that jGPU also outperforms iGPU with 2 SMs, by 51.0% and 15.4% on the non-ML and ML workloads, respectively. Only with 4 SMs does iGPU surpass jGPU, by 9.8% and 15.3% on non-ML and ML workloads. This relative ordering, however, holds only with no power failure, since jGPU never incurs JIT checkpointing overhead in this setting.

In the absence of power failure, more SMs yield a corresponding performance improvement for iGPU, as expanding from 1 to 2 SMs improves the geometric mean by 73.2% on the non-ML workloads and by 48.4% on the ML workloads, while doubling again to 4 SMs provides additional gains of 65.9% and 33.1%, respectively.

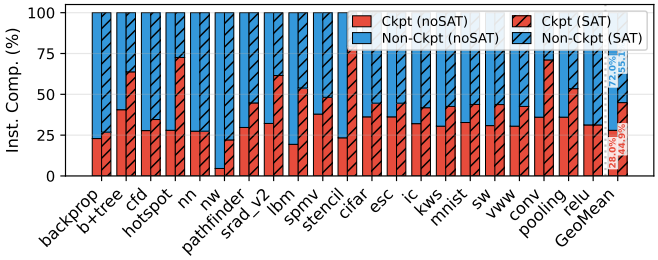


Fig. 13: Instruction breakdown (1 SM)

2) Region Analysis in iGPU Compiler: Before analyzing capacitance-aware region partitioning, we first examine the compiler decisions shown in Figure 5 for dealing with checkpoint overwriting and placement. For all evaluated benchmarks, the checkpoint overwriting problem is resolved entirely through register renaming, and none requires the fallback checkpoint coloring mechanism. Register pressure thus remains manageable even when longer regions require more renamed registers. Regarding the checkpoint-storage allocation decision (Figure 5), fallback to local memory is also rare; for most benchmarks, iGPU stores all checkpoints in shared memory, even after reflecting program's shared-memory usage.

Table III shows how capacitance-aware partitioning changes the number and size of regions across capacitor sizes and SM counts. Note that ml_cifar and ml_esc share identical rows, as do ml_kws and ml_vww, since each pair invokes the same GPU kernels and thus compiles to the same code, though they

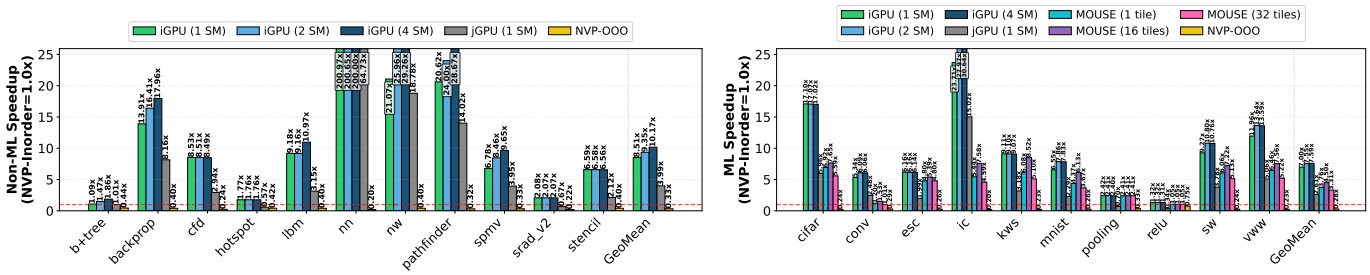


Fig. 14: Speedup of out-of-order NVP, iGPU, jGPU, and MOUSE over the inorder NVP baseline under power failure

TABLE III: Region characteristics. Each entry lists 1/2/4 SMs.

Workload	Region count					Avg region size				
	noSAT	50 µF	100 µF	500 µF	1000 µF	noSAT	50 µF	100 µF	500 µF	1000 µF
backprop	27	143/172/-	38/88/172	27/27/29	-/27/27	9.8	2.5/2.2/-	7.3/3.7/2.2	9.8/9.8/9.2	-/9.8/9.8
b+tree	92	390/399/-	330/357/399	114/131/135	-/114/131	7.3	2.2/2.1/-	2.4/2.4/2.1	6.1/5.4/5.4	-/6.1/5.4
cfd	32	773/811/-	661/687/811	38/49/562	-/37/49	31.7	2.2/2.1/-	2.4/2.3/2.1	27.4/22.8/2.6	-/27.8/22.8
hotspot	18	164/168/-	158/162/168	18/22/30	-/18/22	14.4	2.4/2.4/-	2.5/2.5/2.4	14.4/12.3/9.9	-/14.4/12.3
nn	3	9/24/-	3/4/24	3/3/3	-/3/3	16.0	6.1/2.7/-	16.0/12.5/2.7	16.0/16.0/16.0	-/16.0/16.0
nw	167	981/1016/-	620/637/1016	167/223/207	-/167/223	7.3	2.0/2.0/-	2.8/2.7/2.0	7.3/5.9/6.4	-/7.3/5.9
pathfinder	17	93/97/-	78/80/97	17/17/20	-/17/17	9.6	2.4/2.3/-	2.7/2.7/2.3	9.6/9.6/8.6	-/9.6/9.6
srad_v2	48	361/369/-	281/336/369	48/54/65	-/48/54	11.8	2.1/2.1/-	2.6/2.2/2.1	11.8/10.6/9.1	-/11.8/10.6
lbm	3	237/241/-	226/232/241	3/5/15	-/3/5	102.3	2.1/2.1/-	2.2/2.1/2.1	102.3/64.0/22.4	-/102.3/64.0
spmv	12	134/138/-	85/105/138	14/15/20	-/14/15	17.6	2.3/2.2/-	3.2/2.7/2.2	15.7/15.1/11.8	-/15.7/15.1
stencil	37	219/221/-	191/193/221	37/37/142	-/37/37	9.2	2.2/2.1/-	2.4/2.4/2.1	9.2/9.2/3.0	-/9.2/9.2
ml_cifar	85	480/511/-	365/382/511	87/88/123	-/87/88	10.5	2.4/2.3/-	3.0/2.9/2.3	10.4/10.3/7.7	-/10.4/10.3
ml_esc	85	480/511/-	365/382/511	87/88/123	-/87/88	10.5	2.4/2.3/-	3.0/2.9/2.3	10.4/10.3/7.7	-/10.4/10.3
ml_ic	74	444/495/-	313/338/495	77/80/111	-/76/80	12.4	2.6/2.4/-	3.5/3.3/2.4	12.1/11.8/8.8	-/12.2/11.8
ml_kws	99	652/703/-	518/546/703	102/112/158	-/101/112	12.2	2.5/2.3/-	3.0/2.9/2.3	12.0/11.1/8.3	-/12.0/11.1
ml_mnist	84	524/564/-	388/420/564	87/94/127	-/86/94	11.7	2.4/2.3/-	3.1/2.9/2.3	11.4/10.7/8.2	-/11.5/10.7
ml_sw	88	583/624/-	463/490/624	91/100/146	-/90/100	12.0	2.4/2.3/-	2.9/2.8/2.3	11.7/10.9/7.9	-/11.8/10.9
ml_vww	99	652/703/-	518/546/703	102/112/158	-/101/112	12.2	2.5/2.3/-	3.0/2.9/2.3	12.0/11.1/8.3	-/12.0/11.1
ml_conv	18	124/126/-	117/122/126	19/20/43	-/20/20	11.9	2.2/2.2/-	2.4/2.3/2.2	11.5/11.2/5.6	-/11.2/11.2
ml_pooling	24	134/136/-	120/125/136	24/24/31	-/24/24	9.6	2.3/2.3/-	2.5/2.4/2.3	9.6/9.6/7.6	-/9.6/9.6
ml_relu	3	3/5/-	3/3/5	3/3/3	-/3/3	11.7	11.7/7.8/-	11.7/11.7/7.8	11.7/11.7/11.7	-/11.7/11.7
AVG	**53**	**361/383/-**	**278/297/383**	**55/62/107**	**-/55/62**	**16.7**	**2.9/2.5/-**	**4.0/3.6/2.5**	**16.3/13.8/8.9**	**-/16.3/13.8**

differ in layer shapes and input sizes. In the 1 SM configuration, without the SAT constraint, the compiler generates an average of 53 regions per workload, with an average region size of 16.7 instructions. The 1000 µF entry is omitted at 1 SM because the partitioning has already converged at 500 µF, where the SAT budget is loose enough that relaxing it further leaves the region count and size essentially unchanged. As the capacitor shrinks to 500 µF, 100 µF, and 50 µF, the region count increases to 55, 278, and 361, while the average region size decreases to 16.3, 4.0, and 2.9, respectively. This trend aligns directly with the SAT constraint: a smaller capacitor lowers the number of instructions that a region can safely execute.

Also, increasing the SM count tightens the SAT budget in the same manner as shrinking the capacitor, since the same stored energy now supplies more active SMs simultaneously and drains faster, reducing the safe execution time available to each SM. This trend holds across all capacitor sizes in Table III: the region count rises and the average region size falls as the number of SMs increases. Doubling the SM count has the same effect as halving the capacitance, so some entries repeat across the table. Both 1000 µF on 4 SMs and 500 µF on 2 SMs yield 62 regions, and both 100 µF on 4 SMs and 50 µF on 2 SMs yield 383. The 50 µF configuration on 4 SMs, the strictest setting, is omitted because its SAT budget falls below the minimum viable region size.

Figure 13 shows the instruction breakdown between register-checkpoint stores and non-checkpoint instructions. By adding region boundaries, SAT raises the checkpoint-store fraction from 28.0% to 44.9% on average, yet non-checkpoint instructions remain the majority of the stream at 55.1%. Capacitance-aware partitioning prevents stagnation while keeping checkpoint stores below half of the executed instructions; this translates to only 0.7–1.6% of the power-on period being spent on re-execution (Section VI-C3). This also justifies enforcing SAT in a best-effort manner rather than as a strict guarantee, which would require scaling Equation 3 by the maximum warp occupancy of an SM to cover the worst case across all co-scheduled warps; the resulting excessively short regions would generate far more checkpoints, inflating both the checkpoint-store fraction and run-time overhead.

3) Performance Analysis with Power Failure: Figure 14 presents the performance of iGPU under intermittent power, accounting for both normal execution during power-on periods and recharge time during power-off periods. Overall, iGPU consistently outperforms jGPU under the default setting across both non-ML and ML workloads. On the non-ML workloads, iGPU achieves an average speedup of 8.51× (1 SM) to 10.17× (4 SMs) over the inorder NVP baseline, while jGPU achieves 3.99×; on the ML workloads, iGPU achieves 7.00× to 7.59×, while jGPU achieves 2.61×. The key reason is that jGPU must checkpoint the full volatile SM state upon imminent power failure, including the massive GPU RF, whereas iGPU avoids such JIT checkpointing and instead recovers by re-executing only the interrupted region, allowing more harvested energy to be spent on useful forward progress. Although adding SMs to jGPU could narrow this gap, JIT checkpointing the RF of every SM would drive V_{Backup} to impractical levels (Table I).

Figure 14 also shows that iGPU outperforms the CIM baseline. Although MOUSE benefits from low backup overhead and efficient in-memory execution [42], its best configuration (16 tiles) reaches an average speedup of only 4.50×, compared with 7.00–7.59× for iGPU, and iGPU matches or exceeds MOUSE's performance on every ML workload.

Another interesting observation from Figure 14 is that iGPU's benefit from additional SMs does not scale proportionally with the SM count. Moving from 1 to 2 SMs improves the non-ML and ML geometric means by 9.9% and 7.9%, respectively. Doubling the SM count again to 4 SMs adds 8.8% on the non-ML workloads yet only 0.5% on the ML ones, implying that the second doubling provides almost no additional benefit for ML workloads. Notably, none of these gains approaches the 73.2% and 48.4% obtained from 1 to 2 SMs, or the 65.9% and 33.1% from 2 to 4 SMs, under continuous power (Figure 12). Thus, added SMs offer limited

benefit under intermittent power for either workload type. This is because most of the elapsed time under power failure, *i.e.,* the sum of power-on and power-off periods, is spent recharging the capacitor. Adding SMs accelerates execution, but the time saved is small relative to this dominant recharge time. The total elapsed time thus shrinks only slightly, which limits the speedup from additional SMs. Since scaling to more SMs replicates an entire SM along with iGPU's extended PDOM stack, the 1 SM configuration is the most cost-effective, retaining most of the benefit at the smallest area cost.

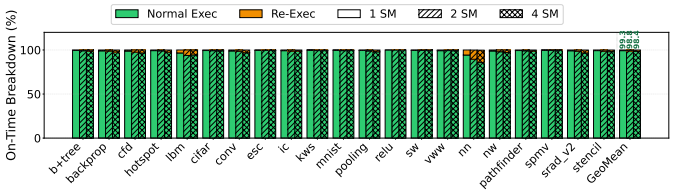


Fig. 15: Breakdown of power-on period

Beyond this scaling behavior, Figure 15 shows how iGPU's power-on time is spent, breaking it into normal execution and re-execution, where re-execution covers the recovery code and the replay of the interrupted region; to be conservative, re-execution is not considered complete while any warp is still recovering or replaying. Across the evaluated workloads, normal execution dominates, averaging 99.3%, 98.8%, and 98.4% of the power-on period at 1, 2, and 4 SMs, respectively, and leaving only 0.7–1.6% for re-execution. Thus, region re-execution, while essential for correct recovery, never compromises the performance advantage of the GPU. This fraction grows slightly with SM count even though re-execution time itself does not grow; each outage costs slightly more because more warps must be recovered, but they recover concurrently across the added SMs, potentially resulting in fewer outages due to energy savings from *race-to-halt* [104] execution. As a result, the overall power-on period shrinks faster than the re-execution time, a trend that Figure 17 confirms.

4) Sensitivity Analysis Varying Power Trace and Capacitor Size: Figure 16 summarizes the performance trends across four power traces and four capacitor sizes. Across all settings, iGPU stays competitive with the CIM baseline and consistently outperforms jGPU, showing that the benefit of avoiding a JIT checkpoint of the entire volatile SM state extends well beyond the default solar, 100 µF setting. The gaps among architectures, however, narrow as the capacitor size grows, with the highest normalized speedup falling from 10.60× at 50 µF to 1.88× at 1000 µF. A larger capacitor stores more energy, so every architecture completes more work before each outage and incurs its backup or recovery overhead less often.

Moreover, recharge time grows with the capacitor size, whereas the power-on time is bounded by the workload itself. Power-on time thus accounts for a shrinking fraction of total elapsed time, falling by more than 10× from 0.0138% at 50 µF to 0.0012% at 1000 µF under the solar trace at 1 SM. This longer recharge time applies to iGPU, jGPU, and CPU and

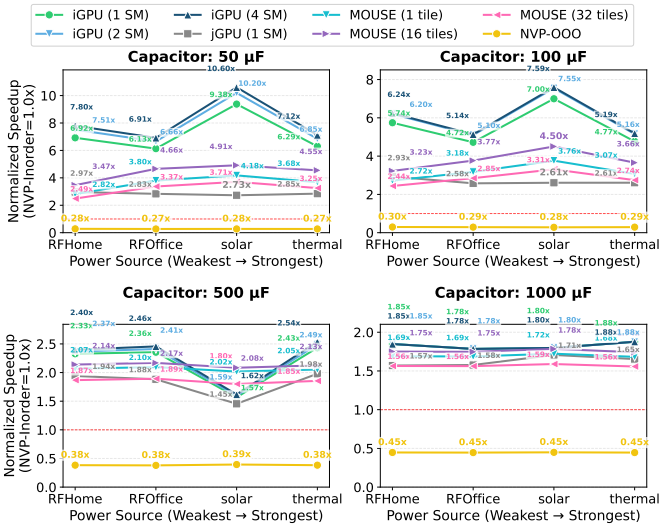


Fig. 16: Overall trend of speedups over inorder NVP

CIM baselines alike, further narrowing their speedup gaps.

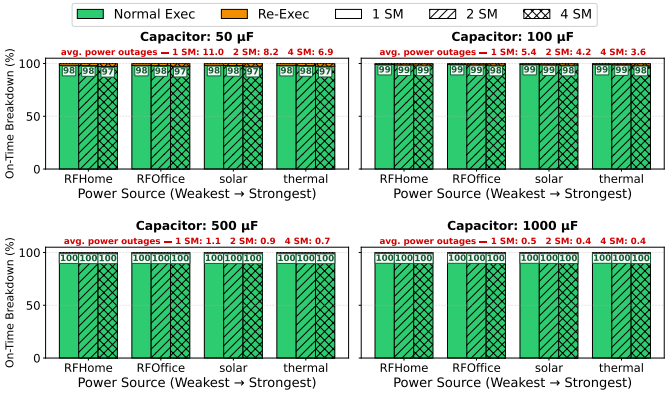


Fig. 17: Overall trend of power-on period breakdown

Figure 17 then decomposes the power-on period into normal execution and re-execution, along with the average number of power outages across four power traces. The larger energy budget makes power outages rarer, with the average number across the 4 traces falling from 11.0, 8.2, and 6.9 at 50 µF to 0.5, 0.4, and 0.4 at 1000 µF for 1, 2, and 4 SMs, respectively. Re-execution shrinks accordingly, occupying about 2–3% of iGPU's power-on time at 50 µF, 1–2% at 100 µF, and roughly 1% or less at 500 µF and 1000 µF. Adding SMs also reduces outages with the help of *race-to-halt* [104] energy savings, *i.e.,* it accelerates execution and allows each application to finish in fewer power cycles despite the higher power consumption. Across every capacitor size and power source, however, the re-execution fraction grows slightly with the SM count, as accelerated execution shortens the power-on period more than reduced outages shorten re-execution time, rather than more SMs incurring additional re-execution.

5) Hardware Cost Analysis: Both iGPU and jGPU require additional hardware. Based on CACTI 7.0 [105] at 22 nm,

jGPU incurs much higher hardware overhead, as it relies on full JIT checkpointing of volatile SM state. The GPU RF dominates this overhead, requiring a 256 KB NVFF per SM (0.18 mm^2). jGPU must also preserve the OCB (48 KB) and the standard PDOM stack (40 KB), which are checkpointed to NVM on power failure, bringing the volatile state saved per SM to 344 KB while leaving the added area at approximately 0.18 mm^2. The RF NVFF also lies on the critical path of SM backup/restore, with an access cost of 0.43 ns/47.8 pJ.

In contrast, iGPU requires only lightweight support for divergence-aware recovery. The added *RecoveryPC* field per PDOM stack entry, the 1-bit volatile *RecoveredBit*, and the per-warp TOR pointer extend the live PDOM stack by 16 KB per SM. Under the worst-case provisioning in Section VI-A, the full extended stack totals 56 KB per SM, checkpointed on power failure as in QuickRecall [71]. This adds only 0.012 mm^2 of SRAM per SM, roughly $15\times$ smaller than jGPU's added hardware area. The extension is also fast and energy-efficient, requiring only 0.18 ns/3.9 pJ for the 16 KB SRAM access, while the 56 KB stack checkpoint is written to NVM only upon power failure, off the SM datapath, and therefore never affects normal execution.

VII. Discussion

Applicability: iGPU is not tied to a specific AI model. As long as a kernel executes through the regular SM datapath at the warp level, it can be compiled by the workflow in Figure 5, whether it implements an attention or recurrent layer.

Tensor Core: It is outside iGPU's scope, as it bypasses the standard warp-level SM datapath; its operands are delivered through specialized datapaths, and the matrix operation proceeds inside the accelerator. Re-executing the region does not by itself restore the Tensor Core's microarchitectural state of the in-flight matrix operation, so a power outage during such an operation cannot be handled by iGPU's per-warp idempotent recovery. Supporting Tensor Cores would therefore require a separate mechanism to preserve or reconstruct this state, *e.g.,* jGPU-style JIT checkpointing, whose evaluation in turn demands an accurate Tensor Core power model that our methodology (Section VI-B) does not yet provide.

VIII. Other Related Work

A. Intermittent Inference and Accelerators

On MSP430 [35], intermittent DNN inference has been accelerated with LEA [34], and subsequent work has further improved forward progress across outages while reducing inference-state overhead [36]–[38], [106]. Beyond these designs, ultra-low-power accelerators for constrained edge platforms include programmable CGRAs, dataflow architectures, and related reconfigurable designs [107]–[111]. While these improve the efficiency and programmability of data-parallel execution, they largely target continuous operation rather than intermittent computation. Targeting EHSs directly, volatile FPGAs have been studied as intermittent accelerators [112], Bobber [113] provides a prototyping platform, and Samoyed [114] enables safe peripheral and accelerator execution.

B. Intermittent Computing Support for CPUs

Idempotent processing [57] has been widely explored by compiler-based approaches as a roll-back recovery solution for intermittent computing on CPU-based EHSs. Prior work forms idempotent regions, checkpoints only minimal volatile state at region boundaries, and reduces re-execution cost via instruction reordering and loop optimizations [46], [115]. Other approaches reduce recovery overhead through task-based execution, compiler-directed control of re-execution, and dynamic selection between roll-back and roll-forward recovery [48], [52], [67], [80], [116]–[118]. Recent work also addresses volatile state management through cache and memory mechanisms [51], [86], [119]–[125], as well as conventional memory hierarchy optimizations [126]–[128].

C. Roll-back Recovery for GPUs

Unlike the EHS works above, prior GPU recovery mechanisms [59], [78], [129] assume that power is never lost, localizing a fault to the affected warp when it is triggered either by an exception [59] or by data corruption induced by soft errors [130]–[134]. Since recovery is confined to that warp's current region, none of these mechanisms reconstructs all threads' state in a warp or preserves metadata for divergent paths already completed. A power outage, in contrast, interrupts all warps at once, including inactive divergent subgroups and unscheduled warps, requiring iGPU to reconstruct the state of every thread. This requirement introduces the divergence challenges described in Sections IV-D1 through IV-D3, because subgroups may have progressed along different control-flow paths before the outage. Also, iGPU must guarantee crash consistency and region-level persistence—which a continuously powered GPU never requires.

IX. Conclusion

This paper presents iGPU, a novel compiler/architecture co-design that enables GPUs for energy harvesting systems. We find that naively applying prior JIT checkpointing to GPUs is impractical due to the excessive hardware and energy overhead of saving the GPU register file. Instead, iGPU leverages both control-divergence- and capacitance-aware idempotent processing to restore volatile registers of warps by restarting power-interrupted regions in a failure-atomic manner. By extending the standard post-dominator stack, iGPU correctly recovers all threads even in the presence of control divergence in a warp. Under the default setting, iGPU achieves average speedups of 2.38–2.71$\times$ over jGPU, 7.76–8.85$\times$ over the CPU baseline, and 1.56–1.69$\times$ over the CIM baseline on ML workloads. We believe that iGPU offers foundational compiler and architectural support for GPU-based EHSs, laying the groundwork for future research in this emerging area.

X. Acknowledgements

We appreciate anonymous reviewers for their invaluable comments. This work is in part supported by the NSF grants CNS-2314681, CNS-2314680, CNS-2135157, CNS-2214980, CCF-2153747, CCF-2153748, and CCF-2153749.

REFERENCES

[1] S. Ahmed, B. Islam, K. S. Yildirim, M. Zimmerling, P. Pawełczak, M. H. Alizai, B. Lucia, L. Mottola, J. Sorber, and J. Hester, "The Internet of batteryless things," *Communications of the ACM*, vol. 67, no. 3, pp. 64–73, 2024.

[2] S. Naifar, O. Kanoun, and C. Trigona, "Energy harvesting technologies and applications for the Internet of Things and wireless sensor networks," *Sensors*, vol. 24, no. 14, p. 4688, 2024.

[3] S. Fu, V. Narayanan, M. L. Wymore, V. Deep, H. Duwe, and D. Qiao, "No battery, no problem: Challenges and opportunities in batteryless intermittent networks," *Journal of Communications and Networks*, vol. 25, no. 6, pp. 806–813, 2023.

[4] K. Geissdoerfer and M. Zimmerling, "Riotee: An open-source hardware and software platform for the battery-free Internet of Things," in *Proceedings of the 22nd ACM Conference on Embedded Networked Sensor Systems*. ACM, 2024, pp. 198–210.

[5] T. Gregersen, T. A. Wild, L. W. Havmøller, P. R. Møller, T. A. Lenau, M. Wikelski, and R. W. Havmøller, "A novel kinetic energy harvesting system for lifetime deployments of wildlife trackers," *PLoS One*, vol. 18, no. 5, p. e0285930, 2023.

[6] E. Bäumker, L. Conrad, L. M. Comella, and P. Woias, "A fully featured thermal energy harvesting tracker for wildlife," *Energies*, vol. 14, no. 19, p. 6363, 2021.

[7] S. Nasser, H. Duarte Moura, D. Subotic, R. K. Singh, M. Weyn, and J. Famaey, "Feasibility of energy-neutral wildlife tracking using multi-source energy harvesting," in *Proceedings of the 2025 International Conference on Information Technology for Social Good*, 2025, pp. 344–352.

[8] Y. Sun, T. Tan, and Z. Yan, "Smart ocean powering and sensing via mechanical energy harvesting: Methods, advances, and challenges," *Nano Energy*, vol. 141, p. 111128, 2025.

[9] Z.-Q. Lu, L. Zhao, H.-L. Fu, E. Yeatman, H. Ding, and L.-Q. Chen, "Ocean wave energy harvesting with high energy density and self-powered monitoring system," *Nature Communications*, vol. 15, no. 1, p. 6513, 2024.

[10] Y. Peng, H. Tang, H. Pan, Z. Zhang, D. Luo, M. Tang, W. Kong, Y. Li, G. Liu, and Y. Hu, "A self-powered and self-monitoring ultra-low frequency wave energy harvester for smart ocean ranches," *iScience*, vol. 27, no. 9, p. 110665, 2024.

[11] S. Pathak, D. V. S. Reddy, A. K. Gautam, and T. Goel, "A dual-source energy harvester for battery-less agricultural IoT sensor networks," *AEU – International Journal of Electronics and Communications*, p. 156243, 2026.

[12] S. Khernane, S. Bouam, and C. Arar, "Renewable energy harvesting for wireless sensor networks in precision agriculture," *International Journal of Networked and Distributed Computing*, vol. 12, no. 1, pp. 8–16, 2024.

[13] P. Xiang, M. Xue, X. Chen, F. Liu, X. Wu, S. Wang, W. Zhang, and X. Xiao, "Self-powered flexible wireless sensing for smart farming machinery," *Nano Energy*, p. 111663, 2025.

[14] H. R. Mendis, K. S. Yildirim, M. Zimmerling, L. Mottola, and P.-C. Hsiu, "Special session—intermittent TinyML: Powering sustainable deep intelligence without batteries," in *Proceedings of the International Conference on Embedded Software*, 2025, pp. 13–22.

[15] S. Lee, B. Islam, Y. Luo, and S. Nirjon, "Intermittent learning: On-device machine learning on intermittently powered systems," *Proceedings of the ACM on Interactive, Mobile, Wearable and Ubiquitous Technologies*, vol. 3, no. 4, pp. 1–30, 2019.

[16] R. David, J. Duke, A. Jain, V. Janapa Reddi, N. Jeffries, J. Li, N. Kreeger, I. Nappier, M. Natraj, T. Wang *et al.*, "TensorFlow Lite Micro: Embedded machine learning for TinyML systems," *Proceedings of Machine Learning and Systems*, vol. 3, pp. 800–811, 2021.

[17] L. Capogrosso, F. Cunico, D. S. Cheng, F. Fummi, and M. Cristani, "A machine learning-oriented survey on tiny machine learning," *IEEE Access*, vol. 12, pp. 23 406–23 426, 2024.

[18] J. Lin, W.-M. Chen, Y. Lin, C. Gan, S. Han *et al.*, "MCUNet: Tiny deep learning on IoT devices," *Advances in Neural Information Processing Systems*, vol. 33, pp. 11 711–11 722, 2020.

[19] Y. Zhang, S.-Y. Huang, B. Min, A. I. Park, G. Fang, J. Choi, and C. Jung, "Achieving speculative intermittent computation on commodity energy harvesting hardware," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 2026.

[20] J. Bito, R. Bahr, J. G. Hester, S. A. Nauroze, A. Georgiadis, and M. M. Tentzeris, "A novel solar and electromagnetic energy harvesting system with a 3-D printed package for energy-efficient Internet-of-Things wireless sensors," *IEEE Transactions on Microwave Theory and Techniques*, vol. 65, no. 5, pp. 1831–1842, 2017.

[21] H. H. R. Sherazi, D. Zorbas, and B. O'Flynn, "A comprehensive survey on RF energy harvesting: Applications and performance determinants," *Sensors*, vol. 22, no. 8, p. 2990, 2022.

[22] A. P. Sample, D. J. Yeager, P. S. Powledge, A. V. Mamishev, and J. R. Smith, "Design of an RFID-based battery-free programmable sensing platform," *IEEE Transactions on Instrumentation and Measurement*, vol. 57, no. 11, pp. 2608–2615, 2008.

[23] F. Ghasemi, L. Liedtke, and M. Jahre, "PES: An energy and throughput model for energy harvesting IoT systems," in *2023 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*. IEEE, 2023, pp. 13–23.

[24] R. Barjami, A. Miele, and L. Mottola, "Intermittent inference: Trading a 1% accuracy loss for a 1.9× throughput speedup," in *Proceedings of the 22nd ACM Conference on Embedded Networked Sensor Systems*, 2024, pp. 647–660.

[25] R. Sahu, V. Deep, and H. Duwe, "HANNA: Harvesting-aware neural network architecture search for batteryless intermittent devices," in *2024 IEEE International Performance, Computing, and Communications Conference (IPCCC)*. IEEE, 2024, pp. 1–10.

[26] J. Choi, J. Park, N. L'Heureux, Y. Solihin, H. Joe, and C. Jung, "Intermittence-aware speculative page coloring for secure nvm," in *2026 ACM/IEEE 53rd Annual International Symposium on Computer Architecture (ISCA)*. IEEE, 2026, pp. 1134–1148.

[27] Y. Liu, Z. Li, H. Li, Y. Wang, X. Li, K. Ma, S. Li, M.-F. Chang, J. Sampson, Y. Xie *et al.*, "Ambient energy harvesting nonvolatile processors: From circuit to system," in *Proceedings of the 52nd Annual Design Automation Conference*, 2015, pp. 1–6.

[28] K. Ma, X. Li, S. Li, Y. Liu, J. Sampson, Y. Xie, and V. Narayanan, "Nonvolatile processor architecture exploration for energy-harvesting applications," *IEEE Micro*, vol. 35, no. 5, pp. 32–40, 2015.

[29] F. Su, Y. Liu, Y. Wang, and H. Yang, "A ferroelectric nonvolatile processor with 46 μs system-level wake-up time and 14 μs sleep time for energy harvesting applications," *IEEE Transactions on Circuits and Systems I: Regular Papers*, vol. 64, no. 3, pp. 596–607, 2017.

[30] Y. Liu, F. Su, Y. Yang, Z. Wang, Y. Wang, Z. Li, X. Li, R. Yoshimura, T. Naiki, T. Tsuwa *et al.*, "A 130-nm ferroelectric nonvolatile system-on-chip with direct peripheral restore architecture for transient computing systems," *IEEE Journal of Solid-State Circuits*, vol. 54, no. 3, pp. 885–895, 2019.

[31] Z. Wang, Y. Liu, A. Lee, F. Su, C.-P. Lo, Z. Yuan, J. Li, C.-C. Lin, W.-H. Chen, H.-Y. Chiu *et al.*, "A 65-nm ReRAM-enabled nonvolatile processor with time-space domain adaption and self-write-termination achieving >4× faster clock frequency and >6× higher restore speed," *IEEE Journal of Solid-State Circuits*, vol. 52, no. 10, pp. 2769–2785, 2017.

[32] Q. Liu and C. Jung, "Lightweight hardware support for transparent consistency-aware checkpointing in intermittent energy-harvesting systems," in *2016 5th Non-Volatile Memory Systems and Applications Symposium (NVMSA)*, 2016, pp. 1–6.

[33] G. Fang and C. Jung, "Speculation under intermittence: A guide for computer architects," *Computer*, vol. 59, no. 8, pp. 141–146, 2026.

[34] Texas Instruments. (2016) Low-energy accelerator (LEA) frequently asked questions (FAQ). Texas Instruments. Accessed: 2026-02-06. [Online]. Available: https://www.ti.com/lit/an/slaa720/slaa720.pdf

[35] Texas Instruments, "MSP430FR599x, MSP430FR596x mixed-signal microcontrollers," Texas Instruments, Tech. Rep. SLASE54D, Jan. 2021, rev. D; accessed 2026-02-06. [Online]. Available: https://www.ti.com/lit/ds/symlink/msp430fr5994.pdf

[36] G. Gobieski, B. Lucia, and N. Beckmann, "Intelligence beyond the edge: Inference on intermittent embedded systems," in *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*, 2019, pp. 199–213.

[37] C.-K. Kang, H. R. Mendis, C.-H. Lin, M.-S. Chen, and P.-C. Hsiu, "More is less: Model augmentation for intermittent deep inference," *ACM Transactions on Embedded Computing Systems*, vol. 21, no. 5, pp. 1–26, 2022.

[38] C.-H. Yen, H. R. Mendis, T.-W. Kuo, and P.-C. Hsiu, "Stateful neural networks for intermittent systems," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 41, no. 11, pp. 4229–4240, 2022.

[39] K. Akhunov, K. S. Yildirim, J. Choi, and C. Jung, "Adaptive computing in memory meets conventional batteryless platforms," *ACM Transactions on Embedded Computing Systems*, vol. 24, no. 6, pp. 1–26, 2025.

[40] K. Akhunov and K. S. Yildirim, "LUTIC: A CRAM-based architecture for power-failure-resilient in-memory computing," in *2023 26th International Symposium on Design and Diagnostics of Electronic Circuits and Systems (DDECS)*. IEEE, 2023, pp. 69–72.

[41] K. Akhunov and K. S. Yildirim, "CRAM-based acceleration for intermittent computing of parallelizable tasks," *IEEE Transactions on Emerging Topics in Computing*, vol. 12, no. 1, pp. 48–59, 2023.

[42] S. Resch, S. K. Khatamifard, Z. I. Chowdhury, M. Zabihi, Z. Zhao, H. Cilasun, J.-P. Wang, S. S. Sapatnekar, and U. R. Karpuzcu, "MOUSE: Inference in nonvolatile memory for energy harvesting applications," in *2020 53rd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*. IEEE, 2020, pp. 400–414.

[43] A. Das, Y. Agarwal, C. Wang, A. Raha, S. K. Thirumala, S. K. Gupta, and V. Raghunathan, "PRISM: Toward energy-efficient AIoT with processing-in-sensor+memory in battery-free devices," in *2025 IEEE International Conference on Omni-layer Intelligent Systems (COINS)*. IEEE, 2025, pp. 1–6.

[44] S. Resch, S. K. Khatamifard, Z. I. Chowdhury, M. Zabihi, Z. Zhao, H. Cilasun, J.-P. Wang, S. S. Sapatnekar, and U. R. Karpuzcu, "Energy-efficient and reliable inference in nonvolatile memory under extreme operating conditions," *ACM Transactions on Embedded Computing Systems*, vol. 21, no. 5, pp. 1–36, 2022.

[45] K. Akhunov, E. Yildiz, and K. S. Yildirim, "Enabling efficient intermittent computing on brand new microcontrollers via tracking programmable voltage thresholds," in *Proceedings of the 11th International Workshop on Energy Harvesting & Energy-Neutral Sensing Systems*. ACM, 2023, pp. 16–22.

[46] J. Van Der Woude and M. Hicks, "Intermittent computation without hardware support or programmer intervention," in *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16)*, 2016, pp. 17–32.

[47] D. Balsamo, A. S. Weddell, G. V. Merrett, B. M. Al-Hashimi, D. Brunelli, and L. Benini, "Hibernus: Sustaining computation during intermittent supply for energy-harvesting systems," *IEEE Embedded Systems Letters*, vol. 7, no. 1, pp. 15–18, 2015.

[48] K. Maeng, A. Colin, and B. Lucia, "Alpaca: Intermittent execution without checkpoints," *Proceedings of the ACM on Programming Languages*, vol. 1, no. OOPSLA, Oct. 2017.

[49] K. Maeng and B. Lucia, "Adaptive dynamic checkpointing for safe efficient intermittent computing," in *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18)*. Carlsbad, CA: USENIX Association, Oct. 2018, pp. 129–144. [Online]. Available: https://www.usenix.org/conference/osdi18/presentation/maeng

[50] S. S. Baghsorkhi and C. Margiolas, "Automating efficient variable-grained resiliency for low-power IoT systems," in *Proceedings of the 2018 International Symposium on Code Generation and Optimization*, 2018, pp. 38–49.

[51] J. Zeng, J. Choi, X. Fu, A. P. Shreepathi, D. Lee, C. Min, and C. Jung, "ReplayCache: Enabling volatile caches for energy harvesting systems," in *MICRO-54: 54th Annual IEEE/ACM International Symposium on Microarchitecture*, 2021, pp. 170–182.

[52] J. Choi, L. Kittinger, Q. Liu, and C. Jung, "Compiler-directed high-performance intermittent computation with power failure immunity," in *2022 IEEE 28th Real-Time and Embedded Technology and Applications Symposium (RTAS)*, 2022, pp. 40–54.

[53] J. Jeong and C. Jung, "PMEM-spec: Persistent memory speculation (strict persistency can trump relaxed persistency)," in *Proceedings of the 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*, 2021, pp. 517–529.

[54] J. Jeong, J. Hong, S. Maeng, C. Jung, and Y. Kwon, "Unbounded hardware transactional memory for a hybrid dram/nvm memory system," in *2020 53rd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, 2020.

[55] Y. Han, Z. Hu, J. Choi, K. A. Zubair, A. Awad, C. Jung, and B. B. Kang, "A novel efficient crash consistency solution enabling rollback recovery for secure nvm in low-power energy harvesting systems," *IEEE Transactions on Dependable and Secure Computing*, vol. 22, no. 3, pp. 2179–2196, 2024.

[56] M. A. de Kruijf, "Compiler construction of idempotent regions and applications in architecture design," Ph.D. dissertation, University of Wisconsin–Madison, 2012.

[57] M. de Kruijf and K. Sankaralingam, "Idempotent code generation: Implementation, analysis, and evaluation," in *Proceedings of the 2013 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*. IEEE, 2013, pp. 1–12.

[58] Q. Liu, C. Jung, D. Lee, and D. Tiwari, "Compiler-directed soft error detection and recovery to avoid DUE and SDC via Tail-DMR," *ACM Transactions on Embedded Computing Systems*, vol. 16, no. 2, pp. 1–26, 2016.

[59] J. Menon, M. de Kruijf, and K. Sankaralingam, "iGPU: Exception support and speculative execution on GPUs," *ACM SIGARCH Computer Architecture News*, vol. 40, no. 3, pp. 72–83, 2012.

[60] S. Feng, S. Gupta, A. Ansari, S. A. Mahlke, and D. I. August, "Encore: Low-cost, fine-grained transient fault recovery," in *Proceedings of the 44th Annual IEEE/ACM International Symposium on Microarchitecture*, 2011, pp. 398–409.

[61] Q. Liu, C. Jung, D. Lee, and D. Tiwari, "Clover: Compiler-directed lightweight soft error resilience," *ACM SIGPLAN Notices*, vol. 50, no. 5, pp. 1–10, 2015.

[62] Q. Liu, C. Jung, D. Lee, and D. Tiwari, "Compiler-directed lightweight checkpointing for fine-grained guaranteed soft error recovery," in *SC'16: Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*. IEEE, 2016, pp. 228–239.

[63] Q. Liu, J. Izraelevitz, S. K. Lee, M. L. Scott, S. H. Noh, and C. Jung, "iDO: Compiler-directed failure atomicity for nonvolatile memory," in *2018 51st Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*. IEEE, 2018, pp. 258–270.

[64] M. de Kruijf, K. Sankaralingam, and S. Jha, "Static analysis and compiler design for idempotent processing," in *Proceedings of the 33rd ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, 2012, pp. 475–486.

[65] S. Muchnick, *Advanced Compiler Design and Implementation*. Morgan Kaufmann, 1997.

[66] W. W. L. Fung, I. Sham, G. Yuan, and T. M. Aamodt, "Dynamic warp formation and scheduling for efficient GPU control flow," in *40th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO 2007)*. IEEE, 2007, pp. 407–420.

[67] G. Fang, J. Choi, and C. Jung, "Hybrid power failure recovery for intermittent computing," in *Proceedings of the 43rd IEEE/ACM International Conference on Computer-Aided Design*, 2024, pp. 1–9.

[68] J. Choi, H. Joe, and C. Jung, "CapOS: Capacitor error resilience for energy harvesting systems," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 41, no. 11, pp. 4539–4550, 2022.

[69] J. Choi, J. Choi, H. Joe, and C. Jung, "Caphammer: Exploiting capacitor vulnerability of energy harvesting systems," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 2024.

[70] J. Choi, H. Joe, C. Jung, and J. Choi, "Defending against emi attacks on just-in-time checkpoint for resilient intermittent systems," in *57th International Symposium on Microarchitecture (MICRO)*, 2024.

[71] H. Jayakumar, A. Raha, and V. Raghunathan, "QuickRecall: A low overhead HW/SW approach for enabling computations across power cycles in transiently powered computers," in *2014 27th International Conference on VLSI Design and 2014 13th International Conference on Embedded Systems*, 2014, pp. 330–335.

[72] A. A. Saki, S. H. Lin, M. Alam, S. K. Thirumala, S. K. Gupta, and S. Ghosh, "A family of compact nonvolatile flip-flops with ferroelectric FET," *IEEE Transactions on Circuits and Systems I: Regular Papers*, vol. 66, no. 11, pp. 4219–4229, 2019.

[73] X. Li, S. George, Y. Liang, K. Ma, K. Ni, A. Aziz, S. K. Gupta, J. Sampson, M.-F. Chang, Y. Liu *et al.*, "Lowering area overheads for FeFET-based energy-efficient nonvolatile flip-flops," *IEEE Transactions on Electron Devices*, vol. 65, no. 6, pp. 2670–2674, 2018.

[74] Y. Gu, Y. Liu, Y. Wang, H. Li, and H. Yang, "NVPsim: A simulator for architecture exploration of nonvolatile processors," in *2016 21st Asia and South Pacific Design Automation Conference (ASP-DAC)*. IEEE, 2016, pp. 147–152.

[75] W. Song, Y. Zhou, M. Zhao, L. Ju, C. J. Xue, and Z. Jia, "EMC: Energy-aware morphable cache design for nonvolatile processors," *IEEE Transactions on Computers*, vol. 68, no. 4, pp. 498–509, 2018.

[76] M. Xie, M. Zhao, C. Pan, H. Li, Y. Liu, Y. Zhang, C. J. Xue, and J. Hu, "Checkpoint-aware hybrid cache architecture for NV processors in energy harvesting powered systems," in *Proceedings of the Eleventh IEEE/ACM/IFIP International Conference on Hardware/Software Codesign and System Synthesis*, 2016, pp. 1–10.

[77] H. Williams, M. Moukarzel, and M. Hicks, "Failure sentinels: Ubiquitous just-in-time intermittent computation via low-cost hardware support for voltage monitoring," in *2021 ACM/IEEE 48th Annual International Symposium on Computer Architecture (ISCA)*. IEEE, 2021, pp. 665–678.

[78] H. Kim, J. Zeng, Q. Liu, M. Abdel-Majeed, J. Lee, and C. Jung, "Compiler-directed soft error resilience for lightweight GPU register file protection," in *Proceedings of the 41st ACM SIGPLAN Conference on Programming Language Design and Implementation*. ACM, 2020, pp. 989–1004.

[79] X. Xie, Y. Liang, X. Li, Y. Wu, G. Sun, T. Wang, D. Fan *et al.*, "CRAT: Enabling coordinated register allocation and thread-level parallelism optimization for GPUs," *IEEE Transactions on Computers*, vol. 67, no. 6, pp. 890–897, 2018.

[80] J. Choi, Q. Liu, and C. Jung, "CoSpec: Compiler-directed speculative intermittent computation," in *Proceedings of the 52nd Annual IEEE/ACM International Symposium on Microarchitecture*, 2019, pp. 399–412.

[81] J. Choi, H. Joe, Y. Kim, and C. Jung, "Achieving stagnation-free intermittent computation with boundary-free adaptive execution," in *2019 IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS)*, 2019, pp. 331–344.

[82] X. Dong, C. Xu, Y. Xie, and N. P. Jouppi, "NVSim: A circuit-level performance, energy, and area model for emerging nonvolatile memory," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 31, no. 7, pp. 994–1007, 2012.

[83] L. Jeanmougin, T. Carle, and C. Rochange, "Bounding the WCET of a GPU thread block with a multi-phase representation of warps execution," in *37th Euromicro Conference on Real-Time Systems (ECRTS 2025)*, vol. 335. Schloss Dagstuhl–Leibniz-Zentrum für Informatik, 2025, pp. 11:1–11:26.

[84] V. Hirvisalo, "On static timing analysis of GPU kernels," in *14th International Workshop on Worst-Case Execution Time Analysis (WCET)*. Schloss Dagstuhl–Leibniz-Zentrum für Informatik, 2014, pp. 43–52.

[85] J. Zeng, T. Zhang, and C. Jung, "Compiler-directed whole-system persistence," in *2024 ACM/IEEE 51st Annual International Symposium on Computer Architecture (ISCA)*. IEEE, 2024, pp. 961–977.

[86] Y. Zhou, J. Zeng, J. Jeong, J. Choi, and C. Jung, "SweepCache: Intermittence-aware cache on the cheap," in *Proceedings of the 56th Annual IEEE/ACM International Symposium on Microarchitecture*, 2023, pp. 1059–1074.

[87] P.-F. Chiu, M.-F. Chang, S.-S. Sheu, K.-F. Lin, P.-C. Chiang, C.-W. Wu, W.-P. Lin, C.-H. Lin, C.-C. Hsu, F. T. Chen, K.-L. Su, M.-J. Kao, and M.-J. Tsai, "A low store energy, low VDDmin, nonvolatile 8T2R SRAM with 3D stacked RRAM devices for low power mobile applications," in *2010 Symposium on VLSI Circuits*, 2010, pp. 229–230.

[88] D. Saida, S. Kashiwada, M. Yakabe, T. Daibou, N. Hase, M. Fukumoto, S. Miwa, Y. Suzuki, H. Noguchi, S. Fujita *et al.*, "Sub-3 ns pulse with sub-100 μa switching of 1x–2x nm perpendicular MTJ for high-performance embedded STT-MRAM toward sub-20 nm CMOS," in *2016 IEEE Symposium on VLSI Technology*. IEEE, 2016, pp. 1–2.

[89] M. Khairy, Z. Shen, T. M. Aamodt, and T. G. Rogers, "Accel-Sim: An extensible simulation framework for validated GPU modeling," in *2020 ACM/IEEE 47th Annual International Symposium on Computer Architecture (ISCA)*, 2020, pp. 473–486.

[90] NVIDIA, *Jetson Nano System-on-Module Data Sheet*, NVIDIA, Aug. 2022, accessed: 2025-10-17. [Online]. Available: https://openzeka.com/en/wp-content/uploads/2022/08/JetsonNano_DataSheet_DS09366001v1.1.pdf

[91] V. Kandiah, S. Peverelle, M. Khairy, J. Pan, A. Manjunath, T. G. Rogers, T. M. Aamodt, and N. Hardavellas, "AccelWattch: A power modeling framework for modern GPUs," in *MICRO-54: 54th Annual IEEE/ACM International Symposium on Microarchitecture*. ACM, 2021, pp. 738–753.

[92] J. Lowe-Power *et al.*, "The gem5 simulator: Version 20.0+," arXiv:2007.03152 [cs.AR], 2020. [Online]. Available: https://arxiv.org/abs/2007.03152

[93] K. Ma, Y. Zheng, S. Li, K. Swaminathan, X. Li, Y. Liu, J. Sampson, Y. Xie, and V. Narayanan, "Architecture exploration for ambient energy harvesting nonvolatile processors," in *2015 IEEE 21st International Symposium on High Performance Computer Architecture (HPCA)*. IEEE, 2015, pp. 526–537.

[94] F. A. Endo, D. Couroussé, and H.-P. Charles, "Micro-architectural simulation of in-order and out-of-order ARM microprocessors with gem5," in *2014 International Conference on Embedded Computer Systems: Architectures, Modeling, and Simulation (SAMOS XIV)*. IEEE, Nov. 2014.

[95] F. A. Endo, D. Couroussé, and H.-P. Charles, "Micro-architectural simulation of embedded core heterogeneity with gem5 and McPAT," in *Proceedings of the 2015 Workshop on Rapid Simulation and Performance Evaluation: Methods and Tools*. ACM, 2015.

[96] NVIDIA. (2019) tegrastats utility. NVIDIA. Accessed: 2025-04-27. [Online]. Available: https://docs.nvidia.com/drive/drive_os_5.1.6.1L/nvvib_docs/index.html#page/DRIVE_OS_Linux_SDK_Development_Guide/Utilities/util_tegrastats.html#

[97] NVIDIA. (2019) NVML API reference. NVIDIA. Accessed: 2025-04-27. [Online]. Available: https://docs.nvidia.com/deploy/nvml-api/nvml-api-reference.html

[98] NVIDIA. (2016) nvidia-smi: NVIDIA system management interface program. NVIDIA. Accessed: 2025-04-27. [Online]. Available: https://developer.download.nvidia.com/compute/DCGM/docs/nvidia-smi-367.38.pdf

[99] NVIDIA, *Jetson Nano Thermal Design Guide*, NVIDIA, accessed: 2026-04-07. [Online]. Available: https://developer.download.nvidia.com/assets/embedded/secure/jetson/Nano/docs/Jetson_Nano_Thermal_Design_Guide_TDG-09383-001_v1.6.pdf

[100] P. Virtanen, R. Gommers, T. E. Oliphant, M. Haberland, T. Reddy, D. Cournapeau, E. Burovski, P. Peterson, W. Weckesser, J. Bright *et al.*, "SciPy 1.0: Fundamental algorithms for scientific computing in Python," *Nature Methods*, vol. 17, no. 3, pp. 261–272, 2020.

[101] S. Che, M. Boyer, J. Meng, D. Tarjan, J. Sheaffer, S.-H. Lee, and K. Skadron, "Rodinia: A benchmark suite for heterogeneous computing," in *2009 IEEE International Symposium on Workload Characterization (IISWC)*. IEEE, 2009, pp. 44–54.

[102] J. A. Stratton, C. Rodrigues, I.-J. Sung, N. Obeid, L.-W. Chang, N. Anssari, G. D. Liu, and W.-m. W. Hwu, "Parboil: A revised benchmark suite for scientific and commercial throughput computing," Center for Reliable and High-Performance Computing, University of Illinois at Urbana-Champaign, Tech. Rep. IMPACT-12-01, Mar. 2012. [Online]. Available: http://impact.crhc.illinois.edu/Shared/Docs/impact-12-01.parboil.pdf

[103] V. J. Reddi, C. Cheng, D. Kanter, P. Mattson, G. Schmuelling, C.-J. Wu, B. Anderson, M. Breughe, M. Charlebois, W. Chou, R. Chukka, C. Coleman, S. Davis, P. Deng, G. Diamos, J. Duke, D. Fick, J. S. Gardner, I. Hubara, S. Idgunji, T. B. Jablin, J. Jiao, T. St. John, P. Kanwar, D. Lee, J. Liao, A. Lokhmotov, F. Massa, P. Meng, P. Micikevicius, C. Osborne, G. Pekhimenko, A. T. R. Rajan, D. Sequeira, A. Sirasao, F. Sun, H. Tang, M. Thomson, F. Wei, E. Wu, L. Xu, K. Yamada, B. Yu, G. Yuan, A. Zhong, P. Zhang, and Y. Zhou, "MLPerf inference benchmark," arXiv:1911.02549 [cs.LG], 2019. [Online]. Available: https://arxiv.org/abs/1911.02549

[104] R. Efraim, R. Ginosar, C. Weiser, and A. Mendelson, "Energy aware race to halt: A down to earth approach for platform energy management," *IEEE Computer Architecture Letters*, vol. 13, no. 1, pp. 25–28, 2012.

[105] R. Balasubramonian, A. B. Kahng, N. Muralimanohar, A. Shafiee, and V. Srinivas, "CACTI 7: New tools for interconnect exploration in innovative off-chip memories," *ACM Transactions on Architecture and Code Optimization*, vol. 14, no. 2, pp. 1–25, 2017.

[106] M. Xiang, P. M. Gholami, and H. Hoffmann, "Lupe: Integrating the top-down approach with DNN execution on ultra-low-power devices," in *Proceedings of the 23rd ACM Conference on Embedded Networked Sensor Systems*, 2025, pp. 357–370.

[107] G. Gobieski, A. Nagi, N. Serafin, M. M. Isgenc, N. Beckmann, and B. Lucia, "MANIC: A vector-dataflow architecture for ultra-low-power embedded systems," in *Proceedings of the 52nd Annual IEEE/ACM International Symposium on Microarchitecture*, 2019, pp. 670–684.

[108] G. Gobieski, A. O. Atli, K. Mai, B. Lucia, and N. Beckmann, "SNAFU: An ultra-low-power, energy-minimal CGRA-generation framework and architecture," in *2021 ACM/IEEE 48th Annual International Symposium on Computer Architecture (ISCA)*. IEEE, 2021, pp. 1027–1040.

[109] G. Gobieski, S. Ghosh, M. Heule, T. Mowry, T. Nowatzki, N. Beckmann, and B. Lucia, "RipTide: A programmable, energy-minimal dataflow compiler and architecture," in *2022 55th IEEE/ACM International Symposium on Microarchitecture (MICRO)*. IEEE, 2022, pp. 546–564.

[110] N. Serafin, S. Ghosh, H. Desai, N. Beckmann, and B. Lucia, "Pipestitch: An energy-minimal dataflow architecture with lightweight threads," in *Proceedings of the 56th Annual IEEE/ACM International Symposium on Microarchitecture*, 2023, pp. 1409–1422.

[111] B. de Bruin, K. Vadivel, M. Wijtvliet, P. Jääskeläinen, and H. Corporaal, "R-Blocks: An energy-efficient, flexible, and programmable CGRA," *ACM Transactions on Reconfigurable Technology and Systems*, vol. 17, no. 2, pp. 1–34, 2024.

[112] A. M. A. Babai and K. Inoue, "Exploring volatile FPGAs potential for accelerating energy-harvesting IoT applications," *IEEE Computer Architecture Letters*, vol. 24, no. 1, pp. 137–140, 2025.

[113] V. Narayanan, R. Sahu, J. Sun, and H. Duwe, "Bobber: A prototyping platform for batteryless intermittent accelerators," in *Proceedings of the 2023 ACM/SIGDA International Symposium on Field Programmable Gate Arrays*, 2023, pp. 221–228.

[114] K. Maeng and B. Lucia, "Supporting peripherals in intermittent systems with just-in-time checkpoints," in *Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation*, 2019, pp. 1101–1116.

[115] V. Kortbeek, S. Ghosh, J. Hester, S. Campanoni, and P. Pawełczak, "WARio: Efficient code generation for intermittent computing," in *Proceedings of the 43rd ACM SIGPLAN International Conference on Programming Language Design and Implementation*. ACM, 2022, pp. 777–791. [Online]. Available: https://doi.org/10.1145/3519939.3523454

[116] Y. Wu, B. Min, M. Ismail, W. Xiong, C. Jung, and D. Lee, "IntOS: Persistent embedded operating system and language support for multi-threaded intermittent computing," in *18th USENIX Symposium on Operating Systems Design and Implementation (OSDI 24)*. Santa Clara, CA: USENIX Association, Jul. 2024, pp. 425–443. [Online]. Available: https://www.usenix.org/conference/osdi24/presentation/wu-yilun

[117] Y. Wu and D. Lee, "Capsule: Efficient multi-threaded intermittent computing with replayable volatile stack," in *2027 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*, 2027.

[118] Y. Wu, "Embedded operating system and application design for intermittent computing with hybrid memory," Ph.D. dissertation, State University of New York at Stony Brook, 2025.

[119] J. Choi, J. Zeng, D. Lee, C. Min, and C. Jung, "Write-light cache for energy harvesting systems," in *Proceedings of the 50th Annual International Symposium on Computer Architecture*. ACM, 2023.

[120] A. Bhattacharyya, A. Somashekhar, and J. San Miguel, "NvMR: Non-volatile memory renaming for intermittent computing," in *Proceedings of the 49th Annual International Symposium on Computer Architecture*. ACM, 2022, pp. 1–13.

[121] J. Zeng, J. Jeong, and C. Jung, "Persistent processor architecture," in *Proceedings of the 56th Annual IEEE/ACM International Symposium on Microarchitecture*, 2023, pp. 1075–1091.

[122] Y. Zhou, J. Zeng, and C. Jung, "LightWSP: Whole-system persistence on the cheap," in *2024 57th IEEE/ACM International Symposium on Microarchitecture (MICRO)*. IEEE, 2024, pp. 215–230.

[123] J. Jeong, J. Zeng, and C. Jung, "Capri: Compiler and architecture support for whole-system persistence," in *Proceedings of the 31st International Symposium on High-Performance Parallel and Distributed Computing*, 2022, pp. 71–83.

[124] Y. Zhou, J. Zeng, and C. Jung, "Anchoring whole-system persistence and resilience in CXL," in *Proceedings of the 40th ACM International Conference on Supercomputing*, 2026, pp. 811–827.

[125] N. Hassan, B. Min, C. Jung, Y. Solihin, and J. Choi, "WarmCache: Exploiting STT-RAM cache for low-power intermittent systems," in *Proceedings of the 52nd Annual International Symposium on Computer Architecture*, 2025, pp. 586–600.

[126] G. Fang and C. Jung, "Rethinking dead block prediction for intermittent computing," in *2025 IEEE International Symposium on High Performance Computer Architecture (HPCA)*. IEEE, 2025, pp. 732–744.

[127] G. Fang, J. Zeng, A. Gupta, and C. Jung, "Rethinking prefetching for intermittent computing," in *Proceedings of the 52nd Annual International Symposium on Computer Architecture*, 2025, pp. 225–240.

[128] G. Fang, J. Zeng, Y. Zhou, and C. Jung, "Intermittence-aware cache compression," in *2026 IEEE International Symposium on High Performance Computer Architecture (HPCA)*. IEEE, 2026, pp. 1–17.

[129] Y. Zhang and C. Jung, "Featherweight soft error resilience for gpus," in *2022 55th IEEE/ACM International Symposium on Microarchitecture (MICRO)*. IEEE, 2022, pp. 245–262.

[130] J. Zeng, S.-Y. Huang, J. Liu, and C. Jung, "Soft error resilience at near-zero cost," in *Proceedings of the 38th ACM International Conference on Supercomputing*, 2024, pp. 176–187.

[131] S.-Y. Huang, J. Zeng, X. Deng, S. Wang, A. Sifat, B. Bharmal, J.-B. Huang, R. Williams, H. Zeng, and C. Jung, "Rtailor: Parameterizing soft error resilience for mixed-criticality real-time systems," in *2023 IEEE Real-Time Systems Symposium (RTSS)*. IEEE, 2023, pp. 344–357.

[132] J. Zeng, H. Kim, J. Lee, and C. Jung, "Turnpike: Lightweight soft error resilience for in-order cores," in *MICRO-54: 54th Annual IEEE/ACM International Symposium on Microarchitecture*, 2021, pp. 654–666.

[133] Q. Liu, C. Jung, D. Lee, and D. Tiwarit, "Low-cost soft error resilience with unified data verification and fine-grained recovery for acoustic sensor based detection," in *2016 49th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, 2016.

[134] S.-Y. Huang, J. Jeong, H. Sasaki, and C. Jung, "Lightweight and tunable soft error resilience for protecting caches and memory against 2-bit errors," in *2026 IEEE Real-Time Systems Symposium (RTSS)*. IEEE, 2026.