Performance Evaluation of Compiler-based Software RMT in an HSA environment

Charu Kalra*, Daniel Lowell†, John Kalamatianos†, Vilas Sridharan‡, and David Kaeli*

* Northeastern University, Boston, MA, †Advanced Micro Devices, Inc. (AMD), USA
‡Northeastern University, Boston, MA

{ckalra, kaeli}@ece.neu.edu, {Daniel.Lowell, John.Kalamatianos, Vilas.Sridharan}@amd.com

Abstract—Reliability is one of the major challenges faced in exascale supercomputing today. While leveraging the inherent parallelism offered by Graphics Processing Units (GPUs) is crucial to accelerate the applications, it is equally important to ensure that applications can overcome data corruption caused by transient or permanent faults. Many hardware fault-tolerant mechanisms such as parity, Error Correction Codes (ECC), residue execution, and redundant execution are available, but their benefits rarely come without significant impact, including power, area, and cost overheads.

In this paper, we explore compiler-based Redundant Multithreading (RMT) as a low-cost software option to provide error detection on a Heterogeneous System Architecture (HSA) compliant Accelerated Processing Unit (APU). We take existing RMT schemes [22] and implement them on the HSA compiler stack. The RMT transformation pass is present in the LLVM compiler framework and automatically converts GPU compute kernels into redundantly-threaded versions. We present two algorithms for RMT execution which differ in their fault coverage (i.e., Sphere of Replication) on the device. We perform a detailed evaluation of both RMT algorithms to determine their performance overhead on real APUs while running scientific applications taken from signal processing, clustering, and linear algebra domains.

I. INTRODUCTION

Graphics Processing Units (GPUs) have become the accelerator of choice on a range of computing platforms and computing domains. The use of GPUs is no longer limited to graphics, but has been extended to support a broad range of compute-based applications. While leveraging the inherent parallelism offered by GPUs is crucial to accelerate an application, it is equally important to ensure higher reliability as we approach the exascale era.

Many hardware fault-tolerant mechanisms such as parity, ECC, residue execution [18], and redundant execution [13] are available, but their benefits rarely come without significant area, power, and cost overheads. Moreover, hardware protection is inflexible and not needed for certain workloads (e.g., graphics) which are inherently fault-tolerant. This calls for exploration of low-cost, portable software solutions for improving reliability on processors. Furthermore, having an automated solution prevents or reduces the need for programmer intervention.

One of the challenges with software solutions is the constant evolution of processor architectures and programming frameworks. The power and performance impact of a software solution may vary as the underlying processor architecture changes. This requires a re-evaluation to determine whether the solution continues to be feasible on newer architectures.

Recently, the HSA Foundation introduced the Heterogeneous System Architecture (HSA), an industry-led effort to improve programmability and efficiency of heterogeneous system programming [9]. HSA systems provide a unified view of memory to multiple processing cores (such as CPUs and GPUs) fabricated on the same die. The close proximity of these two processing cores offers the ability to tightly couple the memory system, which facilitates direct sharing of data structures. We provide the details of HSA features in Section II.

In this paper, our focus is not to develop another resiliency technique, but rather evaluate an existing compiler-based GPU Redundant Multithreading (RMT) technique on an HSA-based Accelerated Processing Unit (APU) [22]. Shared memory systems such as HSA-compliant System-on-Chips (SoCs) are likely to be the future of high-performance compute because of their significant performance and power advantages. It is therefore important to determine whether RMT is a viable choice in such an environment.

The major contributions of our work are:

- We take two existing RMT algorithms and implement them on an HSA compiler stack [22]. The RMT transformation pass is present in the LLVM compiler framework and automatically converts a GPU kernel into a redundantly-threaded version.
- We perform a detailed performance evaluation of both RMT algorithms on a real APU for scientific applications belonging to signal processing, clustering, and linear algebra domains. To our knowledge, this work is the first to explore compiler-based RMT on an HSA-based APU, providing a detailed analysis of the performance trade-offs of using RMT to achieve different levels of fault coverage.
- We show that the RMT variant that uses global memory to detect errors experiences the highest performance loss due to the high latency of synchronization between redundant computations. Kernels with high memory write traffic cause bottlenecks in the Scalar Unit and Local Data Share (LDS) for RMT variant that uses LDS to detect errors.

II. BACKGROUND

A. Fault Modes

The increasing computational capability of processors has caused exponential growth in the number of transistors per chip. Packing more transistors on a chip leads to decreasing feature sizes, which makes circuits more susceptible to
permanent (hard) and transient (soft) faults. Permanent faults can be caused by manufacturing defects, thermal stress, or circuit aging, and may develop over time as a combination of these phenomena. Transient faults can be caused by crosstalk, voltage violations, or electromagnetic interference, but are typically associated with the effects of high-energy particles. An alpha particle or a neutron strike can cause bit flips in multiple storage or logic gates and may drive the wrong value temporarily [12]. These temporary upsets in a transistor’s state are called single-event upsets (SEUs) or, if more than one transistor is affected, single-event multi-bit upsets (SEMUs). SEUs that affect the final outcome of the program are called silent data corruptions (SDC). The goal of RMT is to detect all SEUs before they corrupt the program state.

B. AMD Graphics Core Next (GCN) Architecture

We have implemented and evaluated RMT on GPUs with AMD GCN architecture [2]. Compute Units (CU) are the fundamental units of computation on AMD GPU architectures. Figure 1 shows a high-level diagram of a single GCN CU. Each CU has four 16-wide SIMD units, each SIMD capable of executing a single 64-wide vector instruction over 4 cycles. Each CU also has its own 64KB register file, capable of supporting up to 256 64x32-bit vector general-purpose registers (VGPRs). Each CU also has 64KB of LDS, a low-latency scratchpad memory where HSA local memory is allocated. Each CU also includes one scalar unit (SU) to improve SIMD execution efficiency. Each SU has 8KB space for scalar general-purpose registers (SGPRs).

C. Heterogeneous System Architecture

The Heterogeneous System Architecture (HSA) provides a unified view of fundamental computing elements and encompasses both hardware and software components [17]. An HSA-compliant hardware requires an HSA software stack (namely runtime and driver) to deliver the systems capabilities [4]. HSA provides programming features such as a single address space between the CPU and the GPU. HSA unifies the CPU and the GPU memory by allowing shared page tables, page faulting support, and a coherent memory subsystem. HSA improves inter-device communication by implementing user-space queuing and also supports preemptive context switching to improve Quality-of-Service (QoS) across all computing elements in the system.

HSA exposes the parallel nature of GPUs through the HSA Intermediate Language (HSAIL). HSAIL is translated to the underlying hardware’s ISA (instruction set architecture) using a finalizer. HSAIL is fairly low level and similar to the assembly language of a RISC machine. The smallest unit of execution in HSAIL is referred to as a work-item, which executes on a processing core of a compute unit on the device. Work-items are expressed in terms of independent units of execution known as workgroups. Workgroups execute on the compute units of the device and are organized as an n-dimensional index space known as an NDRange. A set of work-items within a single workgroup, which executes a single instruction in SIMD fashion on the compute units, is known as a wavefront.

D. Redundant Multithreading

All methods that are used to detect unwanted changes in state ultimately rely on some sort of redundancy. In one case, this can either be encoded versions of state or full duplication. Fully replicated state is said to be within a sphere of replication (SoR), and is assumed to be protected. All values that enter the SoR must be replicated (called input replication) and all redundant values that leave the SoR must be compared (called output comparison) before a single correct copy is allowed to leave [15]. RMT accomplishes replication by running two identical redundant threads — a producer and consumer — on replicated input. Whenever state needs to exit the SoR (e.g., on a store to unreplicated global memory), the producer sends its value to the consumer for checking before the consumer is allowed to execute the instruction. Two faults could create simultaneous identical errors in redundant state, but this is considered sufficiently unlikely. Hardware structures outside the SoR must be protected via other means.

III. RELATED WORK

The popularity of GPUs in servers and HPC systems has prompted researchers to develop resiliency techniques to fit the GPU’s unique programming model and architecture. Dimitrov et al. [8] proposed three methodologies of redundant execution, namely: 1) duplicate kernel execution, 2) instruction-level redundancy, and 3) thread-level redundancy, in order to achieve software reliability in GPU applications with large execution overheads. These techniques are applicable to older VLIW architectures and/or were hand-coded, thereby limiting their practical usefulness.

Wadden et al. proposed three compiler-based RMT techniques for AMD’s discrete GPU systems [22]. They carry out a detailed performance and power analysis of these algorithms for applications belonging to the AMD APP SDK [21]. In our work, we implement two of their RMT algorithms, but our work differs from their work in several ways. First, we implement the RMT transformation in the HSA compiler stack, unlike previous work which focused on the OpenCL runtime and compiler stack. The OpenCL compiler uses OpenCL intrinsics based on LLVM v3.2, whereas the HSA compiler

1wavefront is a collection of 64 workitems in GCN, executing in lockstep
uses HSA specific intrinsics and is based on LLVM v3.6 [10]. Second, we carry out our evaluation on an HSA-compliant APU device, which is a new architectural standard proposed by the HSA Foundation, while the work in [22] used a non-HSA compliant discrete GPU as their evaluation platform. Due to differences in hardware platform, compiler, and runtime, a direct comparison of our work with the previous work is not possible.

IV. IMPLEMENTATION

In this paper, we implement two RMT algorithms - Intra-Group RMT and Inter-Group RMT adapted from Wadden et al. [22], on the HSA software stack. These differ in terms of their fault coverage (Sphere of Replication) provided on the device. The state within the sphere of replication (SoR) is duplicated and all structures within the SoR are assumed to be protected from both hard and soft faults. All values that enter the SoR must be replicated and all redundant values that exit the SoR must be compared before a single copy is allowed to leave [15].

- **Intra-Group RMT:** Intra-Group RMT doubles the number of work-items within the workgroup and each work-item has a redundant copy of itself within the same wavefront. Since work-items within a workgroup can communicate via the Local Data Share (LDS), the communication buffer for sharing and comparing values lives in the LDS. For instructions whose values exit the SoR (e.g., global stores, local stores), we insert code to allow work-items to communicate their values with their redundant work-item.

- **SoR:** In the HSA model, each work-item executes its computation on its own SIMD lane and has access to its private vector registers. Therefore, we guarantee that SIMD functional units and the Vector GPRs are protected by Intra-Group RMT. Global memory, LDS, SGPRs and Scalar Unit lie outside the SoR, thus we assume they are protected by other fault-tolerant mechanisms.

- **Inter-Group RMT:** Inter-Group RMT doubles the number of workgroups within an NDRange. The redundant work-items are present in a different workgroup. Work-items across workgroups can communicate using global memory, therefore we insert the communication buffer in the global memory. Unlike Intra-group RMT, where work-items within a wavefront execute in a lock-step manner (which guarantees ordering of work-items), there is no guaranteed ordering of workgroups, hence we use explicit synchronization to coordinate communication between work-items.

- **SoR:** In Inter-Group RMT, redundant threads are present within different wavefronts. All scalar and vector instructions are duplicated and use their own sets of scalar and vector registers. Therefore, the SIMD units, Scalar unit, Vector GPRs, Scalar GPRs, Instruction Fetch, and Decode Logic are within the SoR, and hence, protected by RMT. We also include the LDS within the SoR, as each workgroup accesses its own LDS region.

**HSA compiler stack:** We implement both RMT algorithms in the HSA compiler stack. Figure 2 shows the various stages present in the HSA compiler toolchain. The frontend converts the GPU kernel code (.cl file) into the LLVM IR [10]. The LLVM optimizer produces an optimized version of the LLVM IR, which is fed to the backend. We modified the LLVM optimizer to add Intra-group and Inter-group as two RMT transformation passes. The backend converts the optimized LLVM IR into architecture agnostic HSA intermediate language (HSAIL). The finalizer translates HSAIL into hardware ISA at runtime.

V. EVALUATION METHODOLOGY

A. Platform for Evaluation

We perform a detailed evaluation of both RMT algorithms to determine their performance overheads on an AMD A10-7850K APU. The details of the APU are provided in Table I [3]. We implement both Intra and Inter versions of RMT as a transformation pass in a production-quality HSA compiler. Our HSA software framework comprised of Linux Kernel Driver v1.4 and HSA Runtime 1.0 Final. The Finalizer, which converts HSAIL into hardware ISA, is embedded in the HSA runtime and is invoked during kernel execution. We use CodeXL v1.8 to profile the HSA applications [1]. The hardware/software configuration details are summarized in Table II.

**TABLE I: Device configuration of AMD A10-7850K**

<table>
<thead>
<tr>
<th>Compute Unit Config</th>
<th>Device Configuration</th>
</tr>
</thead>
<tbody>
<tr>
<td># of CU’s</td>
<td>4 CPU + 8 GPU</td>
</tr>
<tr>
<td># of Wavefront Contexts / CU</td>
<td>40</td>
</tr>
<tr>
<td># of SIMD Units / CU</td>
<td>4</td>
</tr>
<tr>
<td># of lanes / SIMD</td>
<td>16</td>
</tr>
<tr>
<td># of vector reg / SIMD</td>
<td>64KB</td>
</tr>
<tr>
<td># of scalar reg / CU</td>
<td>8KB</td>
</tr>
<tr>
<td>GPU clock</td>
<td>120 MHz</td>
</tr>
<tr>
<td>Memory Architecture</td>
<td></td>
</tr>
<tr>
<td>L1 (16KB/CU)</td>
<td></td>
</tr>
<tr>
<td>Total L2 cache size</td>
<td>16KB</td>
</tr>
<tr>
<td>L2 cache size</td>
<td>128KB</td>
</tr>
<tr>
<td>Global Memory</td>
<td>1GB</td>
</tr>
<tr>
<td>Local Memory / CU</td>
<td>64KB</td>
</tr>
</tbody>
</table>

**TABLE II: Details of the evaluation platform.**

<table>
<thead>
<tr>
<th>APU Model</th>
<th>AMD A10-7850K</th>
</tr>
</thead>
<tbody>
<tr>
<td>Motherboard</td>
<td>ASUS A88X PRO</td>
</tr>
<tr>
<td>OS</td>
<td>Ubuntu 14.04 LTS</td>
</tr>
<tr>
<td>Driver</td>
<td>v1.4 [5]</td>
</tr>
<tr>
<td>Runtime</td>
<td>HSA 1.0 Final [6]</td>
</tr>
<tr>
<td>Profiler</td>
<td>CodeXL v1.8</td>
</tr>
</tbody>
</table>

B. Evaluated Benchmarks

We use four HSA kernels - Matrix Multiplication Simple, Matrix Multiplication Tiled, Kmeans, and Finite Impulse Response (FIR) for our evaluation [7], [14].

- **Matrix Multiplication Simple (MMS)** fetches input elements from the global memory, performs operations on them, and stores them back to the global memory.
- **Matrix Multiplication Tiled (MMT)** is an optimized version which uses local memory to load/store elements from the matrix and performs all operations using the local memory.
• KMeans (KM) is a clustering algorithm which partitions a data set into k number of groups, such that the total distance from each data point to its corresponding cluster centroid is minimized [11]. KMeans consists of two kernels - cluster (KMC) and swap (KMS). Cluster kernel takes a list of points with some number of feature dimensions, calculates the distance between the points and the centroid, and assigns membership to clusters based on distance. The swap takes the feature matrix and transposes it.

• A Finite Impulse Response (FIR) filter produces an impulse response of finite duration, which is the response to any finite length input. The kernel uses weighted reduction to find the impulse response of a finite duration of a synthesized audio stream [16].

All four kernels are written using SNACK (Simple No API Compiled Kernel) functions [7]. SNACK uses our modified compiler to convert the .cl file containing the GPU code to an object file. The baseline for the kernels do not have the RMT transformation enabled. SNACK functions use the HSA runtime API calls on the host side. We ensure the input size for all kernels is large enough to saturate the wavefront contexts on the GPU, as shown in Figure III.

<table>
<thead>
<tr>
<th>TABLE III: Benchmarks</th>
</tr>
</thead>
<tbody>
<tr>
<td>Benchmarks</td>
</tr>
<tr>
<td>MMS</td>
</tr>
<tr>
<td>MMT</td>
</tr>
<tr>
<td>KMS</td>
</tr>
<tr>
<td>FIR</td>
</tr>
</tbody>
</table>

VI. RESULTS

In this section, we first present the overall change in execution time for each algorithm versus the baseline. To get better insight into the performance overhead of RMT, we collect performance counter data using the CodeXL tool.

A. Overall Execution Time

Figure 3 shows the overall performance overhead of Intra-Group RMT and Inter-Group RMT over the baseline, expressed as a % change in the execution time of all kernels, for each application. While both flavors of RMT-enabled kernels show a performance loss, Inter-Group RMT experiences several orders of magnitude of increase in execution time. Each kernel stresses the various resources on the CU differently, which is why we see a large variability in performance overheads across applications.

B. Change in Instruction Mix

In order to explain the performance drop, we first examine the dynamic instruction count change in Figure 4 for Intra-Group RMT. We see almost a 100% increase in Vector ALU (VALU), Vector Global Memory (VMEM), Scalar Memory (SMEM), LDS, and Scalar ALU (SALU) instructions for all kernel applications. The increase in VALU, VMEM, SMEM, and SALU is primarily due to the overhead caused by the additional instructions generated to support RMT. In Intra-Group RMT, the local and global memory are outside the SoR, therefore any global or local store exiting the SoR is compared before the value eventually commits to memory. As an example, the Kmeans-swap (KMS) kernel loads a value from one global memory location and stores it into another global memory location and this repeats for all the data points per work-item. Thus, KMS performs a number of global stores that scale with the input size. For each such store, we execute the share-and-compare instructions using local memory before the value is eventually committed, thereby resulting in a significant increase in LDS instructions with Intra-Group RMT. The Matrix Multiplication Tiled (MMT) version, which is highly local memory intensive, shows a similar increase in the number of LDS instructions since all local and global stores, which exit the SoR, trigger share and compare through LDS. Invoking share and compare through LDS first requires branching to a store into the communication buffer, the latter being present in the LDS. The stored value is then loaded from the LDS by the redundant thread and compared with its own value, and then finally stored to the memory location if the compare succeeds. All branch instructions are handled using Scalar ALU (SALU) instructions, which is why we see a sharp increase in the SALU instructions for the KMS and MMT kernels.

Figure 5 shows the corresponding change in dynamic instruction count for Inter-Group RMT. The communication buffer in Inter-Group is present in global memory and makes heavy use of explicit synchronization to allow access by different work-groups. We, therefore, see a large increase in the use of vector/global memory instructions across all application kernels. In Inter-Group RMT, the local memory is within the SoR, therefore only the global stores that exit the SoR have to be shared and compared before committing the final value to
memory. MMT no longer shows the same increase in SALU instruction count as Intra-Group RMT because local memory now lies within the SoR and hence, we do not trigger share and compare for any local memory operations. On the other hand, the KMS kernel has many global store memory operations exiting the SoR, which trigger a store into the communication buffer, and then a load by the redundant work-item, therefore increasing the number of scalar ALU instructions.

C. Change in Wait Latencies

Figures 6 and 7 show the average amount of time a wavefront spends waiting at various kinds of stalls. We quantify five kinds of stalls - stalls caused due to: 1) Vector Memory (VMEM) traffic (number of cycles a wavefront waits for VMEM access to return data), 2) LDS/SMEM traffic (number of cycles a wavefront waits for LDS or SMEM access to return data), 3) Barriers (number of cycles a wavefront spends waiting for other wavefronts to arrive at the barrier), 4) Dependencies (stalls inserted by compiler to preserve Read After Write (RAW) dependencies), and 5) Overall wavefront wait latency caused due to memory (LDS/SMEM and VMEM) stalls. Figure 8 shows the contribution of different stalls to the overall execution of the kernels. To explain the change in various kinds of wait latencies, we also examined the average live wavefronts per clock for all kernels. Figure 9 shows the change in average number of live wavefronts per clock for both RMT versions over baseline.

In Figure 6, we see that Intra-Group RMT shows a reduction in VMEM wait latency across all kernels. This mainly comes from the fact that the average number of live wavefronts per clock reduces substantially for all kernels, as shown in Figure 9. This happens because Intra-Group RMT increases LDS and GPR usage, which prevents more workgroups from being scheduled on the CU. As a result, we reduce the rate of VMEM requests generated by the active wavefronts on the CU. In Figure 8, for FIR and MMT, we observe a large component of the wavefront wait latency caused due to LDS/SMEM traffic. This is because FIR has a high percentage of Scalar memory traffic and MMT generates a high percentage of LDS traffic.

In Intra-Group RMT, we observe that the vector L1 cache performance improves across all kernels. This is because there is a reduction in live wavefronts per clock, as shown in Figure 9. The vector L1 cache is local to a CU, hence reducing the average number of live wavefronts per clock imposes less pressure on the cache hierarchy and helps reduce the vector L1 cache miss rate. We also observe that KMS and MMT generate more scalar data traffic which increases scalar data cache misses. However, since the average wavefront wait latency for accessing scalar data is lower than that of accessing vector data, the contribution of the scalar data cache to the performance loss is small.

In Inter-Group RMT, there is a large component due to VMEM traffic across all kernels, as shown in Figure 7 and Figure 8. This is because inter-work-item communication happens via global memory and requires explicit synchro-
nization between threads across workgroups to access the communication buffer. When one work-item writes to the communication buffer, it grabs the lock, while the redundant work-item (present in a different work group) spins until the lock is released and the value is ready to be read. This happens for every global store that exits the SoR.

For Inter-Group RMT, the access sequence to the L2 cache changes with RMT as we execute twice the number of global memory accesses along with the RMT-related accesses for the communication buffer. Even though the kernel-related global vector/scalar memory accesses are redundant, they do not necessarily arrive one after another to the L2 cache since they are generated by different workgroups (either on the same or different CUs). The instruction schedulers are agnostic of the redundant nature of the workgroups and do not try to issue simultaneously or in close temporal proximity the pairs of redundant instructions from the two workgroups. Both Inter- and Intra-Group RMT double the amount of L2 cache request traffic and can generate irregular cache access patterns which are likely to lower cache reuse and hence contribute to a loss in execution performance.

Figure 10 shows the average wavefront-ready latency for Intra- and Inter-Group RMT for all kernels. Wavefront Ready latency represents the number of cycles a wavefront was ready to issue an instruction but could not because of arbitration with other wavefronts and the cycles when a wavefront issued an instruction. Several factors affect wavefront ready latency: (a) the average number of live wavefronts per CU, since it influences the length of time a wavefront waits for other wavefronts to issue their instructions, (b) the dynamic instruction count executed per wavefront (which increases with RMT), and (c) the issue latency of each instruction (e.g., how many quad-cycles needed per instruction to execute). We see a significant increase in the wavefront ready latency for KMS and MMS with Inter-Group RMT because they experience a large increase in their dynamic instruction count per wavefront primarily due to lock spinning, while the average number of live wavefronts remains approximately the same.

TRADEMARK ATTRIBUTION: AMD, the AMD Arrow logo, and combinations thereof are trademarks of Advanced Micro Devices, Inc.
REFERENCES