Register Availability Analysis in Support of Instrumentation of GPU Kernels

Hsuan-Heng Wu
Computer Sciences Department
University of Wisconsin-Madison
Madison, WI 53711 USA
hwu337@wisc.edu

Barton P. Miller
Computer Sciences Department
University of Wisconsin-Madison
Madison, WI 53711 USA
bart@cs.wisc.edu

***Abstract*— As we started development of an instrumentation and analysis tool for GPU binaries, we received feedback from experienced users in both industry and government labs that there were significant examples of GPU kernels that consumed most, if not all of their registers and had no general-purpose stack or heap. The information contained in the GPU kernel headers seemed to confirm this feedback. Such a situation makes it difficult, if not impossible, to use techniques that are commonly used to instrument CPU binaries.**

To evaluate this belief, we conducted fine-grained code analysis to determine the actual register usage of the kernels in AMD's MIOpen library, a comprehensive machine learning library known to have kernels with extreme register usage. Our analysis was based on the detailed information that we could obtain from the dataflow and liveness analysis that we implemented in our Dyninst binary analysis and instrumentation toolkit. We investigated the global and local register availability of these kernels: A register is globally available if it is allocated but never used by any instruction. A global register is suitable for storing values that require a global lifetime (such as the base address of a stack or heap). A register is locally available at an instruction if it is dead at that instruction but not globally available. Such a register can be used to perform computation if we decide to place instrument at that instruction.

We analyzed the minimum number of registers required to support conventional instrumentation (to enable access to a general purpose and heap so traditional instrumentation schemes can apply), which requires at least two globally available registers to hold the base address of a stack. Contrary to common belief, we found that more than 91% of the kernels in the MIOpen library can support conventional instrumentation without increasing a kernel's default register allocation. The number goes up to 98% when an increase in register allocation is allowed.

For the case when we do not have enough global registers even to hold the base address of a stack, we developed an algorithm that moves register values between locally available registers, allowing us to cover almost any existing GPU binary in the kernels that we analyzed.

Keywords—binary instrumentation, GPU kernels

I. MOTIVATION

Binary code instrumentation is used in a wide range of applications, including performance analysis [1] [14] [20], debugging [2], software reliability [14], and security [8] [10], [19]. Instrumentation of binary code is a critical capability in these applications because it does not require source code to be available and targets the actual software artifact that is executed.

A challenge of instrumenting binary code is having sufficient resources (such as free registers or memory), to save and restore processor state around the instrumentation code so that the behavior of the underlying application code is not affected, and to record instrumentation data. Compared to binary instrumentation of CPU code [3] [7] [13] [16] [18], some properties of the GPU execution model prevent us from directly applying the conventional CPU binary instrumentation approaches on GPU binaries:

1. Compared to CPU programs having access to a general purpose stack and heap, GPU programs are not guaranteed access to these memories, which are useful for register spilling and recording instrumentation data.
2. Compared to CPU programs that can make use of all architecturally available registers, GPU programs are launched with the minimal number of registers required, so that the more threads can run in parallel on the hardware. This means that there will be fewer registers available for instrumentation, and the instrumentation needs to be aware of register allocation to be efficient.
3. The problem worsens when a GPU program tries to use all architecturally available registers, as now we will not be able to trade efficiency for feasibility by asking for more registers to be used to launch the GPU program.

From our informal discussions with GPU vendors, GPU library teams, and application groups, there is the widely held belief that there are GPU codes that frequently use all the available registers in the application code, leaving none available for instrumentation code. The most commonly mentioned worst-case kernels are the hardware-optimized machine learning libraries provided by vendors such as cuDNN [6] for NVIDIA GPUs written in CUDA [17] and MIOpen [12] for AMD GPUs written in HIP [4]. In these libraries, it is not uncommon to see GPU kernels declaring the use of all available registers, making it impossible to claim additional registers for instrumentation purposes.

As we will demonstrate in later sections, the solutions to these issues are actually intertwined: We need enough registers to hold location information of a stack and a heap, and access to stack and heap allows us to free up more registers and use them

for instrumentation. Throughout this paper, our goal of enabling stack and heap access to GPU programs will be referred to as conventional instrumentation, as these accesses will enable us to adopt most if not all instrumentation solutions from the long-developed world of CPU binary instrumentation.

This paper describes our analysis of the register requirements to support conventional instrumentation, and our investigation of the actual register use for a highly demanding library running on the AMD GPU platform. Careful analysis of these kernels reveals that there is almost always sufficient minimum register availability at every instruction in every kernel. In fact, there are often more than the minimum available at many instructions, allowing instrumentation without any required register spilling. The significance of this result is that fine grained binary code instrumentation of these GPU kernels is practical in almost all cases.

There are a few simple ways in which we can measure register usage in a GPU kernel. For example, in an AMD GPU binary, we have the following three types of register usage information obtainable through a simple scan of the GPU application kernel.

1. Registers Allocated: GPU registers are allocated in units of 8 or 16. For each kernel, there exists a field in the kernel's header specifying the number of registers to be allocated at runtime.
2. Register Usage Declared: In the .note section of the kernel's header, there exists a field specifying the number of registers actually used in the kernel. which should be no greater than the amount of register allocated.
3. Registers Explicitly Used: For each instruction, we can decode its operands, checking which registers are used to get an accurate measure of register usage.

We first observed that the above measures often over approximate the number of registers being used. Not all instructions (or even basic blocks or functions) use all registers. And while the above techniques can tell you cumulatively which registers are being used, they cannot tell you, at any given point in the code, which registers might be free. To produce such per-instruction free register information, we use a liveness analysis [11] of the binary code. Those registers that are not in use by the application code can be used by the instrumentation code. The liveness information requires a control and dataflow analysis of the code to produce a conservative estimate of which registers are in use. In this case, conservative means that no register is ever marked as live if it contains data that might be again used by the application.

In this paper, we study the scalar and vector register availability for the 738 kernels in the MIOpen library using a variety of measures, including the free, explicitly used, and allocated registers. We leverage the Dyninst binary analysis and instrumentation toolkit [5] [9] [16] to create a tool that can perform the control flow, dataflow, and liveness analysis of the MIOpen GPU kernels on the AMD GFX908, GFX90A and GFX940 architectures.

Our findings show that free registers are available at almost all instructions in these kernels, demonstrating the feasibility of instrumenting these highly-optimized GPU library kernels with no additional register requirements. We go to show a variety of effective techniques for generating instrumentation of GPU kernels on the register resource availability at that site.

Our main results are as follows:

- The information contained in GPU executable headers is overly conservative by a large margin (Section III).
- For instrumentation needs, either enough registers are globally available or register allocations can be increased in 99% of the kernels on the GFX908, GFX90A, and 95% on the GFX940 processors (Section III)
- Register available can be increased to accommodate any instrumentation needed by a variety of register spilling techniques (Section IV).
- In cases where there are insufficient global registers available to meet the minimal needs, globally needed values for instrumentation can be moved from register to register based on our Sliding Tile Puzzle algorithm. This technique increases the percentage of kernels that we can instrument to 100% of the kernels on the GFX908, GFX90A, GFX940 processors (Section VI).

II. BACKGROUND

Our instrumentation strategies provide a foundation for the later sections of this paper. The GPU programming and execution model and registers play an important role in the GPU binary instrumentation, resulting in differences between global versus local register requirements for instrumentation. In this section, we describe each of these.

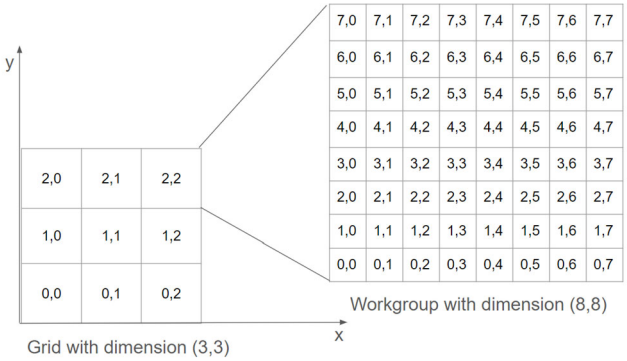


Fig. 1: Illustration of a kernel with grid dimensions (3,3) and workgroup dimensions (8.8)

A. Execution Model

With the Single Instruction Multiple Thread (SIMT) execution model, a GPU kernel program describes how a thread should execute, and the host decides how many threads are associated with the execution of this kernel. The threads are launched as a grid of workgroups, where the programmer determines the number of workgroups within a grid, and the number of threads within a workgroup. Threads within a workgroup are mapped to the same underlying compute unit, allowing them to synchronize and efficiently communicate through shared memory. The dimensionality of a grid and its corresponding workgroups can have one, two, or three

dimensions, depending on what the programmer thinks will better reflect the underlying problem space.

Fig. 1 illustrates an example of a kernel launching a 2-D grid with dimension (3,3) and workgroup dimension (8,8), for a total of 9 workgroups and 576 threads. The workgroup and the thread indices set by the runtime are annotated, with x-coordinates followed by y-coordinates.

At the hardware level, threads in a workgroup are further partitioned into wavefronts (64 threads per wavefront on AMD GPUs) or warps (32 threads per warp on NVIDIA GPUs) to execute in a lock-step fashion. Wavefronts are the unit of scheduling and resource allocation, and the execution of a GPU program essentially boils down to the concurrent execution of these wavefronts.

On AMD GPUs, there are two types of registers, scalar registers that are one instance per wavefront and vector registers that are one instance per thread. Given a GPU kernel program, the compiler determines how many resources (scalar registers, vector registers, and shared memory) are required to execute a wavefront, and outputs this information as a metadata in the header of the executable. This information is later used by the runtime to allocate just enough resources to execute the kernel. As the hardware has limited resources, the more resources that a wavefront uses, the fewer wavefronts that can be launched and executed in parallel.

A direct effect of this programming and execution model on GPU binary instrumentation is that we have a limit on the amount of resources that we can use to implement instrumentation. There are three cases to consider when we attempt to instrument an AMD GPU binary:

1. There are enough free registers available within the ones that are allocated to this kernel. Therefore, our instrumentation can be implemented using the allocated registers.
2. There are not enough free registers and we are able to increase the register allocation, potentially trading thread-level-parallelism and execution efficiency for instrumentation feasibility.
3. In the worst case, when we run out of architecturally-available registers, we admit defeat and label the kernel as not instrumentable.

B. *Global vs. Local Register Requirements*

Registers are key to forming instrumentation instructions, as they are commonly used to hold memory addresses and instruction operands.

There are two types of register requirements for our instrumentation purpose. The first type is global. This can be thought of as global variables holding values that are required throughout the entire execution. The second type is local to the instrumentation site. These local registers can be thought of as temporary variables that have the lifetime of the scope of a single instrumentation site.

To reserve registers to hold global variables, we pick the registers in the set-difference between the set of allocated registers and the set of explicitly used registers, as these registers should never be used by the kernel. To construct instrumentation instructions at an instrumentation site, we pick from the set of registers that are dead at that site.

III. Global Register Requirements: Enabling Conventional Instrumentation

Enabling conventional instrumentation requires enabling access to stack and heap for a GPU kernel, and allocation of registers that point to this memory. As noted in Section II.B, the register requirements discussed here are for global registers, as we want the stack and heap address to be available at any potential instrumentation site.

A. *Enabling Stack and Scratch Access for AMDGPU Kernels*

Scratch memory is a special memory space that is located in the global memory of the GPU that is automatically allocated on wavefront creation and freed on wavefront termination. It is typically used to implement stack.

An AMDGPU kernel is compiled with scratch/stack when the register pressure is too high (to perform register spilling), or when there are function calls in the kernel. If a kernel is compiled with scratch, at the start of the kernel there are instructions that compute the base address of the stack for each wavefront; we store them in a dedicated scalar register pair (FLAT_SCRATCH_LO / FLAT_SCRATCH_HI). Throughout the paper, we will refer to this scalar register pair as stack base. If there are function calls, an additional dedicated scalar register would be used to store the stack pointer (currently s32), which is a 32-bit offset from the stack base.

If a kernel is compiled with a stack, we do not need to do anything other than perhaps increasing the size of the stack. If a kernel is not compiled with a stack, then we modify the ELF header (.note section) to cause scratch memory to be allocated and modify the kernel descriptor to pass the address of this memory to the kernel when it is launched. We note that most GPU kernels do not have stacks as the compiler tries to inline functions and store variables in registers whenever possible.

If a GPU kernel is not compiled with a stack and we want to enable access to it, then we need three scalar registers globally available: **two to hold the stack base, and one to hold the stack pointer**. If possible, we would keep the stack base in the dedicated register pair FLAT_SCRATCH_LO / FLAT_SCRATCH_HI. If these registers are not globally available, but other registers are free, we can perform register swap at the instrumentation site.

B. *Enable Heap Access for AMDGPU Kernels*

A heap contains memory dynamically allocated at runtime, used to hold data whose lifetime is not associated with a particular function invocation. It is natural to record instrumentation data on a heap. However, the concept of the heap itself does not really fit well with the GPU programming model, as GPU memory typically is allocated by the host before the launch of the kernel, with the pointers to the memories passed to the kernel as arguments. Attempts to allocate memory from the GPU result in slow and complicated GPU to CPU interactions.

To enable GPU kernel access to a heap, the host allocates device memory and passes the address to it to the kernel as an

additional argument. Performing such allocation requires that we know the maximum amount of memory required per thread. Assuming there is no recursion (which is a common assumption in GPU programming models), we can determine the maximum memory requirement for each thread based on the instrumentation that we have inserted into the code

To enable access to the heap throughout execution, we require **two globally available scalar registers to hold the base address of the heap (heap base) and one globally available vector register to hold an unique per-thread-ID** that allows each thread to access into its own partition of the heap.

Fig. 1: Heap Layout

In Section II.A, we described how threads are organized in a grid-workgroup structure. To index a specific thread, we require information about its workgroup index into the grid, its thread index into the containing workgroup, number of threads in a workgroup, and number of workgroups in a grid. To simplify the computation and storage, we assign a unique flattened thread ID, *ftid*, to each thread projecting the 3-dimensional grid-workgroup structure onto a contiguous 1-dimensional array of threads.

Fig. 2 shows the layout of such heap, which allows each thread to access the base address of its own memory chunk by: (heap base) + *ftid* × (per thread memory requirement).

To compute *ftid*, we rely on values set by runtime when a thread is launched. The standard runtime launches all threads in the same workgroup with that workgroup's index in the grid, `grid[i,j,k]`. We modified the runtime to also include the dimensions of the grid, a_{grid} x b_{grid} x c_{grid}, at launch time.

Each thread is launched with its index into its workgroup, `grid[i,j,k]->wg[x,y,z]`. We modified the runtime to also include the dimensions of the workgroup, a_{wg} x b_{wg} x c_{wg}. We note that all workgroups are of the same dimension, so we can simply use a_{wg}, b_{wg}, and c_{wg} to describe all workgroups.

Since each thread's index is local to its workgroup, we use the above information to create a unique ftid for each thread:

```
for all i < a_grid, j < b_grid, k < c_grid
  for all x < a_wg, y < b_wg, z < c_wg
    grid[i,j,k]->wg[x,y,z]->ftid =
       (i × a_wg + x) + a_wg × a_grid ×
         ((j × b_wg + y) +
          (b_wg × b_grid × (k × c_wg + z)))
```

IV. Reducing Global Requirements for Enabling Stack and Heap

As we have discussed, we want four scalar registers and one vector register globally available to enable stack and heap access for a GPU kernel. What if we do not have enough registers globally available? To answer this question, we leverage the liveness analysis in the Dyninst binary tool suite. Fig. 3 plots the fraction of kernels that have N or more registers free globally in the MIOpen library. Combining the scalar and vector register requirements, conventional instrumentation can be supported on 18.83%, 50.27% and 46.63% of the kernels on GFX908, GFX90A, GFX940, where the number of globally free vector registers is the limiting factor. On the other hand, if we allow increased register allocation to the maximum provided by the hardware, the number of globally free scalar registers can become the limiting factor. This is because there are a few kernels that use the maximum number of scalar registers. However, increasing allocation to the maxim still allows us to instrument 98.92% of the kernels on GFX 908, 98.78% of the kernels on GFX90A, and 95.80% of the kernels on GFX940 are instrumentable.

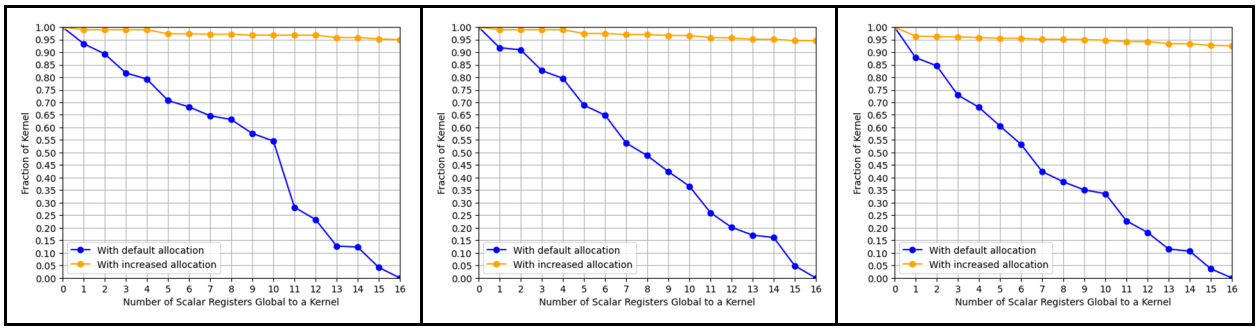
With some loss of efficiency, we can reduce the global register requirement down to two scalar registers or one vector register. As long as we have two global scalar registers holding the address of the stack, we can store the stack pointer, the heap base and the ftid at fixed offset relative to the stack base, and having one global vector register is equivalent to having 64 global scalar registers. This increases the percentage of kernels that can support conventional instrumentation up to 91.33% 96.34% 93.22% with the default register allocation, and 100%, 98.82% and 98.78% with maximum allocation for GFX908, GFX90A and GFX940.

Fig. 4 shows the layout of the stack of a wavefront, where each row corresponds to a stack entry that can store a vector register or 64 scalar registers. To make sure that we do not interfere with the stack access of the unmodified binary, we add a fixed offset to the FLAT_SCRATCH_LO / FLAT_SCRATCH_HI registers such that they point to the new stack base. At the instrumentation site where we need to load the spilled values, we subtract the fixed offset to recover the old stack base and use it to access the spilled values.

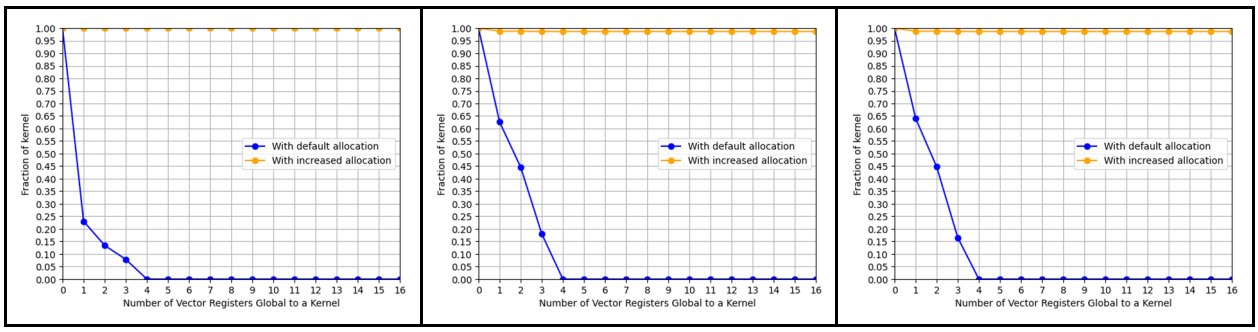
As shown in the figure, in addition to the space needed to store the stack pointer, heap base and ftid, we allocate spill space of the same size. When there are no dead registers at the instrumentation site, we can spill live registers into these locations to free registers to hold the stack offset, heap base and ftid.

V. Local Register Requirements: Example Instrumentation of Per Basic Block Counter

Assuming that we have stack and heap access information available (in global registers), local registers are required for implementing the actual instrumentation. We start by presenting a simple example to understand the register requirements for instrumentation, a counter that records how many times a basic block is executed for each thread. We then give a detailed breakdown of the register requirements for this example. We then present a progression of techniques that increase the number of kernels that we can instrument in the MIOpen library.



(a) Scalar GFX908 (b) Scalar GFX90A (c) Scalar GFX940



(d) Vector GFX908 (e) Vector GFX90A (f) Vector GFX940

Fig. 2: Per Kernel Globally Free Registers in MIOpen

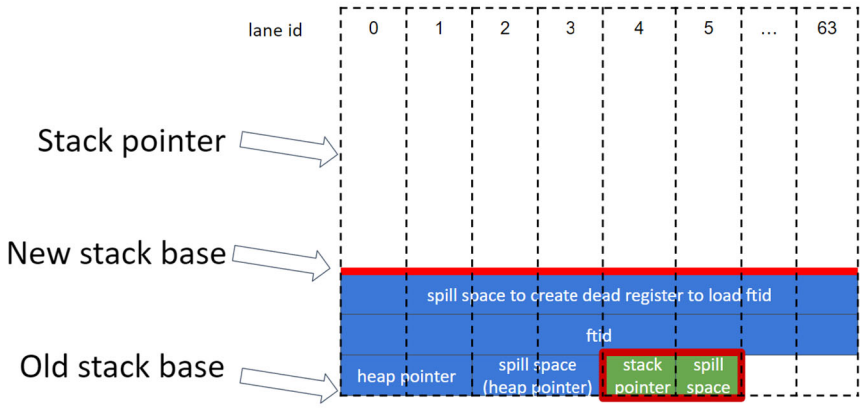


Fig. 3: Stack Layout

A. Example: Basic Block Counter

We want to implement a counter for each thread that gets incremented each time a basic block is entered.

We implement the counters as zero-initialized 8-byte memory slots in the heap. These slots are laid out sequentially, indexed first by the ftid then by the basic block ID, *bid*. The counters can be incremented through GPU vector memory instructions, with the address of a counter determined by

$$\text{counter address} = (\text{heap base}) + \mathit{ftid} \times \#\text{basic-blocks} \times 8 + \mathit{bid} \times 8$$

The total amount of memory needed is **#threads** × **#basic-blocks** × 8 bytes**.** Fig. 5 provides an example heap layout for a kernel with four basic blocks.

In Listing 1 (Appendix), we show the details of a two dimensional example of the instrumentation prologue used to compute the ftid. At the start of a kernel, most registers are dead except those set up by the runtime. In this example, **s[4:5]** is set to the address of the kernel argument list, which we use to access the grid dimensions, the workgroup dimensions, and the heap base; scalar registers **s6** and **s7** are set to the grid index `[i,j]`; and vector registers **v0** and **v1** are set to the workgroup index `[x,y]`. We also have scalar registers **s[16:17]** reserved for holding the heap base and vector register **v14** reserved for holding the ftid. Registers other than these are dead, so they are available for use in the instrumentation prologue. **In total, this requires a reservation of two scalar registers to hold the base address and one vector register to hold the ftid.**

In Listing 2 (Appendix), we show the details of the instrumentation sequence for each basic block, where we actually increment the counters. We use the same assumptions in Listing 1 (Appendix), where we have stored the heap base in **s[16:17]** and the ftid in **v14**. Here we have vector registers **v10** to **v13** available at the instrumentation site and we use them for instrumentation. We use the vector memory instruction `global_atomic_inc_x2`, which increments a 64-bit value pointed by **v[10:11]**. As this instruction also takes a 64-bit threshold value (it resets the memory to 0 when in-memory value exceeds the threshold), we need two more registers to hold the threshold value -1.

In addition to the two scalar registers and one vector register that we reserved in the prologue, we require four more vector registers to be available at the instrumentation site, two holding the heap base and two holding the cutoff value.

B. Feasibility with Local Free Registers Information

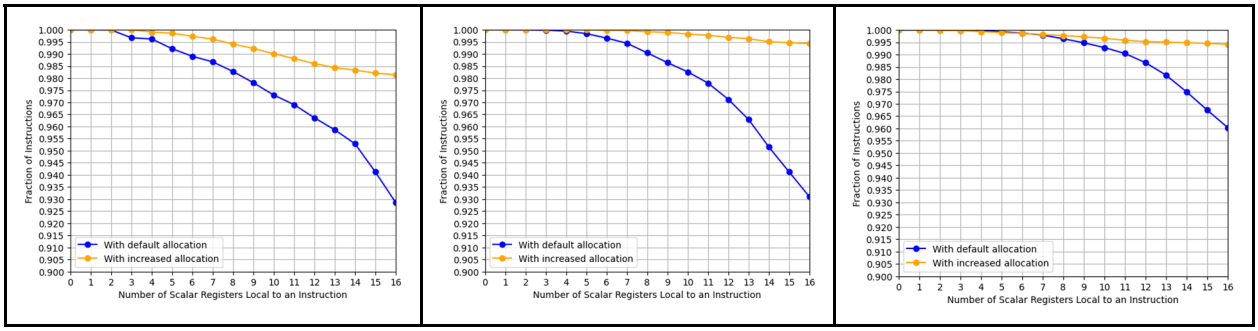
To increase the coverage of instrumentable kernels, we want to leverage locally free registers whenever possible. Looking back at our example in Section V.A, only the heap base and ftid need to reside in globally free registers, while register **v10** to **v13** only need to be available at the instrumentation site. In total, we need only two scalar registers and one vector register globally free, with four vector registers available at the instrumentation site.

Fig. 6 plots the fraction of instructions in the kernels that can support conventional instrumentation (with default allocation or maximum allocation) that have N or more registers free locally. We can see that 99.61% of kernels in GFX908, 99.47% of the kernels in GFX90A, and 99.97% of the kernels in GFX940 have enough globally free registers with their given register allocation. These go up to 99.90%, 100% and 99.93% respectively, when we allow increased register allocation.

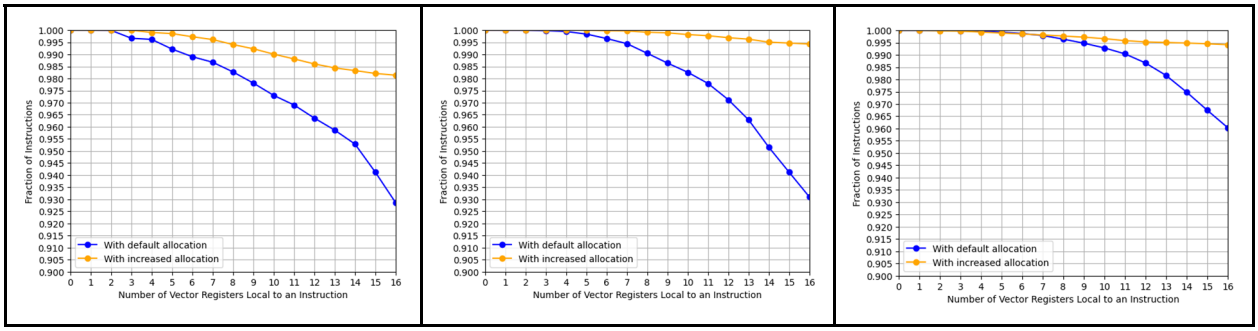
VI. Increase Instrumentation Coverage through Value Sliding and Pre-Spilling

In Section IV, we discussed how we can reduce the global register requirement to enable conventional instrumentation down to two scalar registers for holding the stack base. These registers might be obtained by expanding the number of registers allocated to this kernel or by finding registers that were free across the entire kernel. There are still a few cases where such registers were not globally available, such as when all registers are already allocated by the compiler and used by the kernel, or when the use of additional registers is undesired because of reduced parallelism. As we saw in Section IV, this situation never occurs in MIOpen kernels on GFX908, and about 1% of the kernels on GFX90A, and GFX940.

Fig. 5: Example Heap Layout for kernels with 4 basic blocks



(a) Scalar GFX908 (b) Scalar GFX90A (c) Scalar GFX940



(d) Vector GFX908 (e) Vector GFX90A (f) Vector GFX940

Fig. 6: Per Instruction Local Registers in MIOpen

To be complete and cover these few cases, we ask: Can we still keep the stack base available throughout the execution, therefore enable spilling and heap access for instructions with sufficient dead registers, when it is not possible to find enough globally free registers?

We designed an algorithm inspired by the sliding-tile puzzle game. Assuming that the stack base is available at the instrumentation prologue, we want to keep the stack base in registers at all times, just not always the same registers. The key idea is to move instrumentation values from their current register to a register that will be dead at the next instruction or spill to create new dead registers if all registers are live at the next instruction. Since we need to move the stack base value through control flow edges to different basic blocks, complete knowledge of the control flow is assumed, or act as a restriction of whether this approach applies.

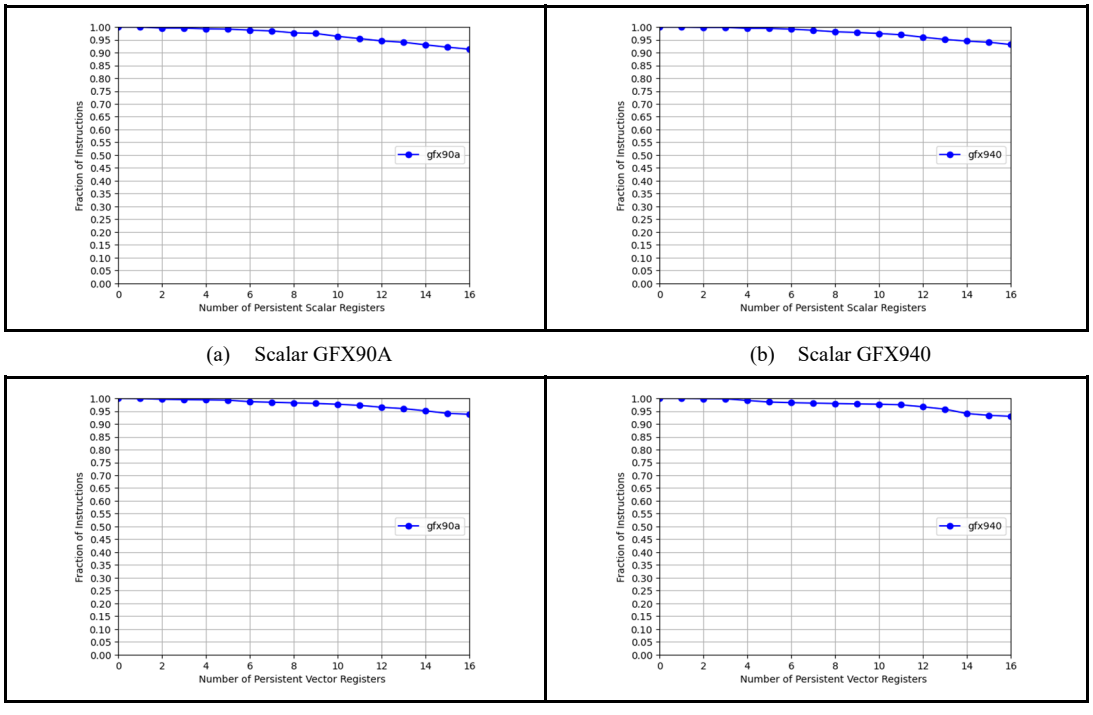
We first discuss the two cases that we need to handle in the algorithm: the case where we can leverage the existing dead registers of an instruction to slide the stack base through execution, and the case where some instructions does not have enough dead registers and we need to create new dead registers to hold these values. We then present the general algorithm.

A. Passing Values through Locally Free (Dead) Registers

Assume you need to reserve a register for instrumentation at instruction *I*. This register should not be used or overwritten by *I*, so we should pick a register that is dead before the instruction and not defined by that instruction. We call these *Persistent Registers, PR(I) = DEAD_PRE(I) - DEF(I).*

In the simplest case where every instruction has enough persistent registers, we can, for each instruction, pick a set of persistent registers, and insert move instructions between each neighboring instructions to slide the *ftid* and the instrumentation memory pointer through.

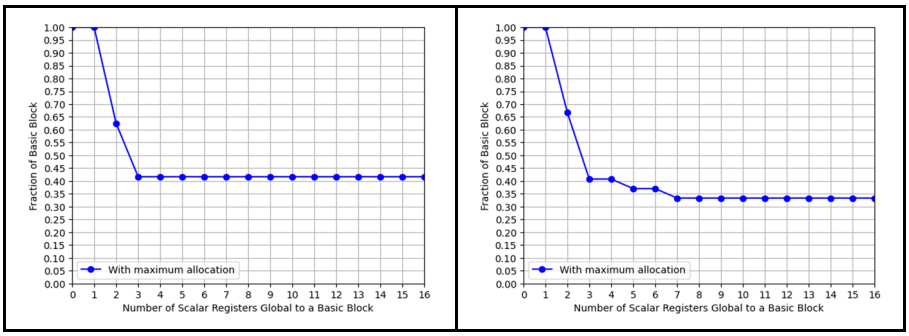
For the 8 kernels on GFX90A and the 9 kernels on GFX940 that cannot support conventional instrumentation under maximum register allocation, we plot the fraction of instructions that have N or more persistent registers available in the MIOpen library. Since all kernels on GFX908 can support conventional instrumentation under maximum register allocation, they are not included in this discussion. If an instruction has at least two persistent scalar registers or one persistent vector register, we have enough registers to slide the stack base through the instruction, meaning more than 99.91% of the instructions in GFX90A and GFX940 satisfy this requirement.



(a) Scalar GFX90A (b) Scalar GFX940

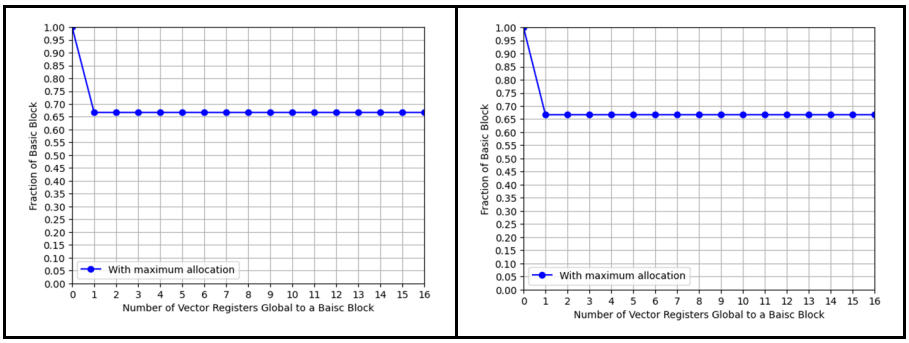
(c) Vector GFX90A (d) Vector GFX940

Fig. 7: Per Instruction Persistent Registers



(a) Scalar GFX90A

(b) Scalar GFX940



(c) Vector GFX90A

(d) Vector GFX940

Fig. 8: Per Basic Block Free Registers

B. Pre-spilling for Instruction without Enough PRs

From Fig. 7, we know that for the kernels that we cannot allocate enough global registers to support conventional instrumentation, only around 0.09% of the instructions do not have enough persistent registers to hold the stack base.

To make the problem easier to understand, we introduce two definitions: critical instruction (CI) and non-critical instruction (*NCI*). A critical instruction is an instruction without enough persistent registers to hold the ftid and the instrumentation memory pointer, whereas a non-critical instruction is the instructions that has enough registers to hold the stack base. For any CI, we want to create more persistent registers for them, by spilling registers that are live but not immediately used to stack, so we can use these newly created persistent registers to hold the stack base and pass them through execution, effectively turning them into a NCI. We will then recover the spilled registers once we find new registers to hold the stack base in later instructions. As the spilling for these instructions must happen at some instruction that gets executed earlier, where the stack base information is still available, we named this approach *pre-spilling*.

1) Critical Instructions not located at the start nor the end of a Basic Block

We start with the simple case, assuming that a critical instruction is not the first or last instructions in a basic block. This assumption comes from the observation that if a basic block corresponds to some scope in the source code, then at the start less registers are live as the computation has not started or values will be overwritten, whereas at the end of a scope intermediate values are no longer required, and holds true for all the critical instructions in the kernels in the MIOpen library.

Fig. 9 illustrates how instrumentation is inserted to perform pre-spilling. Fig. 9a demonstrates the simplest case where a critical instruction I_1 is surrounded by two non-critical instructions in the same basic block. There are three types of the instrumentation that need to be inserted: (1) Move that moves the stack base from the selected persistent registers (denoted by SBASE) of one instruction to the next instruction, highlighted in green, (2) Pre-Spill for critical instructions, highlighted in blue, and (3) Restore for critical instructions, highlighted in orange. The pre-spill of I_1 needs to happen before the execution of I_0, and the restore of registers pre-spilled for I_1 happens between I_1 and I_2, after moving the stack base between the persistent registers.

Fig. 9b shows how pre-spilling can be done when there are consecutive critical instructions, where I_1 and I_2 are critical instructions. Here the pre-spill for I_2 happens after the stack base is moved to the newly created persistent registers of I_1.

2) Critical Instructions located at either the start or the end of a Basic Block

When a critical instruction is located at either the start or the end of a basic block, it might have multiple incoming edges or multiple outgoing edges, making finding the pre-spill/restore location and picking the registers to pre-spill more complicated. To simplify this process, we insert an intermediate block for each control flow edge, as shown in Fig. 10. Fig. 10a has a basic

block ending with a critical instruction, I_{AN}, pointing to two block starting instructions I_{C0} and I_{D0},. We can pick the set of registers to pre-spill independently for basic block C and basic block D as they have separate pre-spill instrumentation, as shown in Fig. 10b. Fig. 10c has a single block starting critical instruction pointed by two critical instructions. By having separate restore instructions, we can pick the registers to be pre-spilled for basic block A and B independently, as shown in Fig. 10d. By inserting pre-spill and restore in the intermediate basic blocks, we effectively turn each basic block to start and end with non-critical instruction, and therefore can apply the same technique as described in Section VI.B.1.

C. Sliding Tile Puzzle Algorithm

The sliding puzzle algorithm has two phases. The first phase makes use of register liveness results to pick the set of registers to hold the stack base (by picking persistent registers for non-critical instructions or spillable registers for critical instructions). The second phase takes these results and instruments the program to produce the necessary sliding of the stack base.

1) Picking Registers to hold the Stack Base

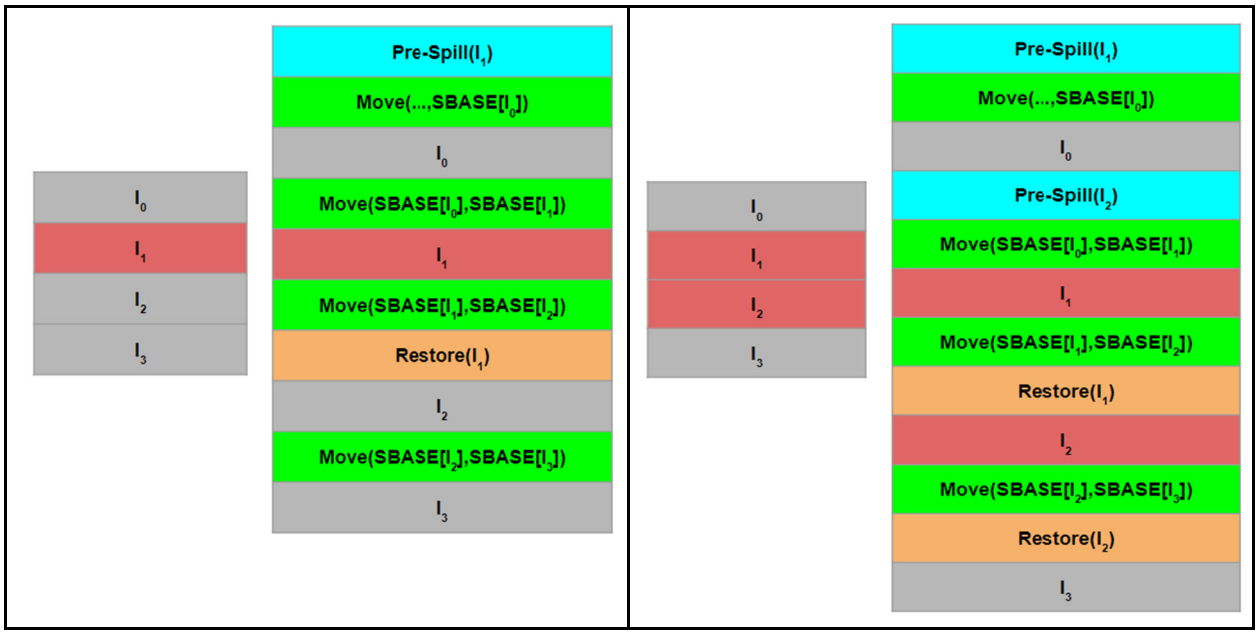
Figuring out the set of persistent registers for a non-critical instruction is simple by definition. As for critical instructions, following the discussion in Section VI.B, we effectively pre-spill each critical instruction at the instruction immediately preceding it, so the computation of the spillable register set of a critical instruction involves only the critical instruction and its immediate predecessor.

Listing 3 (Appendix) shows the details of how we select the registers used to hold the stack base, for each instruction. For any critical instruction CI, we initialize its set of spillable registers to the registers that are not defined nor used in that instruction. Then for each immediate predecessor instruction I_{prev} of CI, we refine CI's spillable register set to exclude the define and use set of I_{prev}. For simplicity, we pick the registers with the smallest register ID from the set of persistent registers or spillable registers to hold the stack base.

2) Insertion of Implementation

For an individual instruction I, there are three related types of instrumentation that need to be done. If it is a critical instruction, there needs to be a pre-spill, a restore, and, regardless of it being critical or not, a move of stack base from the registers in the previous instruction to the current instruction.

The case of the move and restore is pretty straight forward: we insert the move between an instruction and its predecessors, and we insert restore between an instruction and its successors. For a critical instruction internal to a basic block, the pre-spill should happen right before its predecessor, whereas when it resides at the start of a basic block, the pre-spill should happen at the intermediate block that is right before the start of the instruction in question. When multiple instrumentation maps to the same instrumentation site, the pre-spill instrumentation should happen first, followed by move and restore. We present a breadth first search based algorithm that starts from the first instruction of the kernel and records what registers and what kind of instrumentation is needed at each instrumentation point. It then iterates through all the instrumentation points and inserts the corresponding instrumentation, the details shown in Listing 4 (Appendix).



(a) Pre-Spill for a CI I_1 internal to a basic block, surrounded by NCIs

(b) Pre-Spill for two consecutive CIs internal to a basic block, surrounded by NCIs

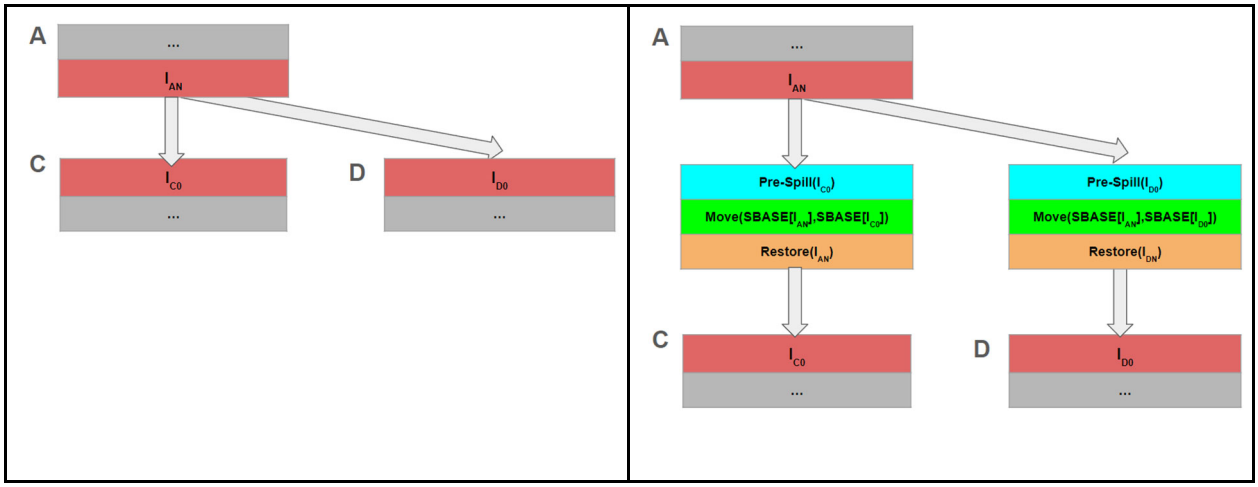
Fig. 9: Pre-Spilling for Critical Instructions Internal to a Basic Block

VII. Conclusion

In this paper, we studied the scalar and vector register availability for the 738 kernels in the MIOpen library using a variety of measures, including the free, explicitly used, and allocated registers. We leveraged the Dyninst binary analysis and instrumentation toolkit to build a tool that performs control flow, dataflow, and liveness analysis of the MIOpen GPU kernels on the AMD GFX908, GFX90A, and GFX940 architectures.

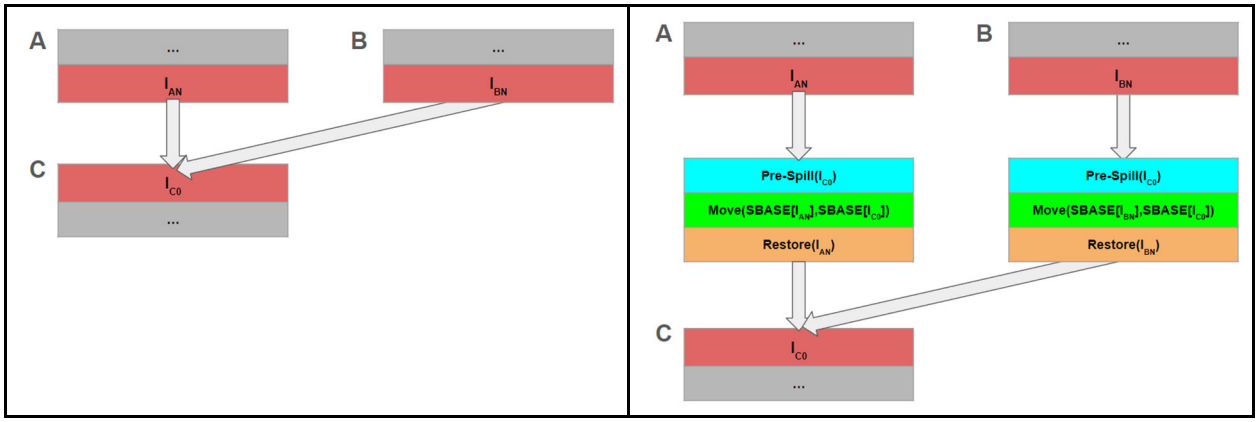
Our findings show that instrumentation is feasible even for these highly optimized library kernels that are commonly believed to have very few registers available.

We described the minimal register requirement for AMDGPU to enable conventional instrumentation and show that conventional instrumentation can be enabled on more than 91% of the kernels in the MIOpen library, across the GFX908, GFX90A, and GFX940 architectures with default register allocation, and this number goes up to 98% when we allow increased register allocation. Finally, we presented an algorithm that enables register spilling when there are not enough globally free registers to even hold a stack base, allowing us to cover almost any existing GPU binary on these architectures.



(a) Critical instruction at the end of block A with multiple successor critical instructions in C and D

(b) Pre-Spill instrumentation for the first instructions of basic blocks C and D, move instrumentation for registers holding the stack base for edges (A_N, C_0) and (A_N, D_0), and restore instrumentation for A_N.



(c) Critical instruction at the start of block C with multiple predecessor critical instructions in blocks A and B

(d) Pre-Spill instrumentation for the C_0, move instrumentation for the registers holding the stack base for edges (A_N, C_0) and (B_N, C_0), and the restore instrumentation for A_N and B_N.

Fig. 10: Instrumentation in Intermediate Blocks

REFERENCES

[1] L. Adhianto, S. Banerjee, M. Fagan, M. Krentel, G. Marin, J. Mellor-Crummey and N.R. Talent, "HPCToolkit: tools for performance analysis of optimized parallel programs", *Concurrency and Computation: Practice and Experience* **22**, 6, April 2010, pp. 685-701.

[2] D.C. Arnold, D.H. Ahn, B.R. de Supinski, G.L. Lee, B.P. Miller and M. Schulz, "Stack trace analysis for large scale debugging", *21st IEEE International Parallel and Distributed Processing Symposium (IPDPS),* Long Beach, Calif, March 2007.

[3] D.L. Brening, "Efficient, transparent, and comprehensive runtime code manipulation", *Ph.D. Dissertation,* Massachusetts Institute of Technology, Cambridge, MA, Order Number AAI0807735, September 2024. http://hdl.handle.net/1721.1/30160.

[4] P. Bauman, N. Chalmers, N. Curtis,C. Freitag, J. Greathouse, N. Malaya, D. McDougall, S. Moe, R. Oostrum, and N. Wolfe, "Introduction to AMDGPU programming with HIP", Presentation at OakRidge National Laboratory, April 2021. https://www.olcf.ornl.gov/wp-content/uploads/2021/04/IntroGPUProgramming-ORNL-Hackathon-May24-26-2021.pdf

[5] B. Buck and J.K. Hollingsworth. 2000. "An API for runtime code patching", International Journal of High Performance Computing Applications **14**, 4, November 2000, pp. 317–329

[6] S. Chetlur, C. Woolley, P. Vandermersch, J. Cohen, J. Tran, B. Catanzaro and E. Shelhamer, "cuDNN: efficient primitives for deep learning", https://arxiv.org/abs/1410.0759, October 2014.

[7] A. Eustace, and A. Srivastava, "ATOM: a flexible interface for building high performance program analysis tools", *USENIX 1995 Technical Conference*, New Orleans, January 1995.

[8] W. Fang, B.P. Miller and J.A. Kupsch, "Automated tracing and visualization of software security structure and properties", *9th International Symposium on Visualization for Cyber Security (VizSec),* Seattle, Washington, October. 2012.

[9] J.K. Hollingsworth, B.P. Miller and J. Cargille, "Dynamic program instrumentation for scalable performance tools", Scalable High-performance Computing Conference (SHPCC), Knoxville, Tennessee, May 1994.

[10] E.R. Jacobson, A.R. Bernat, W.R. Williams and B.P. Miller, "Detecting code reuse attacks with a model of conformant program execution", *International Symposium on Engineering Secure Software and Systems (ESSoS),* Munich, Germany, February 2014.

[11] K. Kennedy, "A global flow analysis algorithm", *International Journal of Computer Mathematics* **3**, 1-4, December 1971, pp. 5–15.

[12] J. Khan, P. Fultz, A. Tamazov, D. Lowell, C. Liu, M. Melesse, M. Nandhimandalam, K, Nasyrov, I. Perminov, T. Shah, V. Filippov, J. Zhang, J. Zhou, B. Natarajan, and M. Daga, "MIOpen: an open source library for deep learning primitives", *30th International Conference on Computer Graphics and Machine Vision (GraphiCon)*, Saint Petersburg, Russia, September 2020**.** https://doi.org/10.51130/graphicon-2020-2-2-2

[13] C.-K. Luk, R. Cohn, R. Muth, H. Patil, A. Klauser, G. Lowney, S. Wallace, V. Janapa Reddi, and K. Hazelwood, "PIN: building customized program analysis tools with dynamic instrumentation", 2*005 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, Chicago, Illinois, June 2005. https://doi.org/10.1145/1065010.1065034.

[14] F. Long, S. Sidiroglou-Douskos and M. Rindard, "Automatic runtime error repair and containment via recovery shepherding", *35th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI),* Edinburgh, Scotland, June 2014.

[15] B.P. Miller, M.D. Callaghan, J.M. Cargille, J.K. Hollingsworth, R.B. Irvin, K.L. Karavanic, K. Kunchithapadam and T. Newhall, "The Paradyn parallel performance measurement tool", *IEEE Computer* **28**, 11, November 1995, pp. 37-46.

[16] X. Meng and B.P. Miller, "Binary code is not easy", *25th International Symposium on Software Testing and Analysis (ISSTA)*, Saarbrucken, Germany, July 2016, pp. 24-35. https://doi.org/10.1145/2931037.2931047

[17] J. Nickolls, I. Buck, M. Garland and K. Skadron, "Scalable parallel programming with CUDA", *ACM Queue* **6,** 2, March 2008.

[18] N. Nethercote and J. Seward, "Valgrind: a framework for heavyweight dynamic binary instrumentation", *28th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI),* San Diego, California, June 2007. https://doi.org/10.1145/1273442.1250746

[19] P. O'Sullivan, K. Anand, A. Kotha, M. Smithson, R. Barua and A.D. Keromytis, "Retrofitting security in COTS software with binary rewriting", *26th IFIP TC-11 International Information Security Conference (IFIP SEC),* Hamburg, Germany, June 2011.

[20] S.S. Shende and A.D. Malony, "The Tau parallel performance system", *International Journal of High Performance Computing Application*s **20**, 2, May 2006, pp. 287-311.

```
s_load_dwordx2 s[10:11], s[4:5], 0x1c      ;s10 = a_grid, s11 = b_grid
s_load_dwordx2 s[12:13], s[4:5], 0x24      ;s12 = a_wg, s13 = b_wg
s_load_dwordx2 s[16:17], s[4:5], 0x10      ;heap base in
                                           ;s[16:17]
s_waitcnt vmcnt(0) expcnt(0) lgkmcnt(0)    ;wait until load is done

s_mul_i32 s15, s7, s13                     ;s15 = j  × b_wg
v_mov_b32_e32 v5, s15                      ;v5  = j  × b_wg
v_add_u32_e32 v5, v1, v5                   ;v5  = j  × b_wg + y

s_mul_i32 s14, s6, s12                     ;s14 = i  × a_wg
v_mov_b32_e32 v4, s14                      ;v4  = i  × a_wg
v_add_u32_e32 v4, v0, v4                   ;v4  = i  × a_wg + x

s_mul_i32 s14, s10, s12                    ;s14 = a_grid  × a_wg
v_mul_lo_u32 v5, s14, v5                   ;v5  = (j  × b_wg + y)  × a_grid  × a_wg
v_add_co_u32_e32 v14, vcc, v5, v4          ;v14 = (j  × b_wg + y)  × a_grid  × a_wg +
                                           ;      (i  × a_wg + x)
. . .                                      ;v14 now holds ftid
Original Kernel Starting Instructions
```

Listing 1: Instrumentation Prologue

```
BB0:
 v_mov_b32_e32 v12, s17                            ;v12 = s17 for later computation
 v_mov_b32_e32 v10,  32                            ;v10 = #basic-blocks  × 8 = 32
 v_mul_lo_u32  v10, v10, v14
 v_mul_hi_u32  v11, v10, v14                       ;v[10:11] = (ftid  × 48)
 v_add_co_u32  v10, v10, s16
 v_addc_co_u32 v11, v11, v12                       ;v[10:11] = s[16:17] + tid  × 32
 v_mov_b32_e32 v12, -1
 v_mov_b32_e32 v13, -1                             ;v[12:13] = -1
 global_atomic_inc_x2 v[10:11], offset:0, v[12:13] ;[v[10:11]+0]+=1 (for BB0)

 Original BB0 instructions
. . .

BB1:
 v_mov_b32_e32      v12, s17                       ;v12 = s17 for later computation
 v_mov_b32_e32      v10, 32                        ;v10 = #basic-blocks  × 8 = 32
 v_mul_lo_u32       v10, v10, v14
 v_mul_hi_u32       v11, v10, v14                  ;v[10:11] = (ftid  × 32) >> 32
 v_add_co_u32       v10, v10, s16
 v_addc_co_u32      v11, v11, v12                  ;v[10:11] = s[16:17] + tid  × 32
 v_mov_b32_e32      v12, -1
 v_mov_b32_e32      v13, -1                        ;v[12:13] = -1
 global_atomic_inc_x2 v[10:11], offset:8, v[12:13] ;[v[10:11]+8]+=1 (for BB1)

 Original BB1 instructions
. . .
```

Listing 2: Instrumentation Site

```
// Compute the set of persistent registers of non-critical instructions, NCI
DEF ComputePersistentRegs(kernel):
   FOREACH NCI in the kernel:
      persistentRegs[NCI] = REGS_ALLOCATED - NCI.LivePre() - NCI.Def() - NCI.Use()

// Compute the set of spillable registers of critical instructions, CI
DEF ComputeSpillable(kernel):
   FOREACH CI in the kernel:
      spillableRegs[CI] = REGS_ALLOCATED - CI.Def() - CI.Use()
      FOREACH instruction PI that has an edge from PI to CI:
         spillableRegs[CI] = spillableRegs[CI] - PI.Def() - PI.Use()

// Pick the stack base holder for each instruction
//   Pick from persistent registers if the instruction is non-critical
//   Pick from spillable registers if the instruction is critical
DEF ComputeStackBaseHolder(kernel):
   ComputePersistentRegs(kernel)
   ComputeSpillable(kernel)
   FOREACH instruction I in the kernel:
      IF I is non-critical:
         stackbaseRegs[I] = 2 registers in persistentRegs[I] with smallest ID
      IF I is critical:
         stackbaseRegs[I] = 2 registers in spillableRegs[I] with smallest ID
```

Listing 3: Picking the Registers to Hold the Stack Base

```
// Below are three functions that record the registers involved in three maps for
// each type of instrumentation, indexed by two consecutive instructions

DEF RegisterPreSpill(curInstr,prevInstr):
   IF curInstr == BB.Instructions()[0]:
      preSpillRegs[prevInstr,curInstr] = stackBaseRegs[curInstr]
   ELSE // For CI internal to a Basic Block, pre-spill before its predecessor
      preSpillRegs[prevInstr.Predecessor(),prevInstr] = stackBaseRegs[curInstr]

DEF RegisterRestore(curInstr,nextInstr):
   restoreRegs[curInstr,nextInstr] = stackBaseRegs[curInstr]

DEF RegisterMove(curInstr,prevInstr):
   moveRegs[prevInstr,curInstr] = (stackBaseRegs[prevInstr],stackBaseRegs[curInstr])

// Below are three helper functions that insert instrumentation between the two
// input instructions (or their corresponding intermediate block)

// Instruments preSpill the registers in preSpillRegs[prevInstr,curInstr]
DEF InstrumentPreSpill(prevInstr,curInstr)

// Instruments move from stackBaseRegs[prevInstr] to stackBaseRegs[curInstr]
DEF InstrumentMove(prevInstr,curInstr)

// Instruments restore for registers in restoreRegs[curInstr]
DEF InstrumentRestore(curInstr,nextInstr)

DEF SlidingTilePuzzle(kernel):
   ComputeStackBaseHolder(kernel)
   workQueue = Queue(kernel.Instructions()[0])
   WHILE workQueue.length() !=0 :
      curInstr = workQueue.Dequeue()
      FOREACH prevInstr in curInstr.Predecessor():
         RegisterMove(curInstr,prevInstr)
         IF curInstr.IsCritical():
            RegisterPreSpill(curInstr,prevInstr)
      FOREACH postInstr in curInstr.Successor():
         RegisterRestore(curInstr,postInstr)
         workQueue.Enqueue(postInstr)
      FOREACH (prevInstr,curInstr) in preSpillRegs:
         InstrumentPreSpill(prevInstr,curInstr)
      FOREACH (prevInstr,curInstr) in moveRegs:
         InstrumentMove(prevInstr,curInstr,])
      FOREACH (curInstr,nextInstr) in restoreRegs:
         InstrumentRestore(curInstr,nextInstr)
```

Listing 4: Sliding Tile Puzzle Algorithm