Binary Analysis and Instrumentation for AMDGPU in Dyninst

Hsuan-Heng Wu and Ronak Chauhan

Computer Sciences Department
University of Wisconsin-Madison

Scalable Tools Workshop

July 2025

WISCONSIN
UNIVERSITY OF WISCONSIN–MADISON

Dyninst Paradyn

High-Level Goals

Boring Goal

- Normal Dyninst binary analysis and instrumentation should work for AMDGPU

Interesting Goal

- Adapt analysis and instrumentation for SIMD/SIMT architectures:
  - Abstract SIMD/SIMT control flow operations to a conventional CFG
  - Represent SIMD/SIMT data flow accurately and compactly
  - Efficient Fine Grained Instrumentation
    - Fine grained:
      - Record information at wave or thread level granularity
    - Efficient:
      - Synchronization-free instrumentation
      - Use control and data flow analysis to guide instrumentation

Outline

- Background
  - SIMT, wavefronts and execution mask
  - Predicated execution and predicated control flow
- Binary Analysis of AMDGPU Kernels
  - Control flow analysis of GPU binaries
  - Dataflow analysis of GPU binaries
- Design Efficient Instrumentation for GPU Applications
  - Leverage GPU architecture for efficient GPU instrumentation

Background

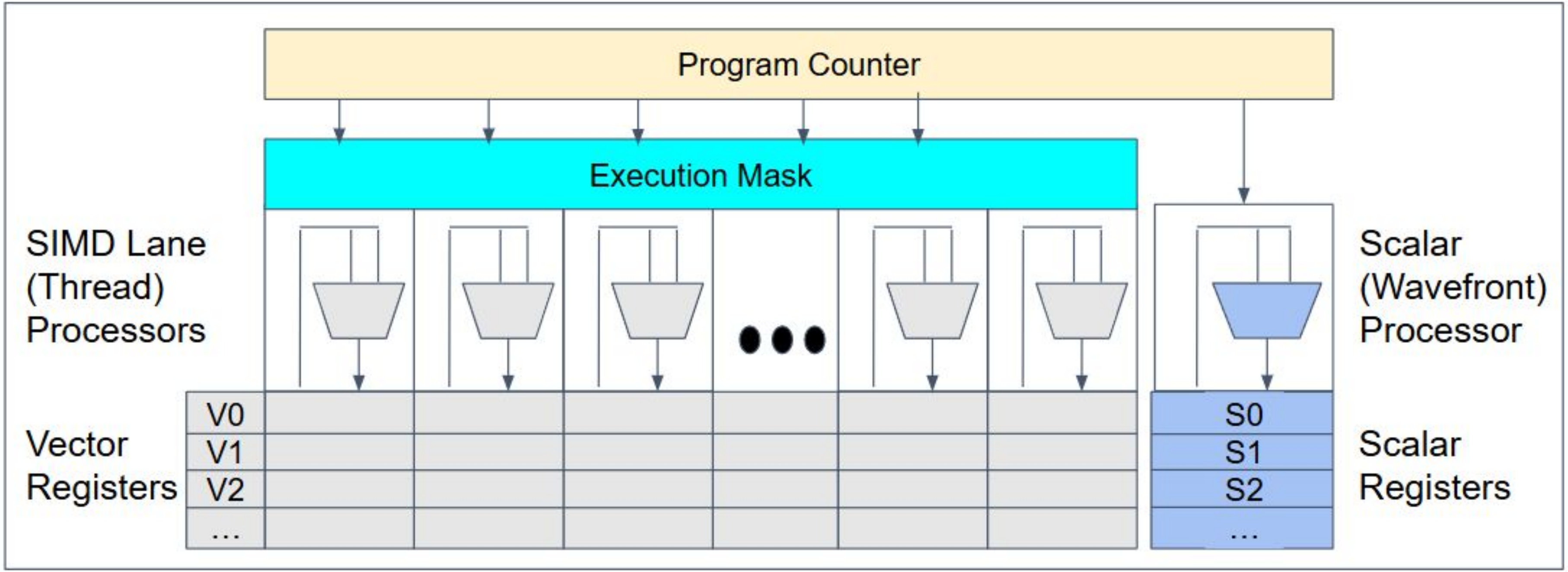
Background – SIMT, Wavefronts and Execution Mask

Threads grouped in **wavefronts (warps)** to execute in Single Instruction Multiple Data (SIMD)

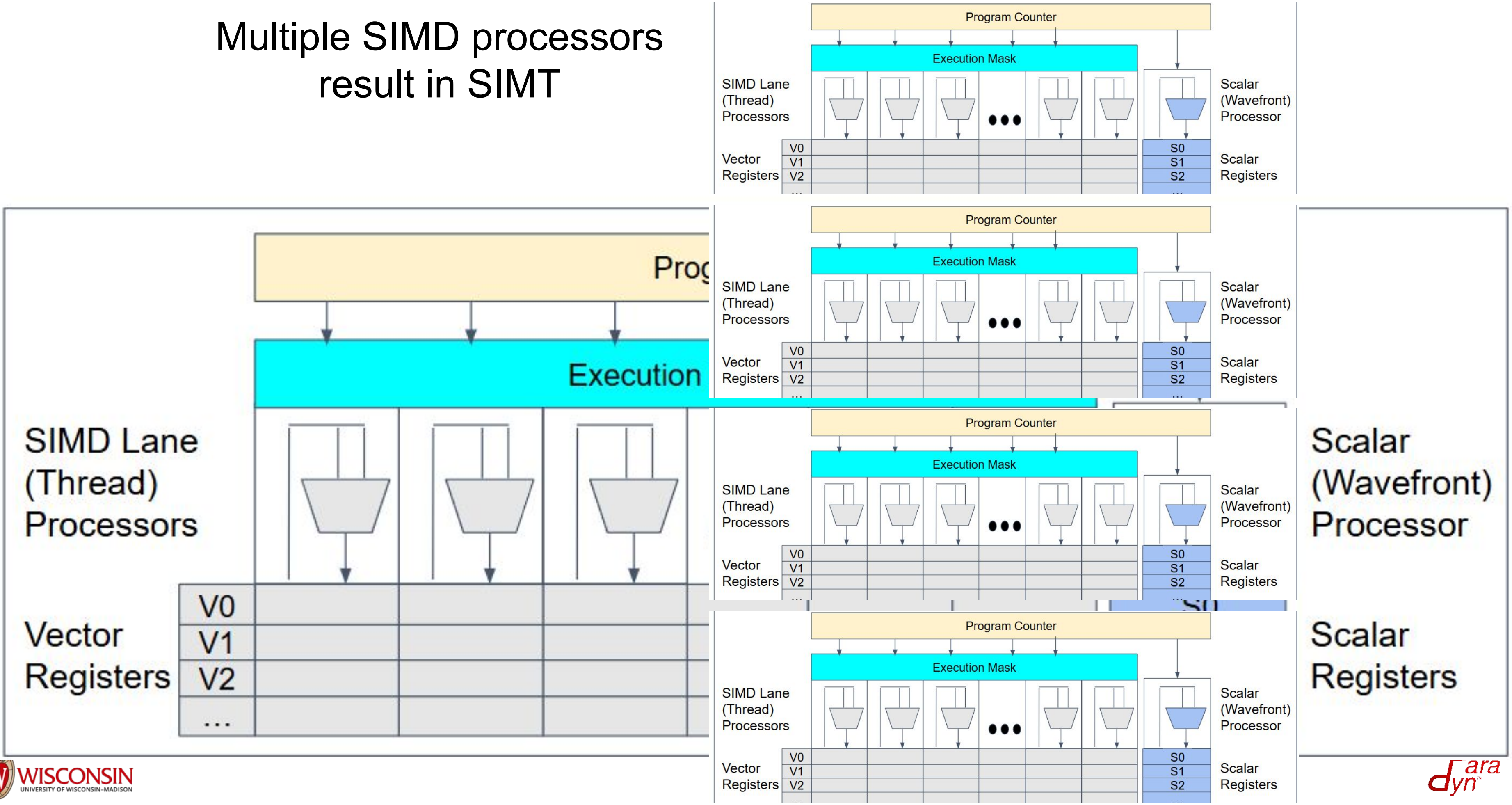
- AMDGPU Kernels uses 64 threads in a wavefront
- Each thread in a wavefront (lane)
  - Shares the same program counter
  - By default operates on their corresponding lane of the vector data
  - Can be disabled by clearing a bit in a execution mask register

Single Instruction Multiple Thread - Multiple wavefronts executing the same kernel

SIMD Processor

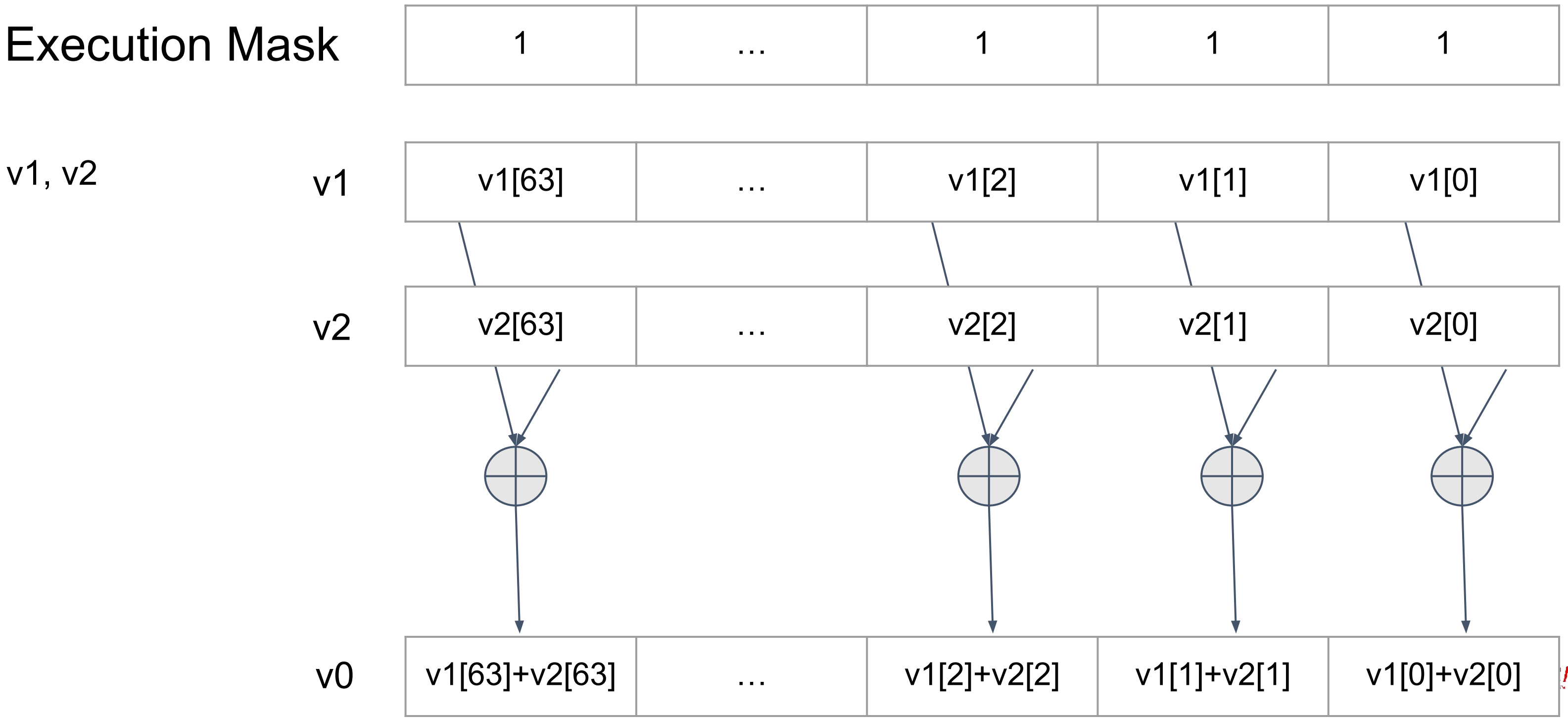


Multiple SIMD processors result in SIMT



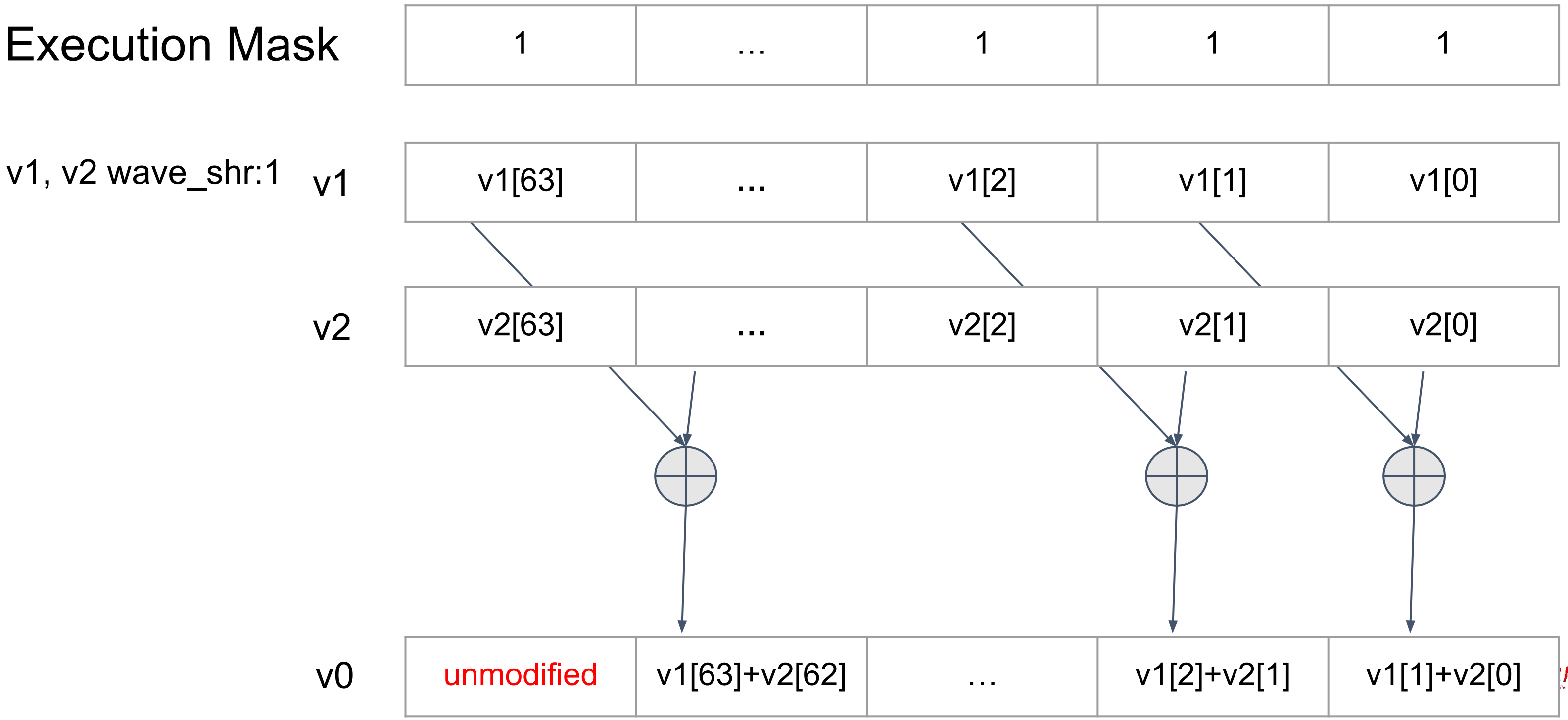
Background – Vector Registers & Vector Instructions

v_add_u32 v0, v1, v2



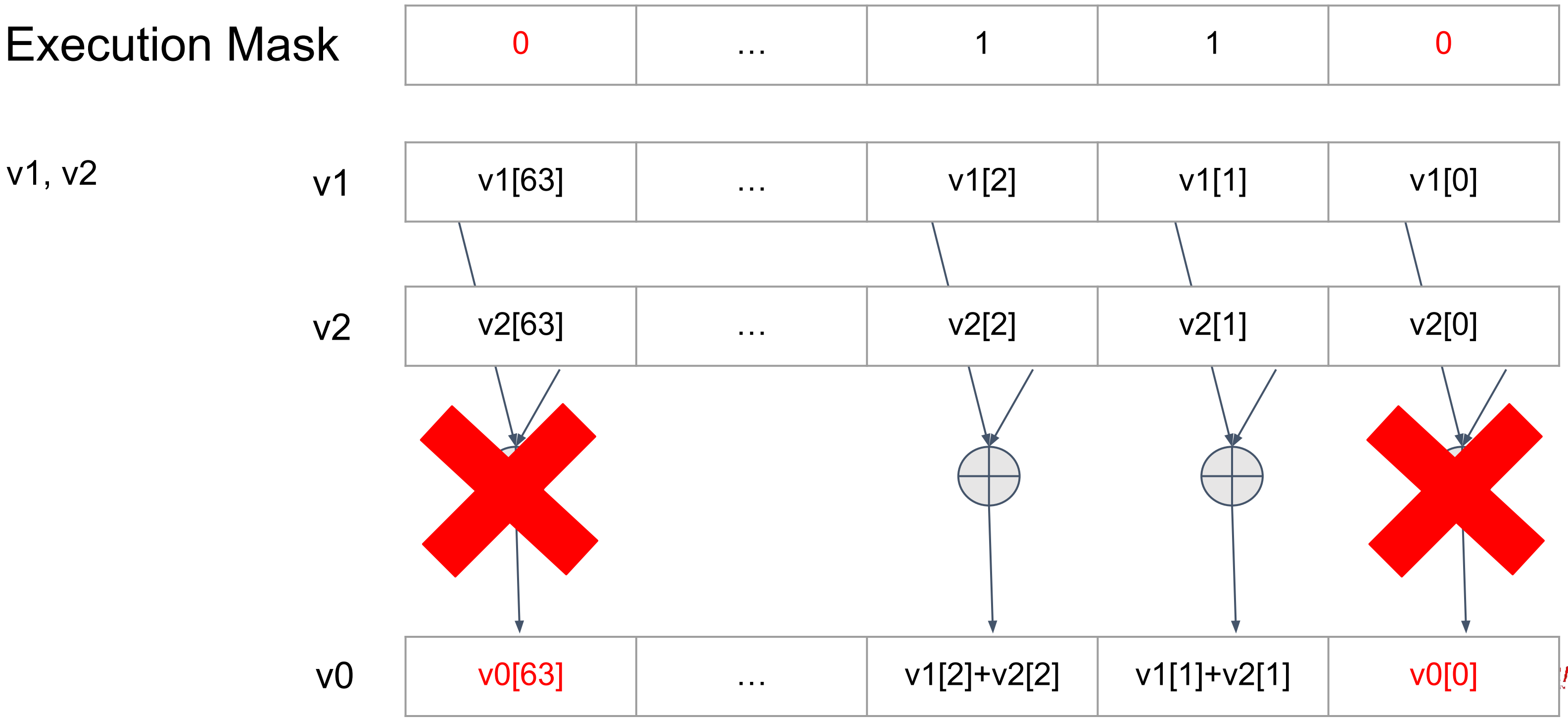
Cross Lane Operations – Lane Shifting

v_add_u32 v0, v1, v2 wave_shr:1



Background – Predicated Execution

v_add_u32 v0, v1, v2



WISCONSIN
UNIVERSITY OF WISCONSIN–MADISON

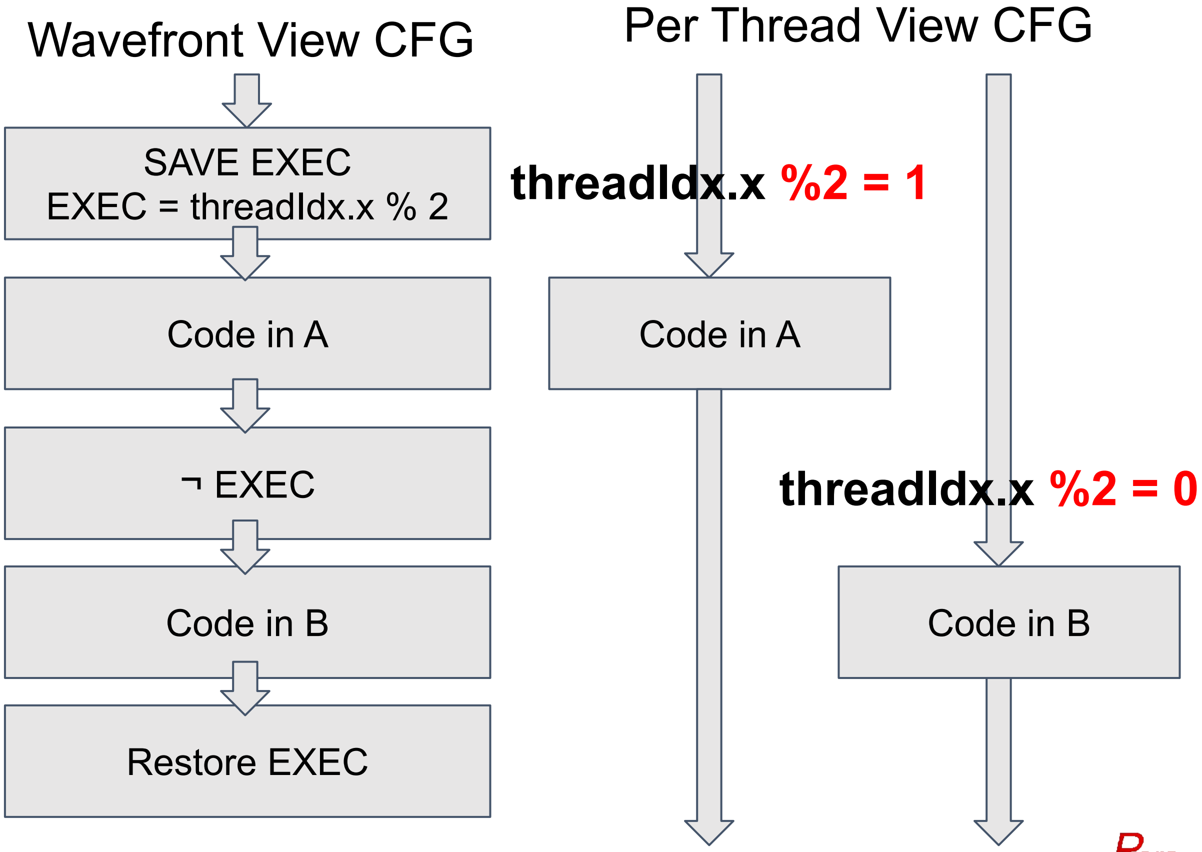
```
IF (threadIdx.x % 2)
{

   A; // odd threads

} ELSE  {

   B; // even threads

}
```



Binary Analysis for AMDGPU Kernels

Control Flow Analysis for AMDGPU Kernels

Analyzing control flow at the wavefront level is similar to that of CPU binary

- Threads in a wavefront share the same PC
- Control flow instructions are scalar

Analyzing per thread control flow is the interesting topic here

- **Control flow analysis enables data flow analysis**
- Under predicated control flow, threads can have different control flow based on EXEC
  - **IF**-THEN-ELSE, DO-**WHILE**, SWITCH-**CASE**, SIMT Jump **Table**
- Map these code constructs to conventional code constructs for programmer's to understand the semantics of the binaries

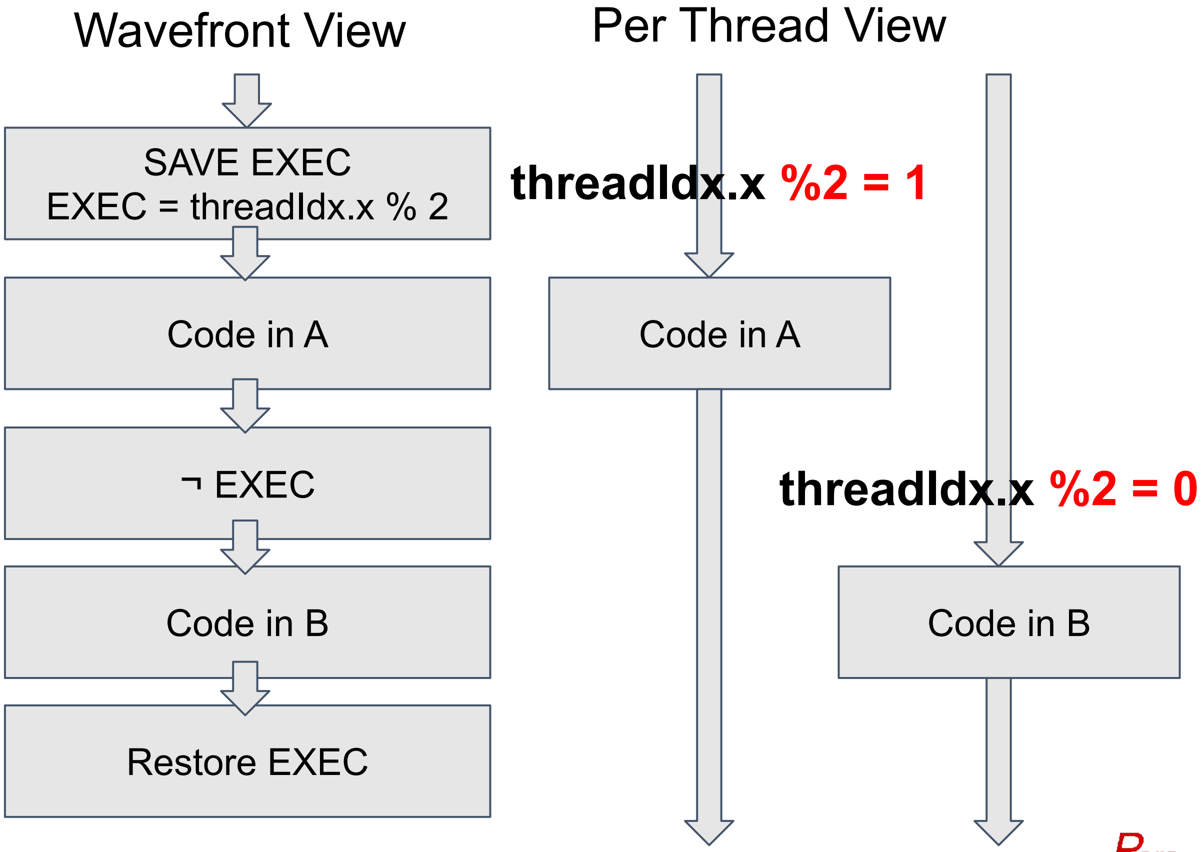
```
IF (threadIdx.x % 2)
{

   A; // odd threads

} ELSE  {

   B; // even threads

}
```



Control Flow Analysis for AMDGPU Kernels

Capture and represent per-thread control flow compactly

- Threads with the same execution mask shares the same CFG
  - The exact value of an execution mask *might be* only known at runtime
  - Rely on symbolic analysis to determine the set of equivalence classes of the symbolic execution mask value (when possible)
- For each predicated control flow construct
  - Generate a per-thread control flow graph per equivalence classes of the symbolic execution mask
  - In the cases where all threads have the same mask, this looks like a traditional CFG

Data Flow Analysis for AMDGPU Kernels

Data flow analysis needs to capture SIMD/SIMT data dependencies accurately

- Threads can be disabled by the execution mask
- For each instruction
  - Only active threads contribute to the data flow
  - Vector registers can be partially updated by active threads
  - The DEF and USE set of instructions need to expand to individual lanes of a vector register

```
int index = threadIdx.x;
if (index%2) {
    v[index] = index + 10;
} else {
    v[index] = index * 20;
}
```

```
vcc = if(index%2)
s_and_saveexec_b64 s[0:1], vcc
v_add_u32_e32 v1, 10, v0
s_xor_b64 exec, exec, s[0:1]
v_mul_u32_u24_e32 v1, 20, v0
...
```

Data Flow Analysis for AMDGPU Kernels

Capture In-Lane dataflow compactly

- In-lane operations have the same data flow for all threads

Capture Cross-Lane & Predicated dataflow accurately

- Vector instructions under predicated control flow
- Permutation instructions can introduce dependencies between any two lanes of two vector registers
  - Need to track the dependencies between all lanes

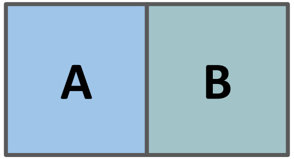
Design and Implementation of Efficient AMDGPU Binary Instrumentation

Instrumentation - Accessing Instrumentation Variables

- Insert instructions to observe the behavior of the kernel
  - Insert instructions to bookkeep execution in instrumentation variables
    - Counters / Timestamps / Tracing
- Data eventually needs to be available on the host
  - Simplest: at the end of kernel execution. Effective for counters and timers.
  - More complex: periodically at initiative of the device. Allows counters and times to be viewed over time on the host. Allows trace buffers to be flushed when filled.
  - Perhaps more complex: periodically at initiative of the host. Allows interactive tools to poll current state of the instrumentation results.

Instrumentation variables: per kernel launch

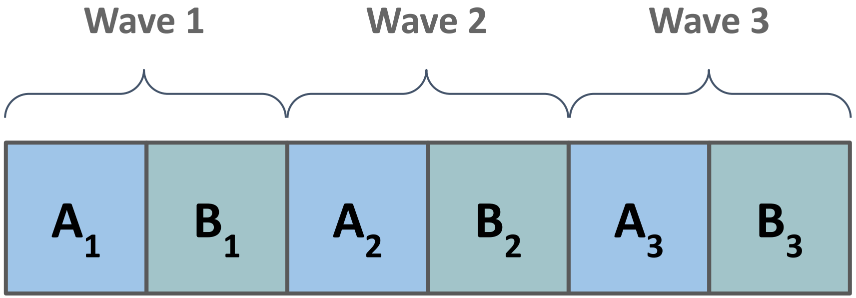
Two regular scalar variables A and B



Count cumulative information across waves

Requires synchronization or atomic operations as all waves write to the same locations

Instrumentation variables: per wavefront



Each wave has own instance of the scalar variable

No synchronization need

Measure per-wave information on device and aggregate if necessary

Instrumentation variables: per thread

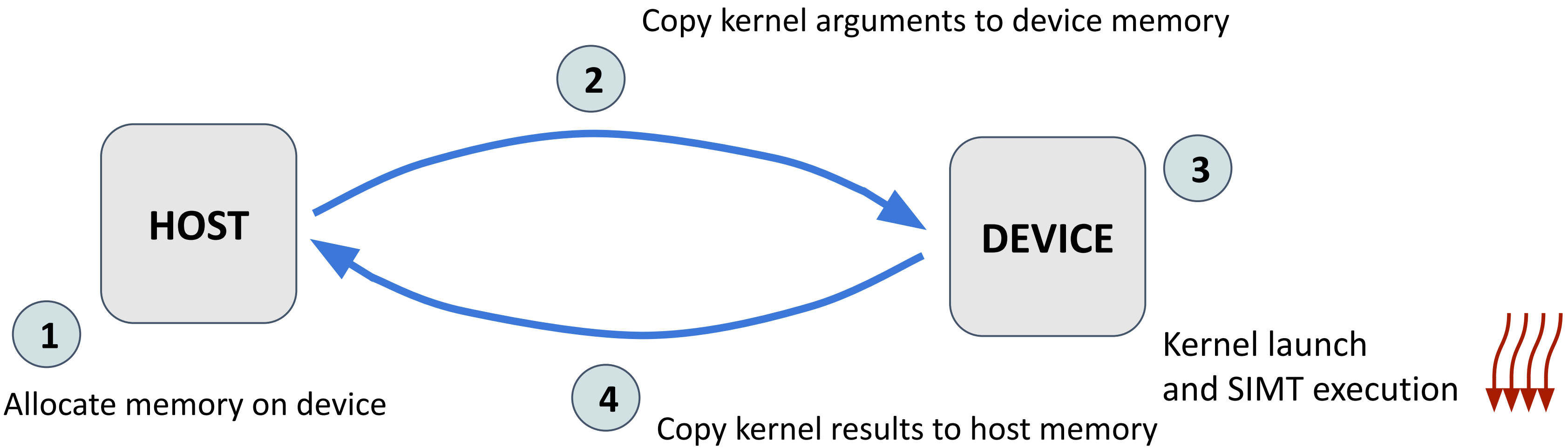
Wave 1		Wave 2		Wave 3	
$A_{1,1}$	$B_{1,1}$	$A_{2,1}$	$B_{2,1}$	$A_{3,1}$	$B_{3,1}$
$A_{1,2}$	$B_{1,2}$	$A_{2,1}$	$B_{2,2}$	$A_{3,2}$	$B_{3,2}$
$A_{1,3}$ … $A_{1,64}$	$B_{1,3}$ … $B_{1,64}$	$A_{2,3}$ … $A_{2,64}$	$B_{2,3}$ … $B_{2,64}$	$A_{2,3}$ … $A_{2,64}$	$B_{3,3}$ … $B_{3,64}$

Each thread has own its instance of the scalar variable (so, a vector per wave)

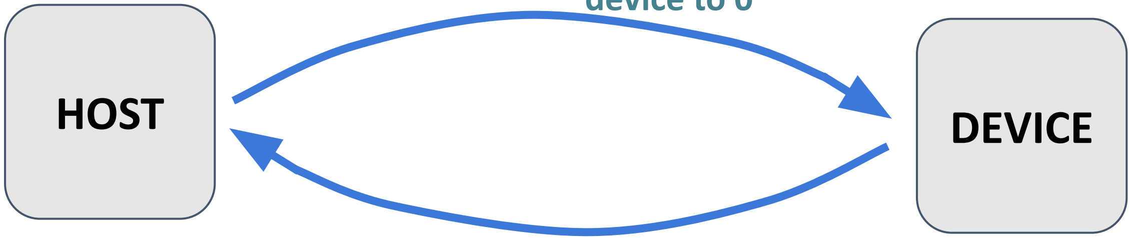
Computations on thread-level variables use vector operations

Only active threads read/write to their instances

Mechanics of offloading on the device (GPU)



Host-side implementation for Instrumentation Variables



1

Allocate memory on device

Allocate additional memory for instrumentation variables

2

Copy kernel arguments to device memory

Initialize instrumentation variables on device to 0

3

Pass instrumentation variables as an additional argument

Kernel launch
and SIMT execution

4

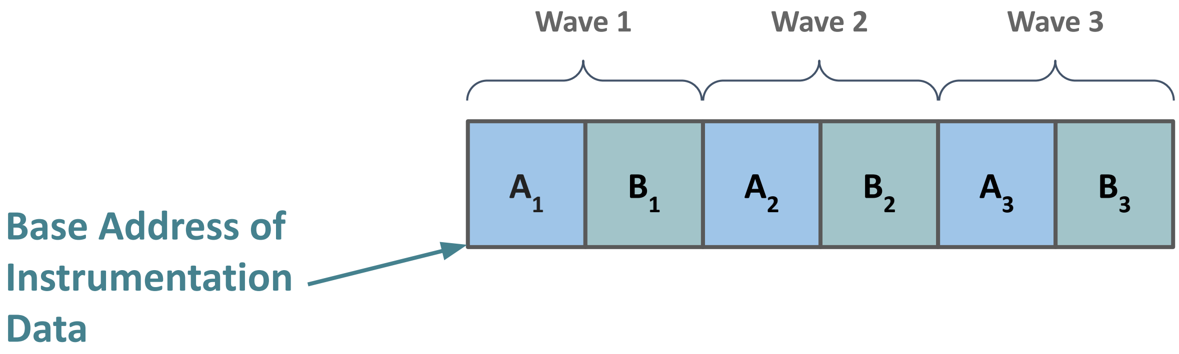
Copy kernel results to host memory

Copy instrumentation variables to host

We modify kernel signature to take extra argument
The extra argument is the address for instrumentation variables

Device-side implementation for Instrumentation Variables

Example for accessing wave-level instrumentation variables

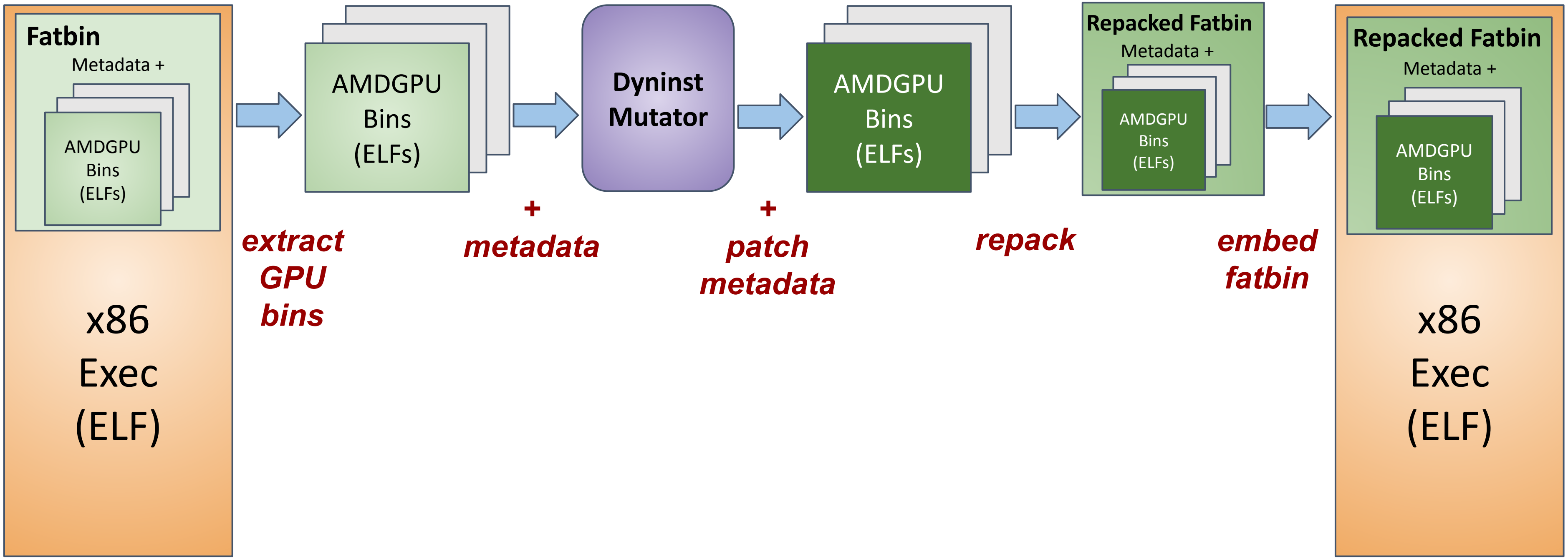


For each wave, calculate and store base address for its variables once in a register

Offsets for instances are same across waves

Therefore base + offset addressing will work at granularity of individual instances

Instrumenting AMDGPU code objects



Current Implementation Status

Control Flow Analysis

- Wavefront-level control flow analysis

Dataflow analysis

- Experimental support for instruction semantics and symbolic execution based on AMD's machine readable instruction semantic specification

Instrumentation

- Initial support for instrumentation at the granularity of a kernel launch
- Support tools
  - Tools that work alongside the Dyninst mutator for AMDGPU
  - Alpha level of readiness

Next Steps

Develop per-thread control flow analyses

Develop per-thread data flow analyses

Testing infrastructure for the instrumentation tools to make it ready for general use

Complete support for wave-level instrumentation

Extend instrumentation to thread level granularity

Questions ?

hwu337@wisc.edu ronak@cs.wisc.edu