Warp-level CFG construction for GPU kernel WCET analysis

L. Jeanmougin[1] P. Sotin[2] C. Rochange[1] T. Carle[1]

[1]IRIT - Univ. Toulouse 3 - CNRS, France

[2]IRIT - Univ. Toulouse 2 - CNRS, France

WCET workshop, July 2023

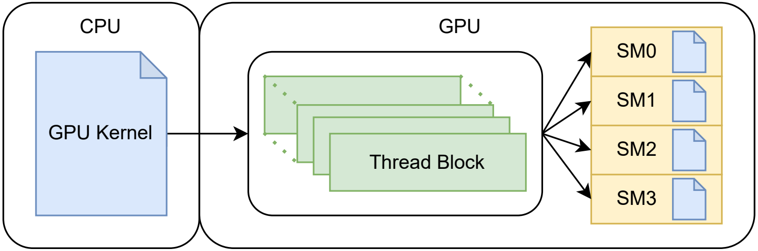
Institut de Recherche
en Informatique de Toulouse
CNRS - INP - UT3 - UT1 - UT2J

agence nationale de la recherche

Motivation

- Graphical Processing Units combine massive parallelism and versatility
- Embedded systems could benefit from the high throughput of GPUs (e.g. Autonomous Vehicles)
- Safe WCET calculation techniques are required for GPUs to be used in safety-critical real-time systems
- The WCET techniques for GPUs are still immature

SIMT execution semantics

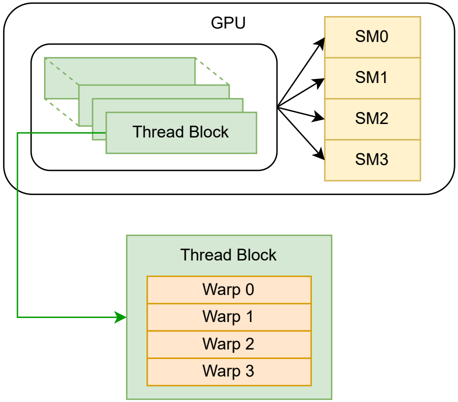


SM : Streaming Multiprocessor

Offloading of a kernel program from CPU to GPU

- GPU programs are called kernels
- The CPU requests the execution of a kernel to the GPU

SIMT execution semantics



Thread Block composition

Thread Block:

- Dispatched to one SM
- Composed of one to multiple warps

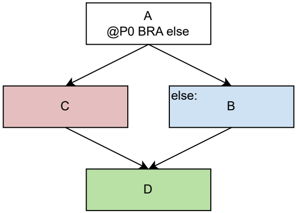
Warp:

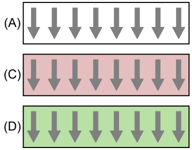
- Fixed number of threads
- Unique Program Counter
- Lockstep execution
- Smallest schedulable unit

SIMT execution semantics

Some source code

```
int i = tid.x;
if(i < n)
    /* do block C */
else
    /* do block B */
/* do block D */
```



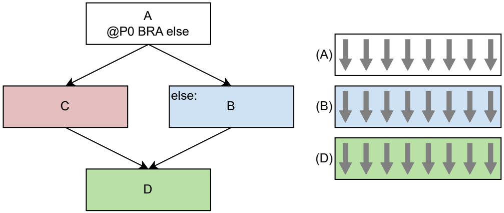


Uniform control flow (if)

SIMT execution semantics

Some source code

```
int i = tid.x;
if(i < n)
    /* do block C */
else
    /* do block B */
/* do block D */
```

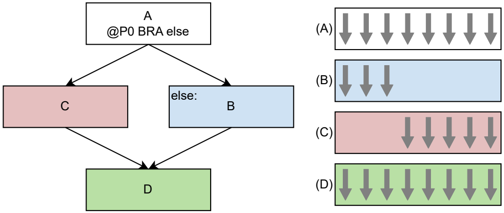


Uniform control flow (else)

SIMT execution semantics

Some source code

```
int i = tid.x;
if(i < n)
    /* do block C */
else
    /* do block B */
/* do block D */
```



Divergent control flow

Contributions

A CFG building technique to describe the execution of a warp :

- Description of the thread divergence on NVIDIA Pascal GPUs
- Analyse the Pascal SASS in order to derive a Warp-level CFG
- Value agreement analysis interleaved with the CFG production

Goal : Ability to apply the IPET method on the generated CFG

Concrete activation stack

Address	Disassembly	
0x38	...	A
0x40	SSY TARGET2;	
0x48	@P0 BRA TARGET1;	
0x50	...;	B
	...	
0x80	SYNC;	
0x88	TARGET1:	C
	...	
0xA0	SYNC;	
0xA8	TARGET2:	D
	...	

SASS Example

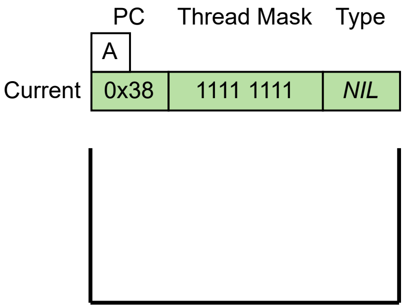
Some source code

```
int i = tid.x;
if(i < n)
    /* do block C */
else
    /* do block B */
/* do block D */
```

Concrete activation stack

Address	Disassembly	
0x38	...	A
0x40	SSY TARGET2;	
0x48	@P0 BRA TARGET1;	
0x50	...;	B
	...	
0x80	SYNC;	
0x88	TARGET1:	C
	...	
0xA0	SYNC;	
0xA8	TARGET2:	D
	...	

SASS Example

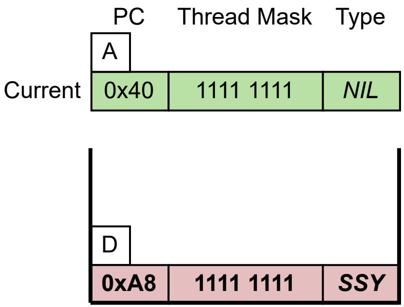


Concrete activation stack state

Concrete activation stack

Address	Disassembly	
0x38	...	A
0x40	SSY TARGET2;	
0x48	@P0 BRA TARGET1;	
0x50	...;	B
	...	
0x80	SYNC;	
0x88	TARGET1:	C
	...	
0xA0	SYNC;	
0xA8	TARGET2:	D
	...	

SASS Example

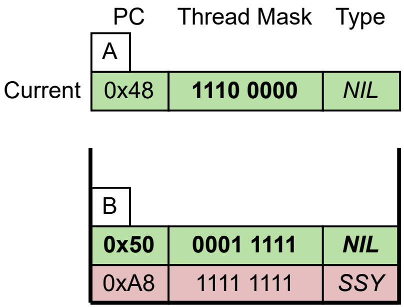


Concrete activation stack state

Concrete activation stack

Address	Disassembly	
0x38	...	A
0x40	SSY TARGET2;	
0x48	@P0 BRA TARGET1;	
0x50	...;	B
	...	
0x80	SYNC;	
0x88	TARGET1:	C
	...	
0xA0	SYNC;	
0xA8	TARGET2:	D
	...	

SASS Example

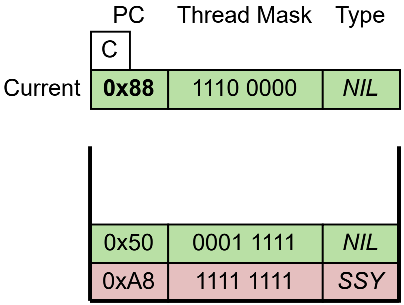


Concrete activation stack state

Concrete activation stack

Address	Disassembly	
0x38	...	A
0x40	SSY TARGET2;	
0x48	@P0 BRA TARGET1;	
0x50	...;	B
	...	
0x80	SYNC;	
0x88	TARGET1:	C
	...	
0xA0	SYNC;	
0xA8	TARGET2:	D
	...	

SASS Example



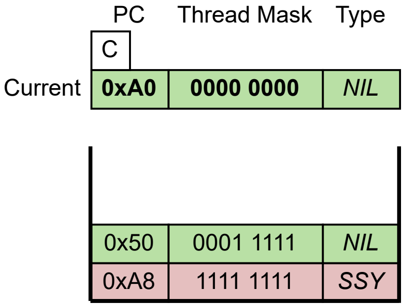
	PC	Thread Mask	Type
	C		
Current	**0x88**	1110 0000	*NIL*
	0x50	0001 1111	*NIL*
	0xA8	1111 1111	*SSY*

Concrete activation stack state

Concrete activation stack

Address	Disassembly	
0x38	...	A
0x40	SSY TARGET2;	
0x48	@P0 BRA TARGET1;	
0x50	...;	B
	...	
0x80	SYNC;	
0x88	TARGET1:	C
	...	
0xA0	SYNC;	
0xA8	TARGET2:	D
	...	

SASS Example



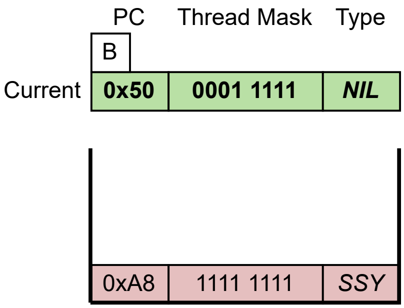
	PC	Thread Mask	Type
Current	**0xA0**	**0000 0000**	*NIL*
	0x50	0001 1111	*NIL*
	0xA8	1111 1111	*SSY*

Concrete activation stack state

Concrete activation stack

Address	Disassembly	
0x38	...	A
0x40	SSY TARGET2;	
0x48	@P0 BRA TARGET1;	
0x50	...;	B
	...	
0x80	SYNC;	
0x88	TARGET1:	C
	...	
0xA0	SYNC;	
0xA8	TARGET2:	D
	...	

SASS Example

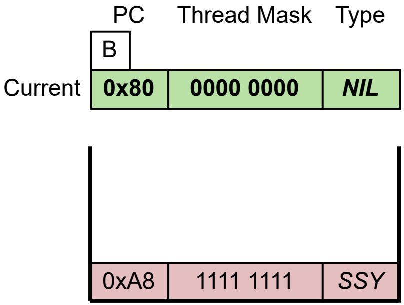


Concrete activation stack state

Concrete activation stack

Address	Disassembly	
0x38	...	A
0x40	SSY TARGET2;	
0x48	@P0 BRA TARGET1;	
0x50	...;	B
	...	
0x80	SYNC;	
0x88	TARGET1:	C
	...	
0xA0	SYNC;	
0xA8	TARGET2:	D
	...	

SASS Example

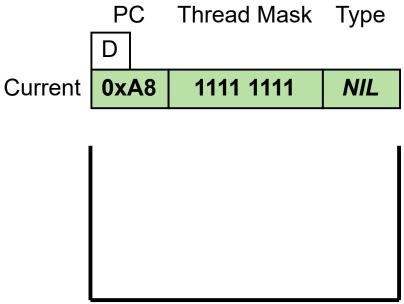


Concrete activation stack state

Concrete activation stack

Address	Disassembly	
0x38	...	A
0x40	SSY TARGET2;	
0x48	@P0 BRA TARGET1;	
0x50	...;	B
	...	
0x80	SYNC;	
0x88	TARGET1:	C
	...	
0xA0	SYNC;	
0xA8	TARGET2:	D
	...	

SASS Example



	PC	Thread Mask	Type
	D		
Current	**0xA8**	**1111 1111**	***NIL***

Concrete activation stack state

Abstract activation stack

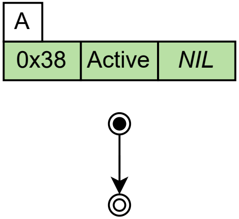
Processing all the possible concrete thread masks would be too costly.
Solution:

- Consider abstract thread groups
- Only register the relations between these groups

Abstract activation stack

Address	Disassembly	
0x38	...	A
0x40	SSY TARGET2;	
0x48	@P0 BRA TARGET1;	
0x50	...;	B
	...	
0x80	SYNC;	
0x88	TARGET1:	C
	...	
0xA0	SYNC;	
0xA8	TARGET2:	D
	...	

SASS Example

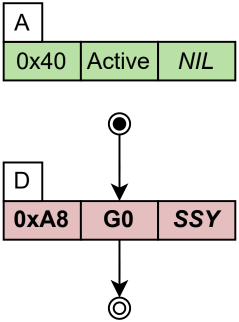


Abstract activation stack state

Abstract activation stack

Address	Disassembly	
0x38	...	A
0x40	SSY TARGET2;	
0x48	@P0 BRA TARGET1;	
0x50	...;	B
	...	
0x80	SYNC;	
0x88	TARGET1:	C
	...	
0xA0	SYNC;	
0xA8	TARGET2:	D
	...	

SASS Example

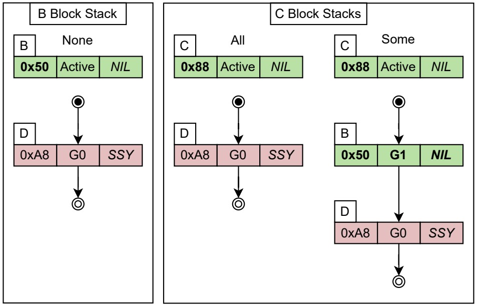


Abstract activation stack state

Abstract activation stack

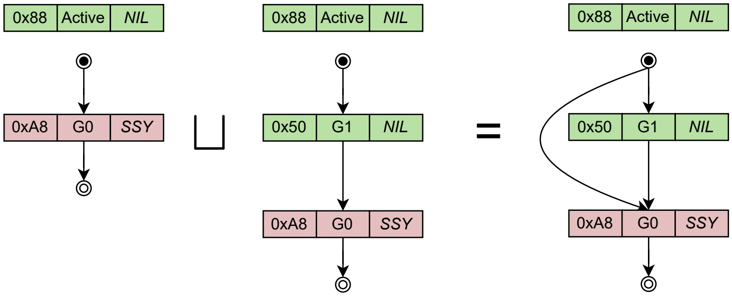
Address	Disassembly	
0x38	...	A
0x40	SSY TARGET2;	
0x48	@P0 BRA TARGET1;	
0x50	...;	B
	...	
0x80	SYNC;	
0x88	TARGET1:	C
	...	
0xA0	SYNC;	
0xA8	TARGET2:	D
	...	

SASS Example



Abstract activation stack states

Abstract activation stack

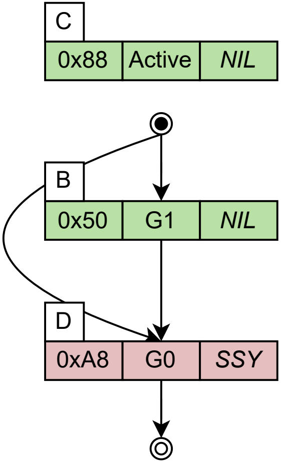


Union operation on abstract stack states

Abstract activation stack

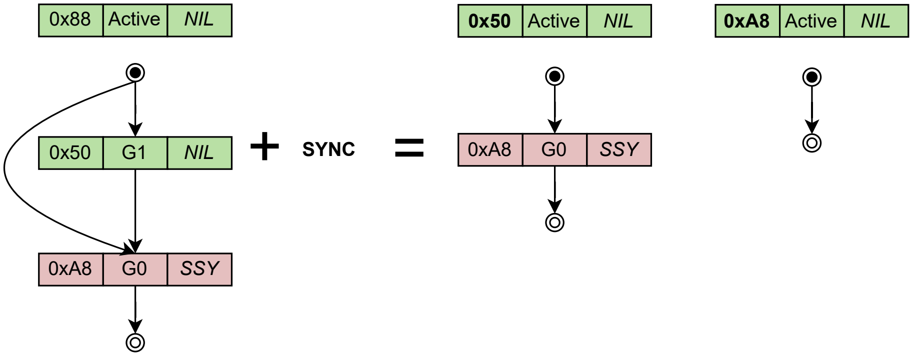
Address	Disassembly	
0x38	...	A
0x40	SSY TARGET2;	
0x48	@P0 BRA TARGET1;	
0x50	...;	B
	...	
0x80	SYNC;	
0x88	TARGET1:	C
	...	
0xA0	SYNC;	
0xA8	TARGET2:	D
	...	

SASS Example



Abstract activation stack state

Abstract activation stack

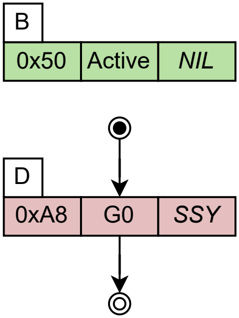


SYNC effect on abstract activation stack

Abstract activation stack

Address	Disassembly	
0x38	...	$\boxed{\text{A}}$
0x40	SSY TARGET2;	
0x48	@P0 BRA TARGET1;	
0x50	...;	$\boxed{\text{B}}$
	...	
0x80	SYNC;	
0x88	TARGET1:	$\boxed{\text{C}}$
	...	
0xA0	SYNC;	
0xA8	TARGET2:	$\boxed{\text{D}}$
	...	

SASS Example

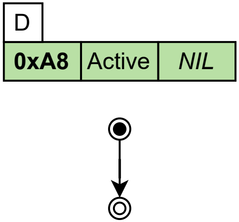


Abstract activation stack state

Abstract activation stack

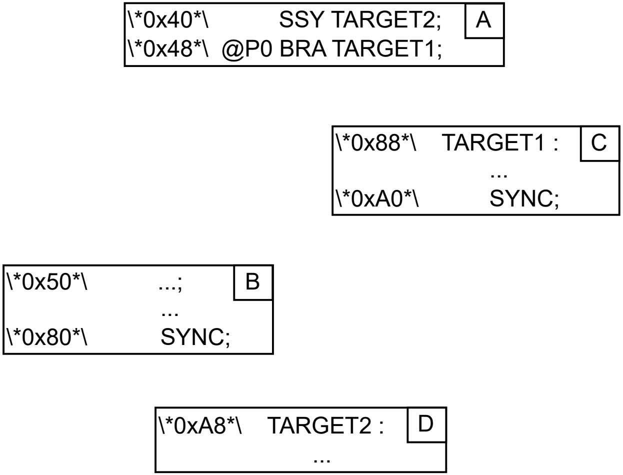
Address	Disassembly	
0x38	...	A
0x40	SSY TARGET2;	
0x48	@P0 BRA TARGET1;	
0x50	...;	B
	...	
0x80	SYNC;	
0x88	TARGET1:	C
	...	
0xA0	SYNC;	
0xA8	TARGET2:	D
	...	

SASS Example



Abstract activation stack state

Warp-level CFG construction



```
\*0x40*\          SSY TARGET2;   A
\*0x48*\  @P0 BRA TARGET1;
```

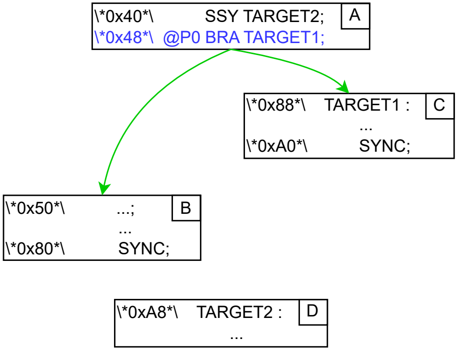
```
\*0x88*\    TARGET1 :   C
                 ...
\*0xA0*\          SYNC;
```

```
\*0x50*\          ...;      B
                  ...
\*0x80*\          SYNC;
```

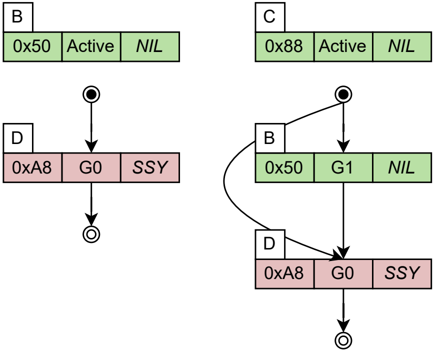
```
\*0xA8*\    TARGET2 :   D
                 ...
```

Kernel's basic blocks

Warp-level CFG construction

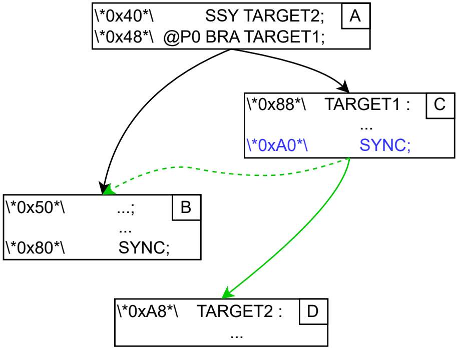


CFG state

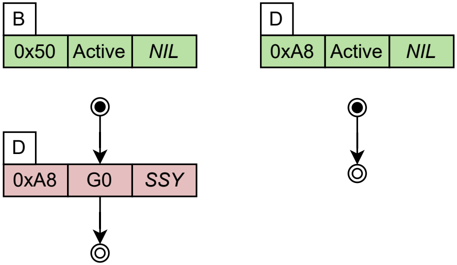


Explored abstract stack state

Warp-level CFG construction

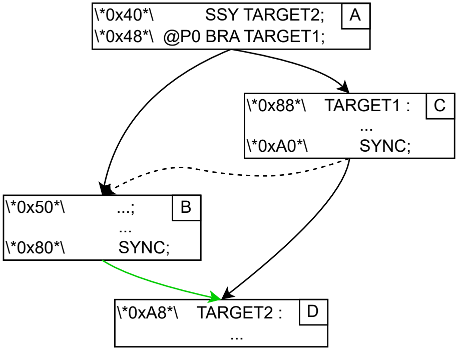


CFG state

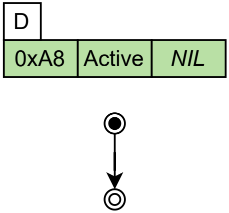


Explored abstract stack state

Warp-level CFG construction



CFG state



Explored abstract stack state

Abstract value agreement

We want to obtain a more precise CFG:

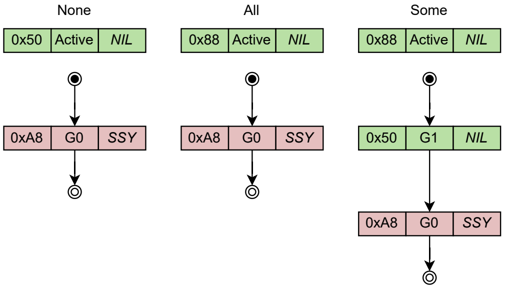
- All threads have their own registers
- Some registers depend on identical data
- Knowing the relation between threads predicates would help

Abstract value agreement

Address	Disassembly
0x48	@P0 BRA TARGET1;

Refine the analysis by:

- Keeping track of the thread groups agreement
- Agreement on all registers including predicates



Abstract activation stack states

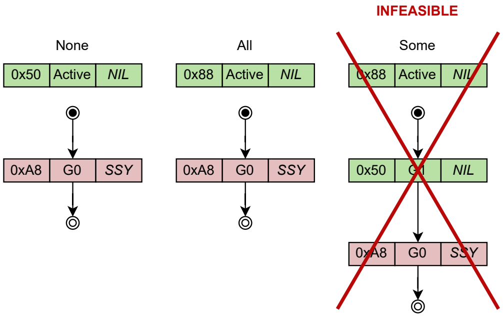
Can we know if all threads agree on P0 ?

Abstract value agreement

Address	Disassembly
0x48	@P0 BRA TARGET1;

Refine the analysis by:

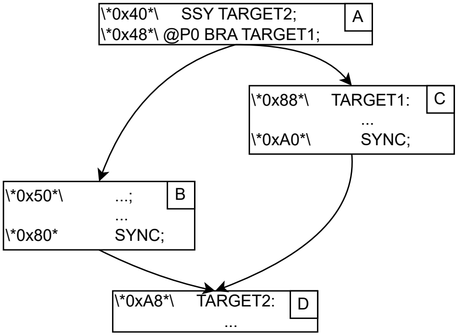
- Keeping track of the thread groups agreement
- Agreement on all registers including predicates



Abstract activation stack states

All threads agreed on the values of the registers used to calculate P0

Abstract value agreement



CFG without infeasible divergence

- Corresponds to the CFG of a thread in isolation
- More precise control flow of a warp

Evaluation

Evaluation of the analysis method was made on kernels extracted from the GPU Benchmark "Rodinia"

- 37 out of 57 kernels have been tested
  - Activation stack behavior might be different for calls
- For each tested kernel a CFG was successfully generated
- An ILP system following the IPET method was performed on each produced CFG
- Without value agreement half of the kernels' WCET are severely degraded (estimation x10 or more)

Future work

- Improve the analysis by supporting calls
- Adapt classic analyses to our framework (e.g. loop bound analysis)
- Strengthen our knowledge on GPU micro-architecture to derive precise duration for the CFG's blocks
- Find benchmarks with more divergence