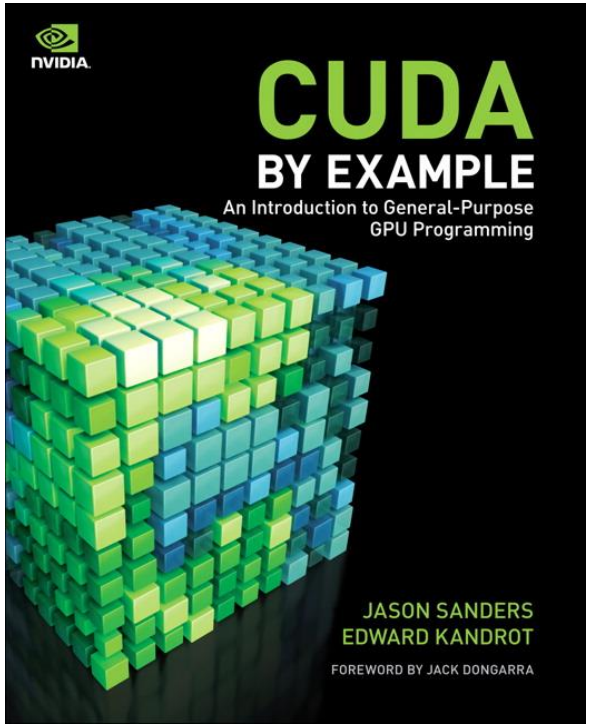
Exposing Errors Related to Weak Memory in GPU Applications

Tyler Sorensen
Imperial College London

Supervisor: Alastair F. Donaldson
PLDI 2016

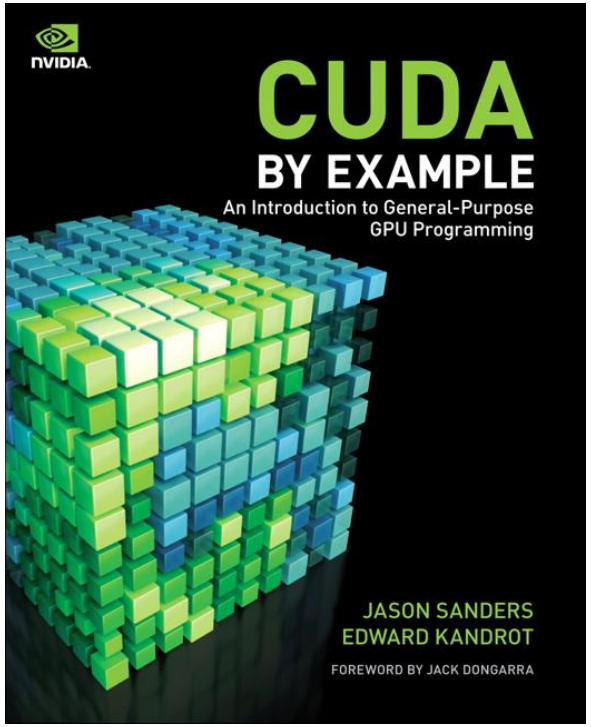
Example

- GPU dot product



Example

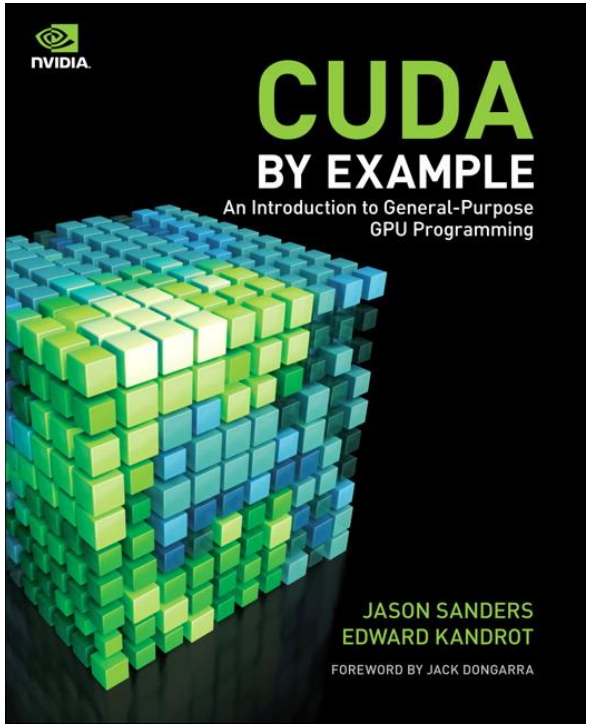
- GPU dot product




```
__global__ void dot(int *mutex, float *a, float *b, float *c) {
   int tid = threadIdx.x + blockIdx.x * blockDim.x;
   float temp = 0;
   while (tid < N) {
      temp += a[tid] * b[tid];
      tid += blockDim.x * gridDim.x;
   }

   // local computation code omitted

   if (threadIdx.x== 0) {
     lock(mutex);
     *c += cache[0];
     unlock(mutex);
   }
}

__device__ void lock(int *mutex) {
 while (atomicCAS(l, 0, 1) != 0 );
}

 __device__ void unlock(int *mutex) {
 atomicExch(l, 0);
}
```

Example

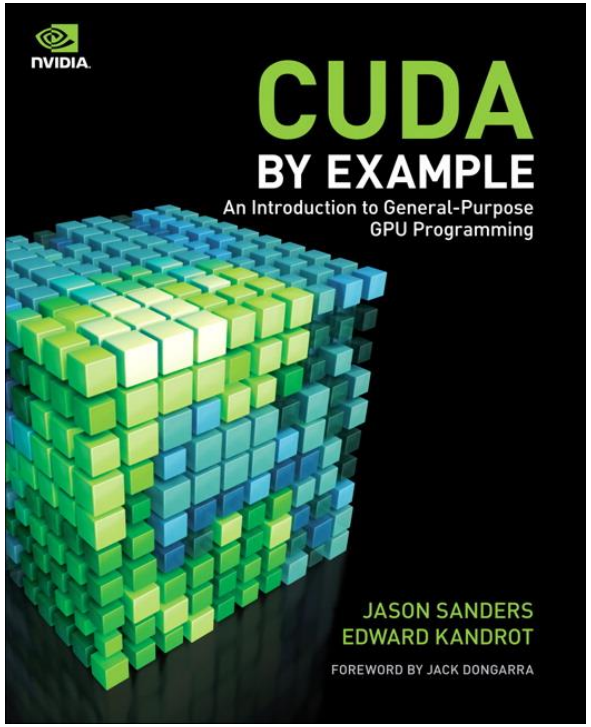
- GPU dot product



- Testing the code:
  - Run for 1 hour (~2 seconds per run) and check for errors

Example

- GPU dot product

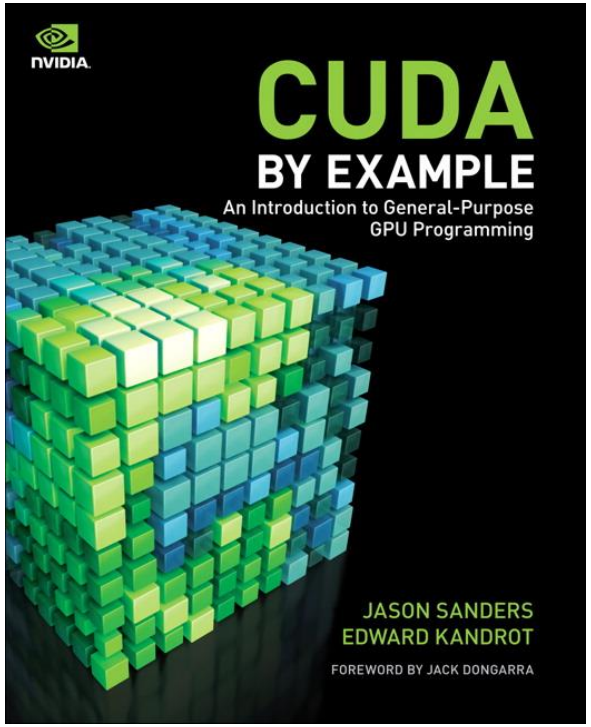


- Testing the code:
  - Run for 1 hour (~2 seconds per run) and check for errors

No errors observed!
Code is probably correct right?

Example

- GPU dot product



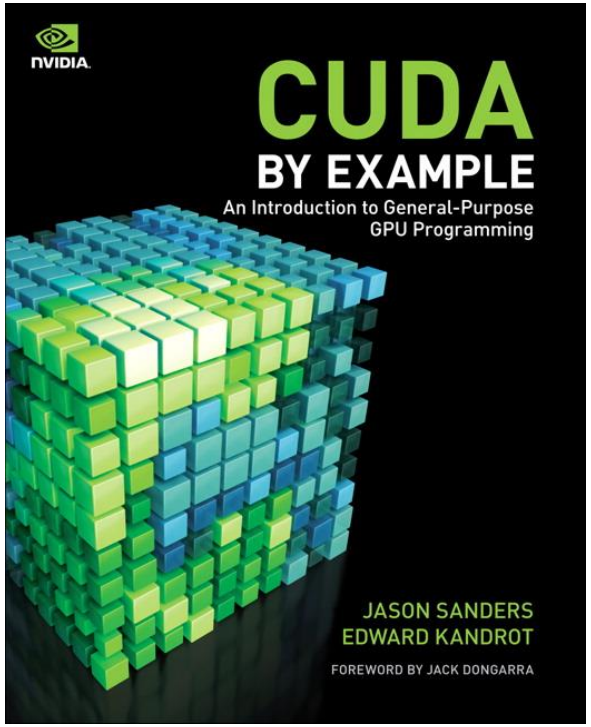
- Testing the code:
  - Run for 1 hour (~2 seconds per run) and check for errors

No errors observed!
Code is probably correct right?

Wrong! Weak memory bug reported in previous work

Example

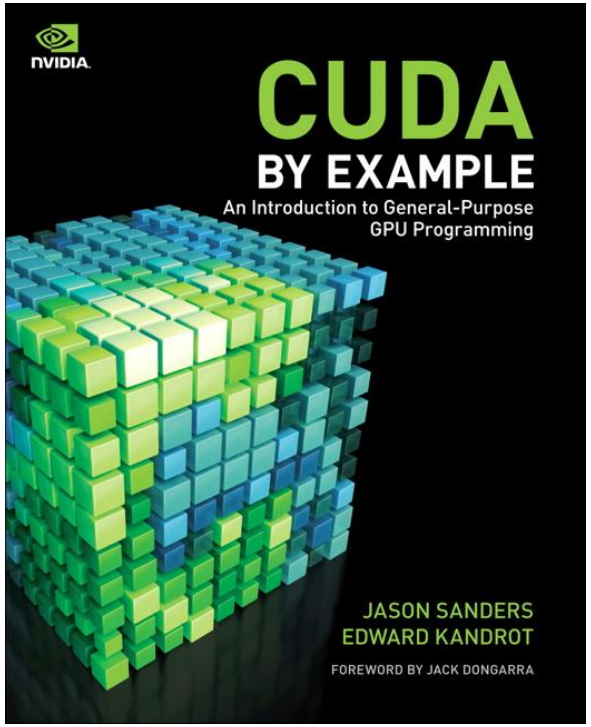
- GPU dot product



- Previous bug found through hand analysis by experts
- This is laborious and no guarantees

Example

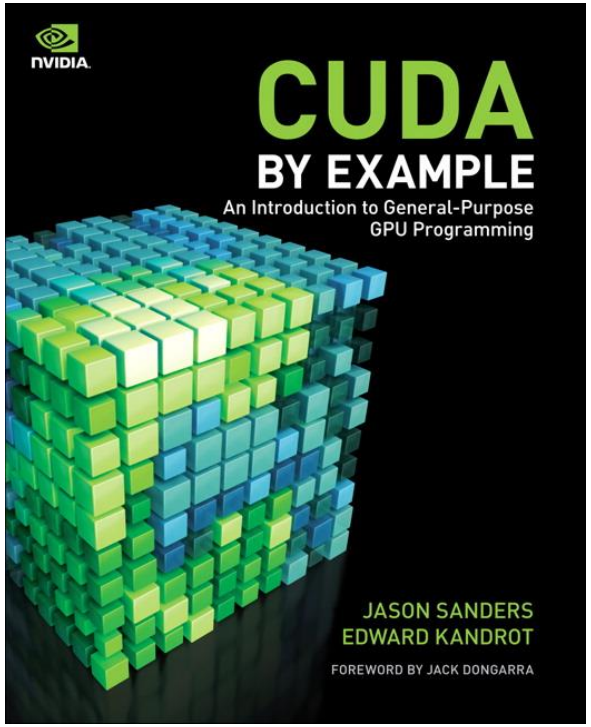
- GPU dot product



- **Our goals:**

Example

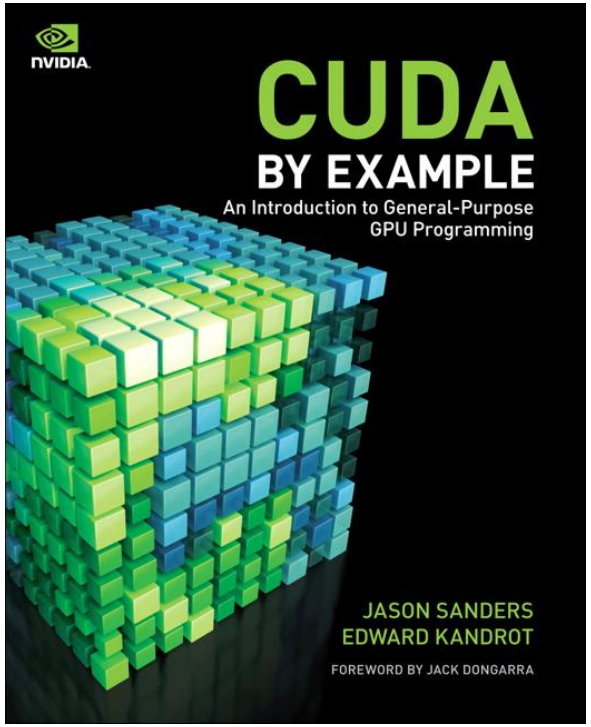
- GPU dot product



- **Our goals:**
  - Create a method for programmers to find bugs

Example

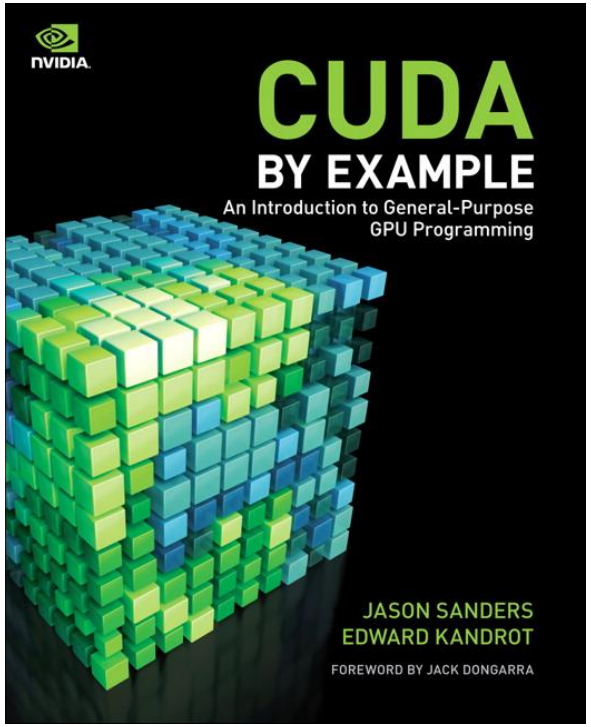
- GPU dot product



- **Our goals:**
  - Create a method for programmers to find bugs
  - Automatically suggest bug fixes

Example

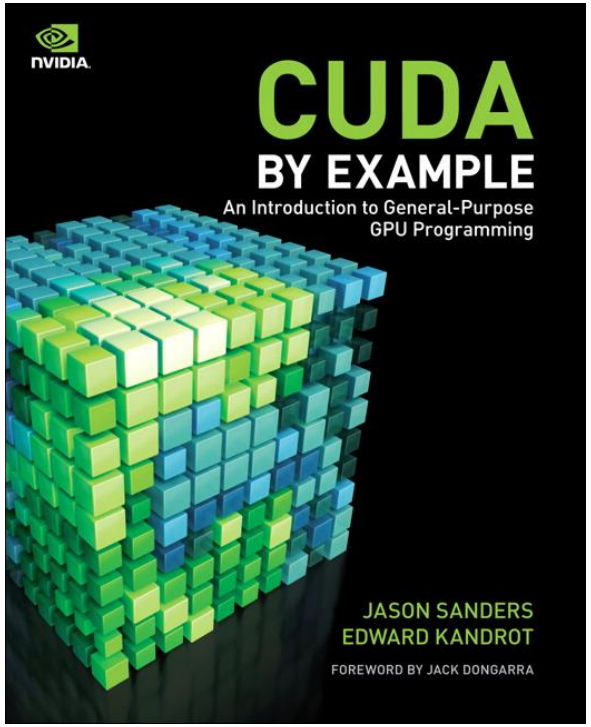
- GPU dot product



- Developed a stress/fuzz testing environment
  - Run for 1 hour (~2 seconds per run) and check for errors

Example

- GPU dot product

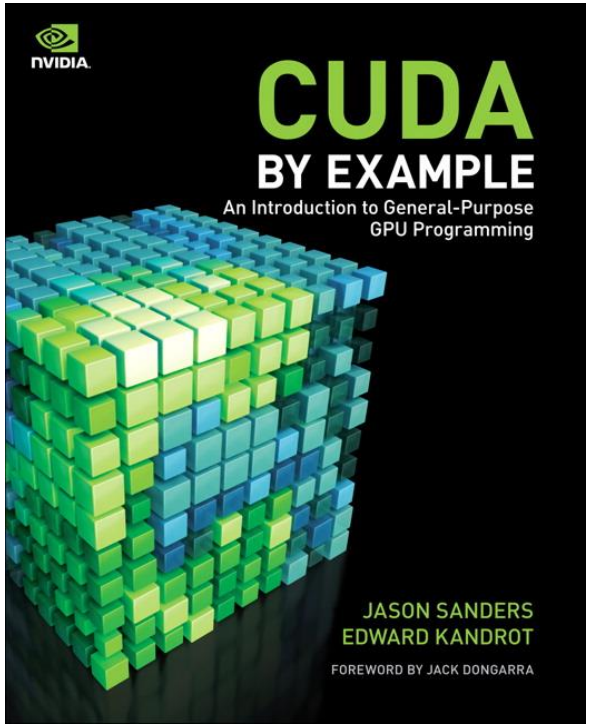


- Developed a stress/fuzz testing environment
  - Run for 1 hour (~2 seconds per run) and check for errors

396 erroneous runs observed

Example

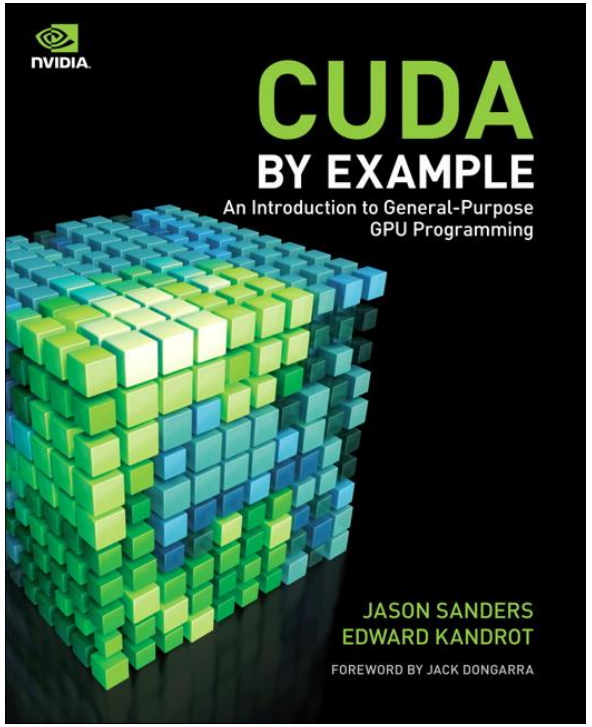
- GPU dot product



- How to fix the bug?

Example

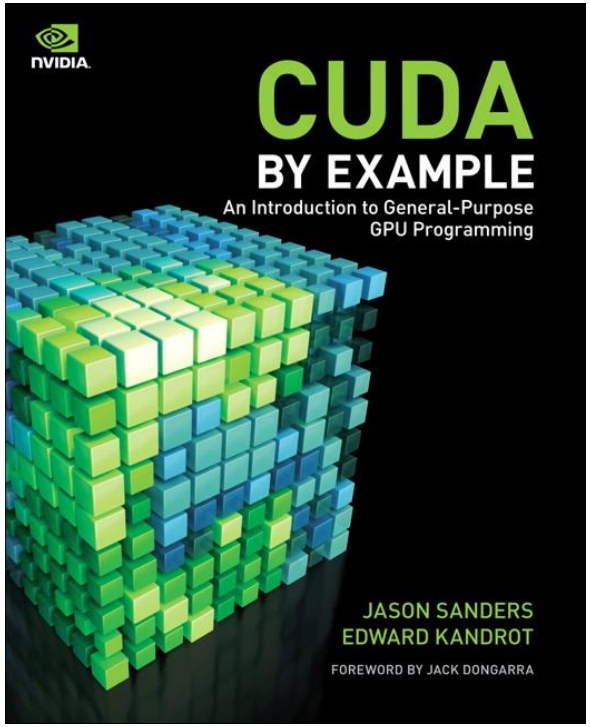
- GPU dot product




```
__global__ void dot(int *mutex, float *a, float *b, float *c) {
   int tid = threadIdx.x + blockIdx.x * blockDim.x;
   float temp = 0;
   while (tid < N) {
      temp += a[tid] * b[tid];
      tid += blockDim.x * gridDim.x;
   }

   // local computation code omitted

   if (threadIdx.x== 0) {
     lock(mutex);
     *c += cache[0];
     unlock(mutex);
   }
}

__device__ void lock(int *mutex) {
 while (atomicCAS(l, 0, 1) != 0 );
}

 __device__ void unlock(int *mutex) {
 atomicExch(l, 0);
}
```

Example

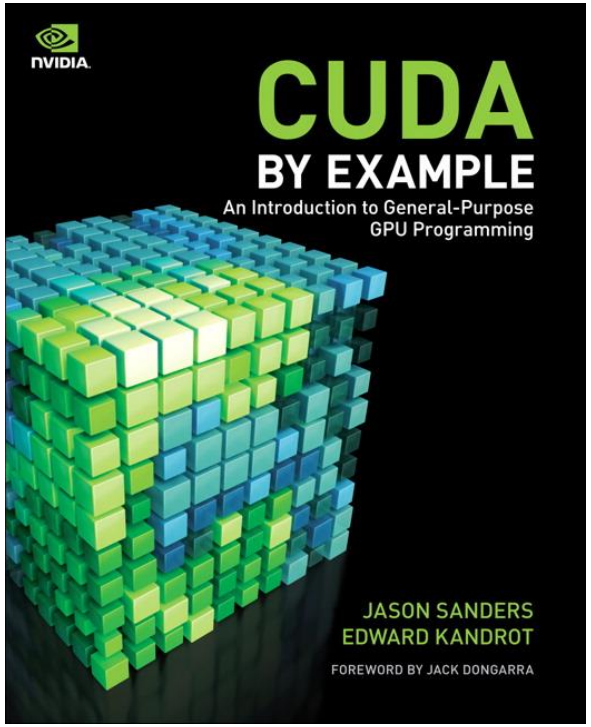
- GPU dot product




```
__global__ void dot(int *mutex, float *a, float *b, float *c) {
   int tid = threadIdx.x + blockIdx.x * blockDim.x;
   float temp = 0;
   while (tid < N) {    threadfence();
      temp += a[tid] * b[tid];
      tid += blockDim.x * gridDim.x;
   }

   // local computation code omitted

   if (threadIdx.x== 0) {
     lock(mutex);    threadfence();
     *c += cache[0];
     unlock(mutex);
   }
}

__device__ void lock(int *mutex) {    threadfence();
 while (atomicCAS(l, 0, 1) != 0 ) threadfence();
}

 __device__ void unlock(int *mutex) {threadfence();
 atomicExch(l, 0);
}
```

Example

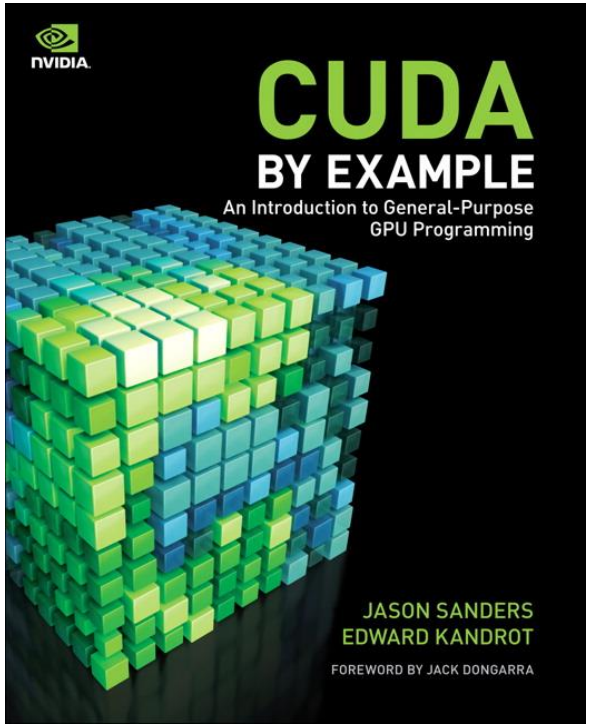
- GPU dot product



- Developed a stress/fuzz testing framework
  - Run for 1 hour (~2 seconds per run) and check for errors

Example

- GPU dot product



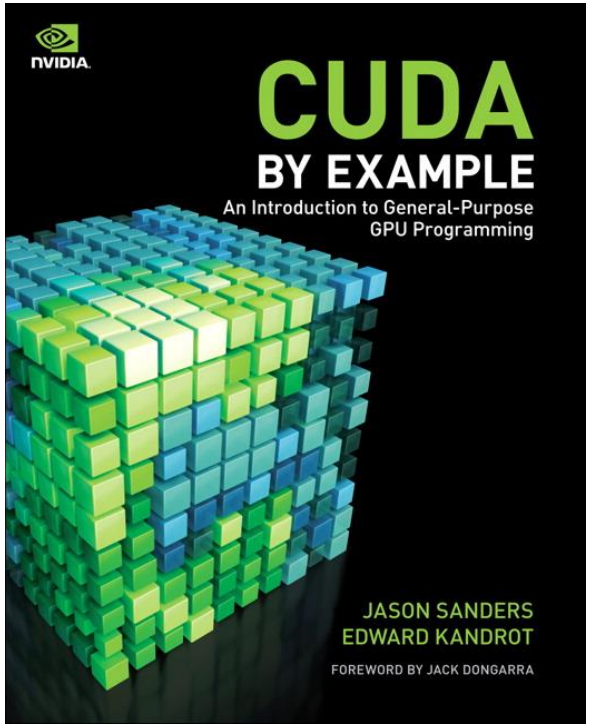
- Developed a stress/fuzz testing framework
  - Run for 1 hour (~2 seconds per run) and check for errors

0 erroneous runs observed

Did we fix it?

Example

- GPU dot product



0 erroneous runs observed

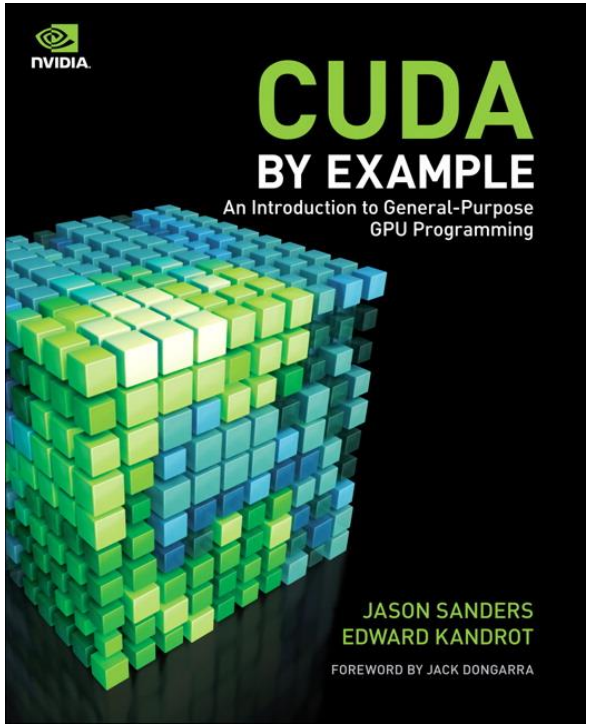
Did we fix it?

- Fences cost performance overhead:

60% runtime and energy overhead

Example

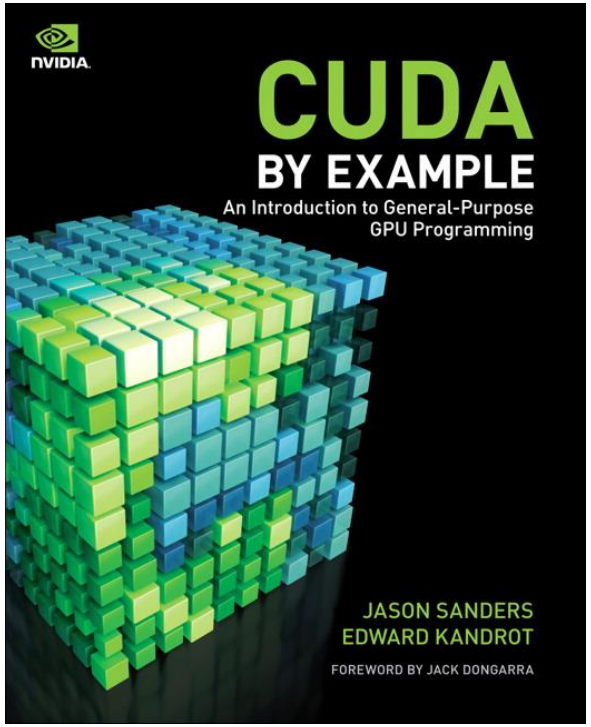
- GPU dot product




```
__global__ void dot(int *mutex, float *a, float *b, float *c) {
   int tid = threadIdx.x + blockIdx.x * blockDim.x;
   float temp = 0;
   while (tid < N) {    threadfence();
      temp += a[tid] * b[tid];
      tid += blockDim.x * gridDim.x;
   }

   // local computation code omitted

   if (threadIdx.x== 0) {
     lock(mutex);    threadfence();
     *c += cache[0];
     unlock(mutex);
   }
}

__device__ void lock(int *mutex) {    threadfence();
 while (atomicCAS(l, 0, 1) != 0 ) threadfence();
}

 __device__ void unlock(int *mutex) {threadfence();
 atomicExch(l, 0);
}
```

Example

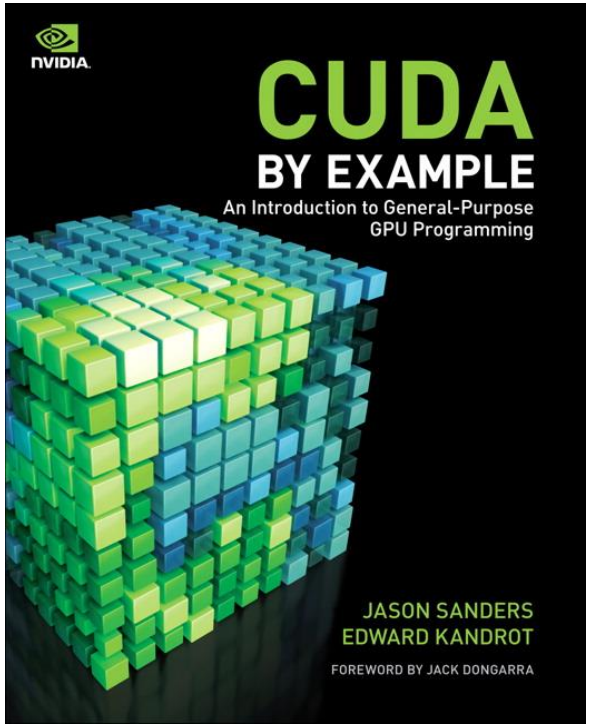
- GPU dot product




```
__global__ void dot(int *mutex, float *a, float *b, float *c) {
   int tid = threadIdx.x + blockIdx.x * blockDim.x;
   float temp = 0;
   while (tid < N) {
      temp += a[tid] * b[tid];
      tid += blockDim.x * gridDim.x;
   }

   // local computation code omitted

   if (threadIdx.x== 0) {
     lock(mutex);
     *c += cache[0];
     unlock(mutex);
   }
}

__device__ void lock(int *mutex) {
 while (atomicCAS(l, 0, 1) != 0 );
}

 __device__ void unlock(int *mutex) {  threadfence();
 atomicExch(l, 0);
}
```

Example

- GPU dot product



0 erroneous runs observed

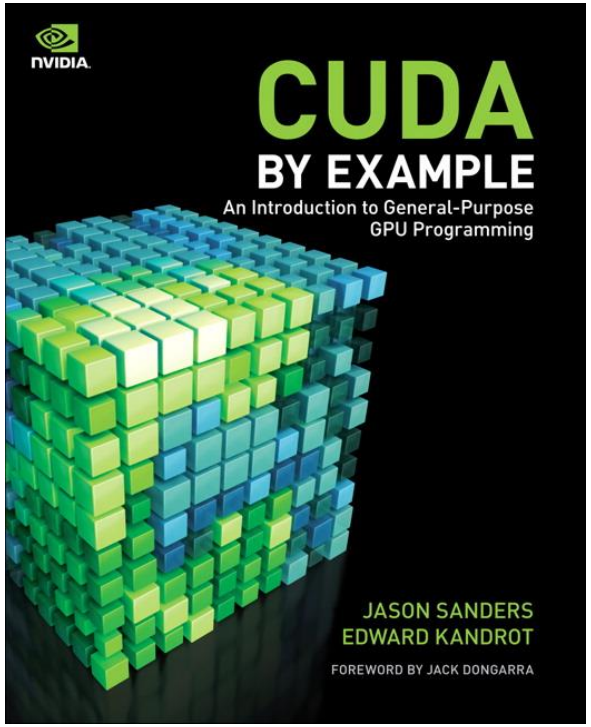
Did we fix it?

- Fences cost performance overhead:

60% runtime and energy overhead

Example

- GPU dot product



0 erroneous runs observed

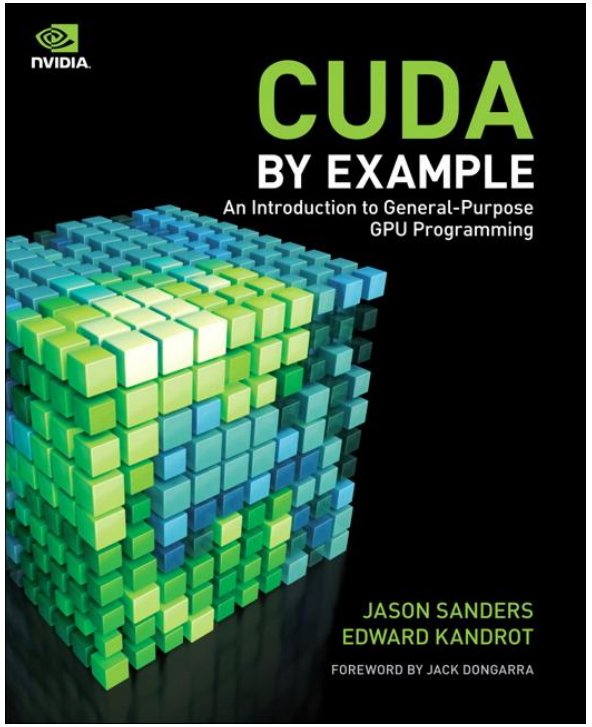
Did we fix it?

- Fences cost performance overhead:

Less than 3% runtime and energy overhead

Example

- GPU dot product



0 erroneous runs observed

Did we fix it?

- Fences cost performance overhead:

Empirically fixed, not formally fixed!

Roadmap

- Background
- Testing environment
- Environment evaluation
- Fence placement

Roadmap

- Background
- Testing environment
- Environment evaluation
- Fence placement

Weak memory models

- consider the test known as *message passing* (MP)

initial state: $x = 0, y = 0$	
Thread 0	Thread 1
`a: x ← 1;`	`c: r1 ← y;`
`b: y ← 1;`	`d: r2 ← x;`
assert: $r1 = 1 \wedge r2 = 0$	

Weak memory models

- consider the test known as *message passing* (MP)

initial state: $x = 0, y = 0$	
Thread 0	Thread 1
`a: x ← 1;`	`c: r1 ← y;`
`b: y ← 1;`	`d: r2 ← x;`
assert: $r1 = 1 \wedge r2 = 0$	

Weak memory models

- consider the test known as *message passing* (MP)

initial state: $x = 0, y = 0$	
Thread 0 a: x ← 1; b: y ← 1;	Thread 1 c: r1 ← y; d: r2 ← x;
assert: $r1 = 1 \wedge r2 = 0$	

Weak memory models

- consider the test known as *message passing* (MP)

initial state: $x = 0, y = 0$	
Thread 0	Thread 1
`a: x ← 1;`	`c: r1 ← y;`
`b: y ← 1;`	`d: r2 ← x;`
assert: $r1 = 1 \wedge r2 = 0$	

Message passing (MP) test

- Tests how to implement a handshake idiom

initial state: $x = 0, y = 0$	
Thread 0	Thread 1
a: $x \leftarrow 1$;	c: $r1 \leftarrow y$;
b: $y \leftarrow 1$;	d: $r2 \leftarrow x$;
assert: $r1 = 1 \wedge r2 = 0$	

Data

Data

Message passing (MP) test

- Tests how to implement a handshake idiom

initial state: $x = 0, y = 0$	
Thread 0	Thread 1
`a: x ← 1;`	`c: r1 ← y;`
`b: y ← 1;`	`d: r2 ← x;`
assert: $r1 = 1 \wedge r2 = 0$	

Flag

Flag

Message passing (MP) test

- Tests how to implement a handshake idiom

initial state: $x = 0, y = 0$	
Thread 0	Thread 1
`a: x ← 1;`	`c: r1 ← y;`
`b: y ← 1;`	`d: r2 ← x;`
assert: $r1 = 1 \wedge r2 = 0$ **Stale Data**	

initial state: $x = 0, y = 0$	
Thread 0	Thread 1
`a: x ← 1;`	`c: r1 ← y;`
`b: y ← 1;`	`d: r2 ← x;`
assert: $r1 = 1 \wedge r2 = 0$	

initial state: $x = 0, y = 0$	
Thread 0	Thread 1
`a: x ← 1;`	`c: r1 ← y;`
`b: y ← 1;`	`d: r2 ← x;`
assert: $r1 = 1 \wedge r2 = 0$	

Interleaving 1

```
a: x ← 1;
b: y ← 1;
c: r1 ← y;
d: r2 ← x;
```

Final: $r1 = 1 \wedge r2 = 1$

initial state: $x = 0, y = 0$	
Thread 0	Thread 1
`a: x ← 1;`	`c: r1 ← y;`
`b: y ← 1;`	`d: r2 ← x;`
assert: $r1 = 1 \wedge r2 = 0$	

Interleaving 1	**Interleaving 2**
`a: x ← 1;`	`a: x ← 1;`
`b: y ← 1;`	`c: r1 ← y;`
`c: r1 ← y;`	`b: y ← 1;`
`d: r2 ← x;`	`d: r2 ← x;`
Final: $r1 = 1 \wedge r2 = 1$	Final: $r1 = 0 \wedge r2 = 1$

initial state: $x = 0, y = 0$	
Thread 0	Thread 1
`a: x ← 1;`	`c: r1 ← y;`
`b: y ← 1;`	`d: r2 ← x;`
assert: $r1 = 1 \wedge r2 = 0$	

Interleaving 1	**Interleaving 2**	**Interleaving 3**
`a: x ← 1;`	`a: x ← 1;`	`a: x ← 1;`
`b: y ← 1;`	`c: r1 ← y;`	`c: r1 ← y;`
`c: r1 ← y;`	`b: y ← 1;`	`d: r2 ← x;`
`d: r2 ← x;`	`d: r2 ← x;`	`b: y ← 1;`
Final: $r1 = 1 \wedge r2 = 1$	Final: $r1 = 0 \wedge r2 = 1$	Final: $r1 = 0 \wedge r2 = 0$

initial state: $x = 0, y = 0$

Thread 0	Thread 1
`a: x ← 1;`	`c: r1 ← y;`
`b: y ← 1;`	`d: r2 ← x;`

assert: $r1 = 1 \wedge r2 = 0$

Interleaving 1	**Interleaving 2**	**Interleaving 3**
`a: x ← 1;`	`a: x ← 1;`	`a: x ← 1;`
`b: y ← 1;`	`c: r1 ← y;`	`c: r1 ← y;`
`c: r1 ← y;`	`b: y ← 1;`	`d: r2 ← x;`
`d: r2 ← x;`	`d: r2 ← x;`	`b: y ← 1;`
Final: $r1 = 1 \wedge r2 = 1$	Final: $r1 = 0 \wedge r2 = 1$	Final: $r1 = 0 \wedge r2 = 0$
Interleaving 4	**Interleaving 5**	**Interleaving 6**
`c: r1 ← y;`	`c: r1 ← y;`	`c: r1 ← y;`
`a: x ← 1;`	`a: x ← 1;`	`d: r2 ← x;`
`b: y ← 1;`	`d: r2 ← x;`	`a: x ← 1;`
`d: r2 ← x;`	`b: y ← 1;`	`b: y ← 1;`
Final: $r1 = 0 \wedge r2 = 1$	Final: $r1 = 0 \wedge r2 = 1$	Final: $r1 = 0 \wedge r2 = 0$

initial state: $x = 0, y = 0$	
Thread 0	Thread 1
`a: x ← 1;`	`c: r1 ← y;`
`b: y ← 1;`	`d: r2 ← x;`
assert: $r1 = 1 \wedge r2 = 0$	

assertion cannot be satisfied by interleavings

Interleaving 1	**Interleaving 2**	**Interleaving 3**
`a: x ← 1;`	`a: x ← 1;`	`a: x ← 1;`
`b: y ← 1;`	`c: r1 ← y;`	`c: r1 ← y;`
`c: r1 ← y;`	`b: y ← 1;`	`d: r2 ← x;`
`d: r2 ← x;`	`d: r2 ← x;`	`b: y ← 1;`
Final: $r1 = 1 \wedge r2 = 1$	Final: $r1 = 0 \wedge r2 = 1$	Final: $r1 = 0 \wedge r2 = 0$
Interleaving 4	**Interleaving 5**	**Interleaving 6**
`c: r1 ← y;`	`c: r1 ← y;`	`c: r1 ← y;`
`a: x ← 1;`	`a: x ← 1;`	`d: r2 ← x;`
`b: y ← 1;`	`d: r2 ← x;`	`a: x ← 1;`
`d: r2 ← x;`	`b: y ← 1;`	`b: y ← 1;`
Final: $r1 = 0 \wedge r2 = 1$	Final: $r1 = 0 \wedge r2 = 1$	Final: $r1 = 0 \wedge r2 = 0$

Weak memory models

- can we assume assertion will never pass?

initial state: $x = 0, y = 0$	
Thread 0	Thread 1
`a: x ← 1;`	`c: r1 ← y;`
`b: y ← 1;`	`d: r2 ← x;`
assert: $r1 = 1 \wedge r2 = 0$	

Weak memory models

- can we assume assertion will never pass? No!

initial state: $x = 0, y = 0$	
Thread 0	Thread 1
`a: x ← 1;`	`c: r1 ← y;`
`b: y ← 1;`	`d: r2 ← x;`
assert: $r1 = 1 \wedge r2 = 0$	

Weak memory models

- what happened?

- architectures implement *weak memory models* where the hardware is allowed to re-order certain memory instructions.

- weak memory models can allow *weak behaviors* (executions that do not correspond to an interleaving)

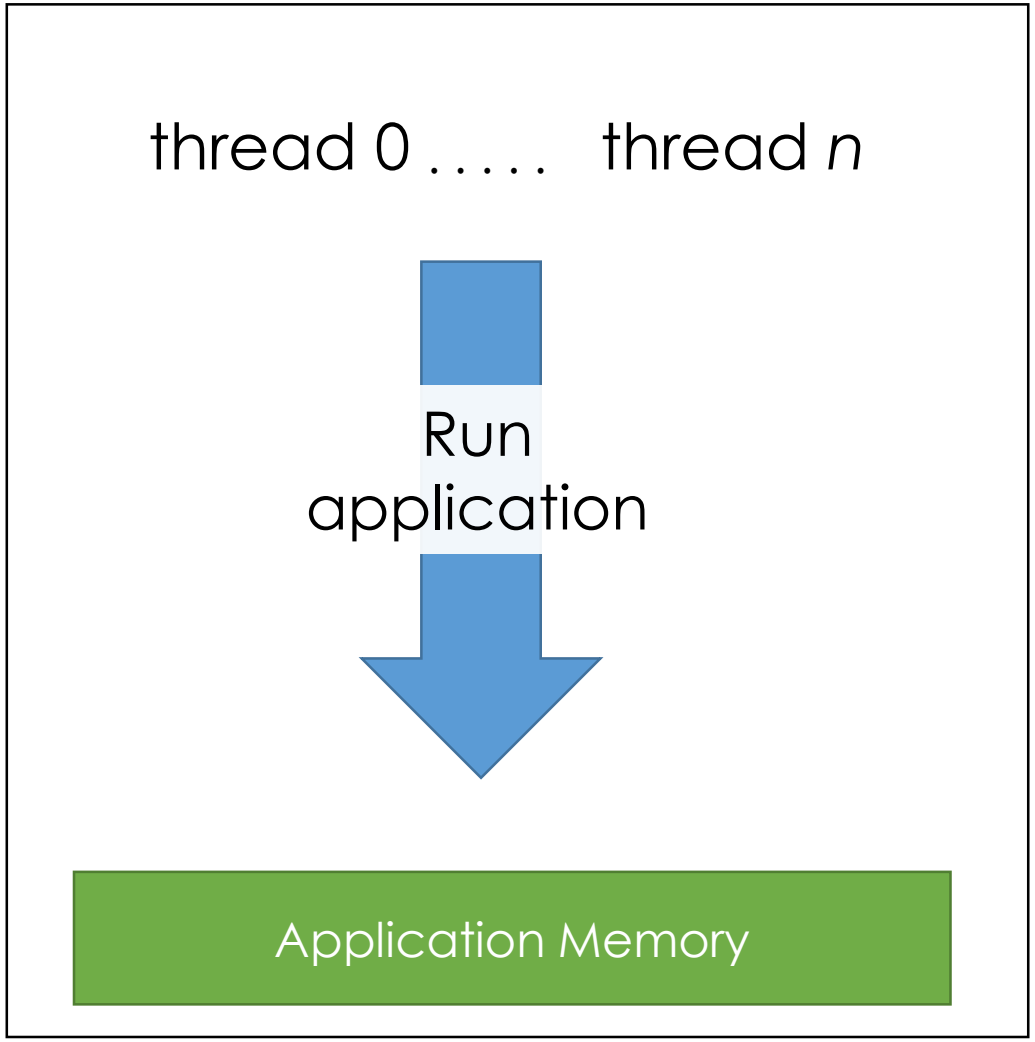
Roadmap

- Background
- Testing environment
- Environment evaluation
- Fence placement

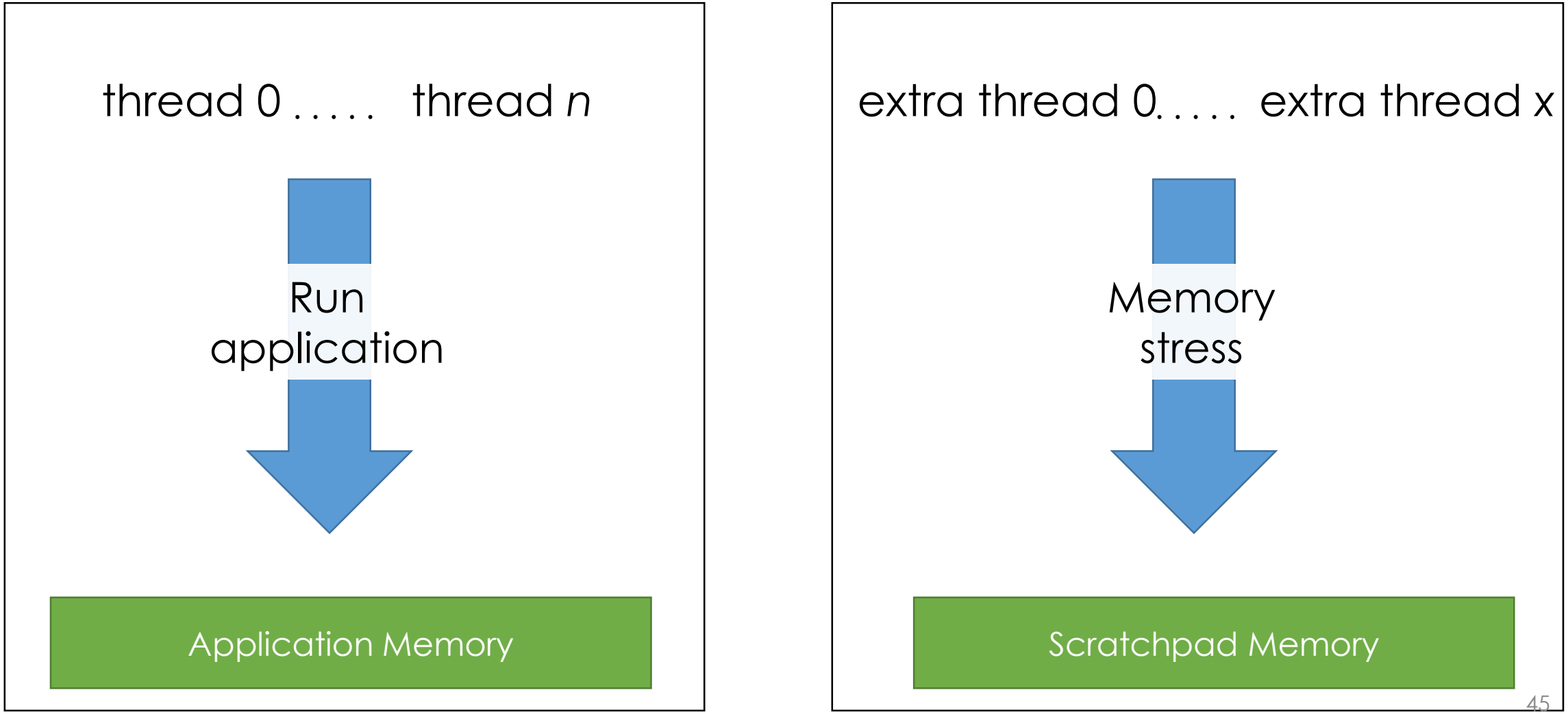
Testing environment

- Goal: Design a testing environment to reveal weak memory behaviors in GPU applications

Memory stress



Memory stress



Litmus tests

Message Passing (MP)	
initial state: x = 0, y = 0	
Thread 1	Thread 2
`a: x = 1;`	`c: r1 = y;`
`b: y = 1;`	`d: r2 = x;`
weak behavior: r1 = 1 ∧ r2 = 0	

Load Buffering (LB)	
initial state: x = 0, y = 0	
Thread 1	Thread 2
`a: r1 = x;`	`c: r2 = y;`
`b: y = 1;`	`d: x = 1;`
weak behavior: r1 = 1 ∧ r2 = 1	

Store Buffering (SB)	
initial state: x = 0, y = 0	
Thread 1	Thread 2
`a: x = 1;`	`c: y = 1;`
`b: r1 = y;`	`d: r2 = x;`
weak behavior: r1 = 0 ∧ r2 = 0	

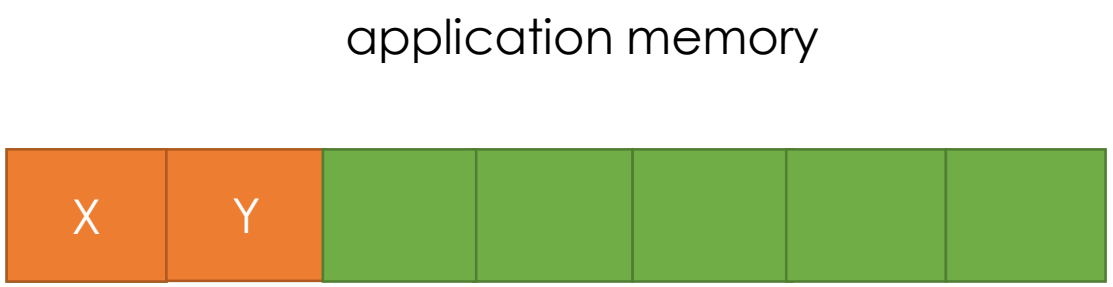
Memory stress

Where to stress:

Memory stress

Where to stress:

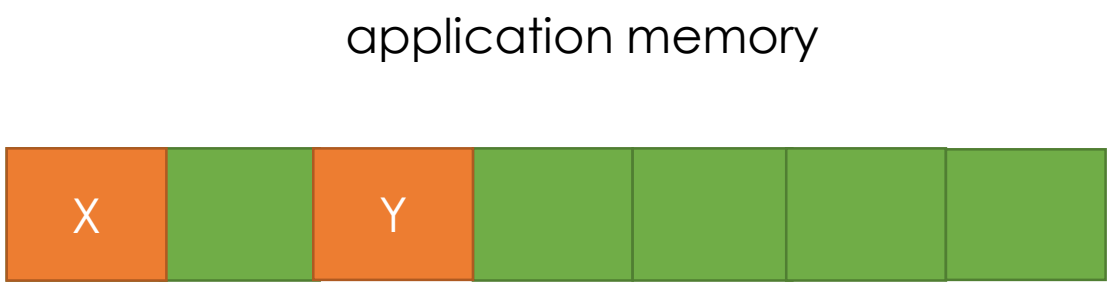
- For each distance *D*:



Memory stress

Where to stress:

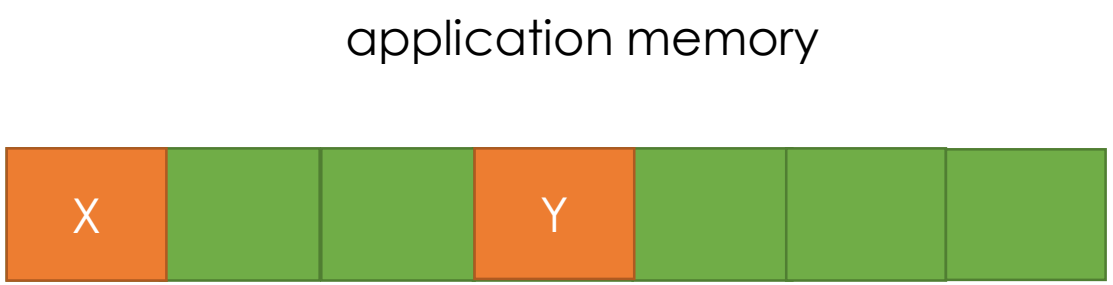
- For each distance *D*:



Memory stress

Where to stress:

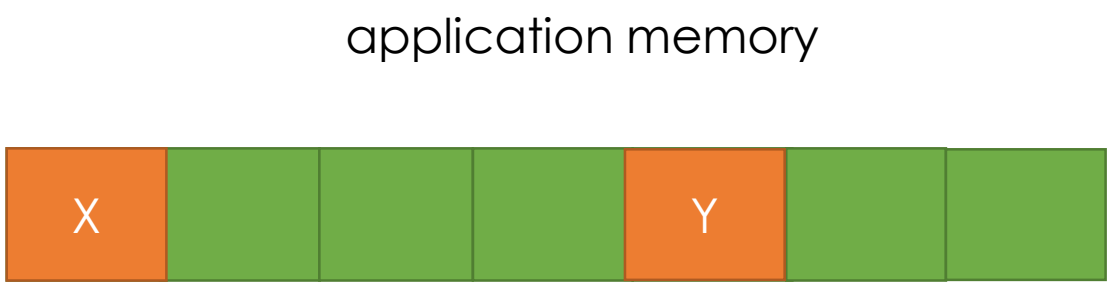
- For each distance *D*:



Memory stress

Where to stress:

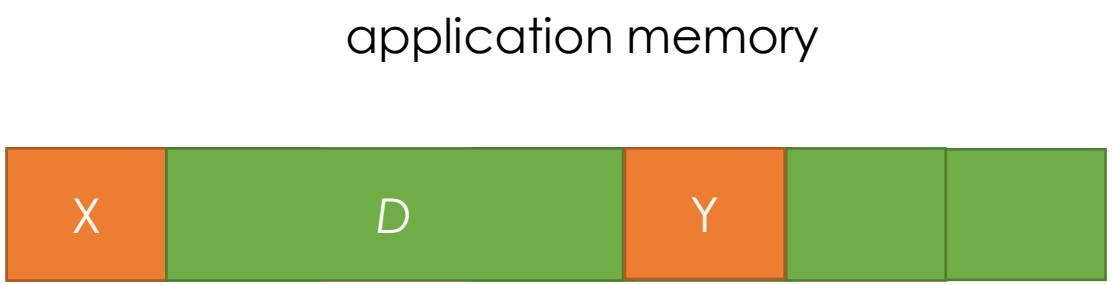
- For each distance *D*:



Memory stress

Where to stress:

- For each distance *D*:



Memory stress

Where to stress:

- For each distance *D*:

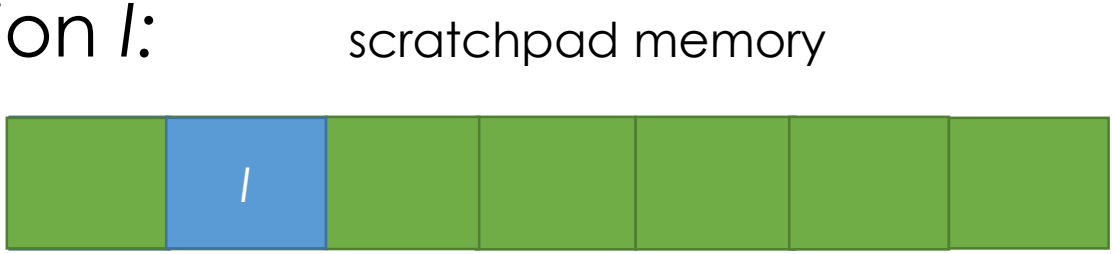
- For each scratchpad location *I*:

Memory stress

Where to stress:

- For each distance *D*:

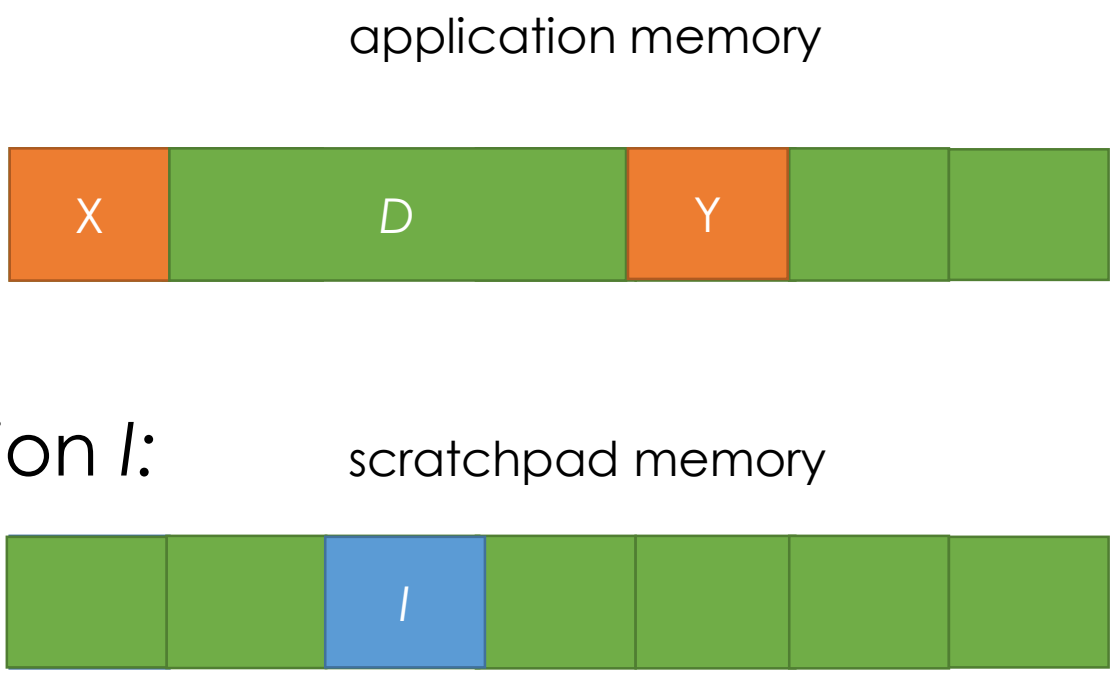
- For each scratchpad location *I*:



Memory stress

Where to stress:

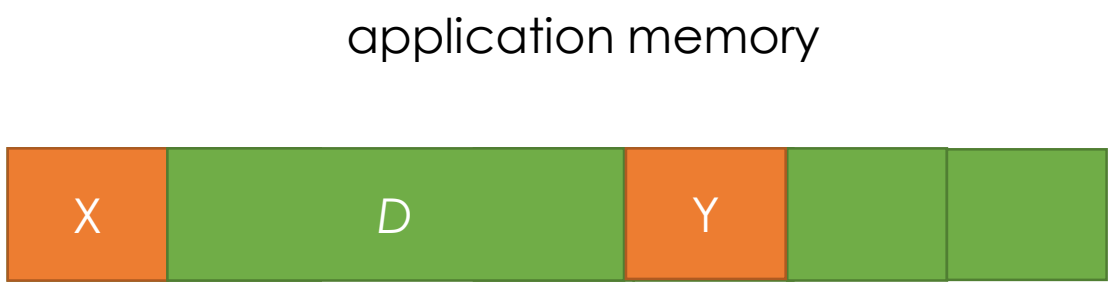
- For each distance *D*:
- For each scratchpad location *I*:



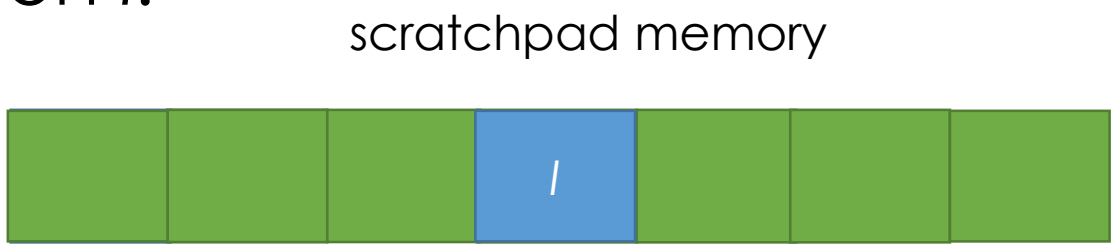
Memory stress

Where to stress:

- For each distance *D*:

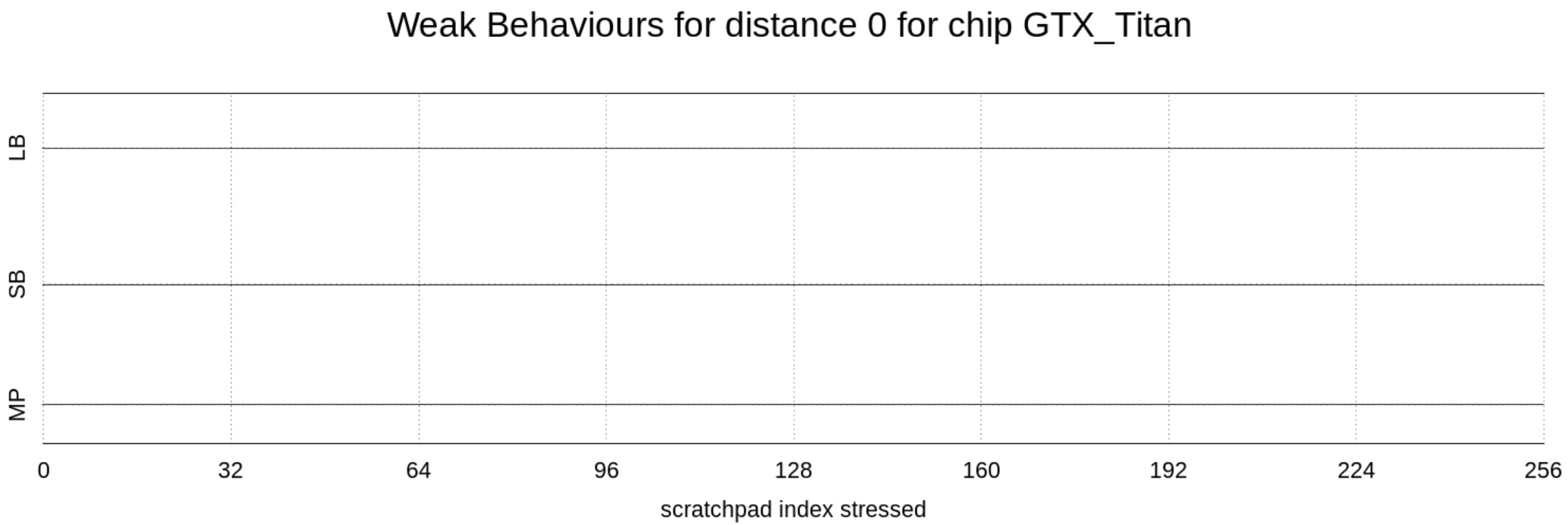


- For each scratchpad location *l*:

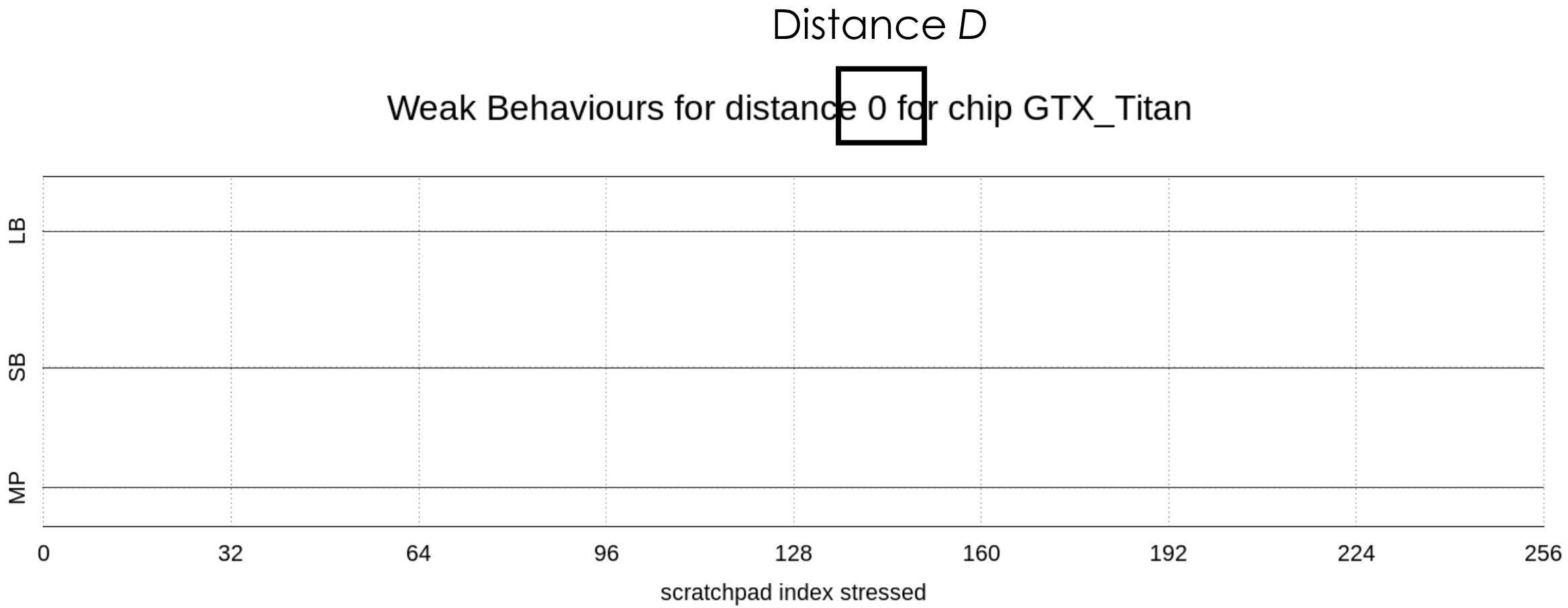


- **Run MP, SB, LB at distance *D* litmus tests stressing only location *l* for 1000 iterations**

Memory stress



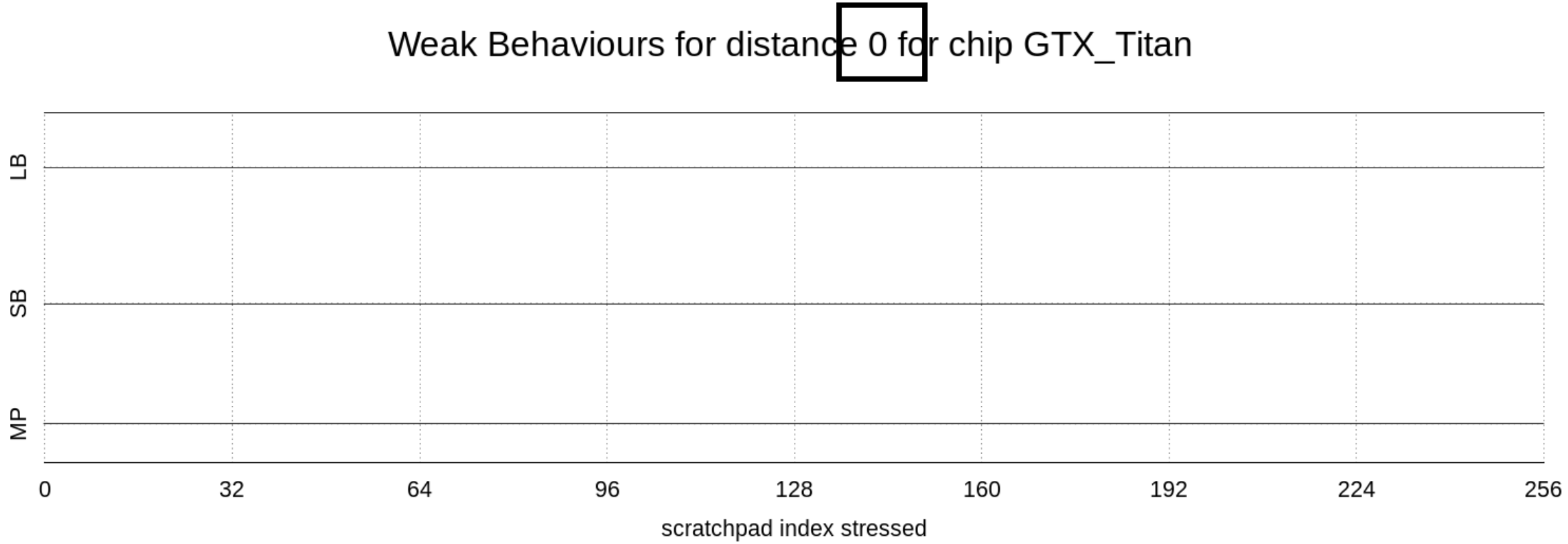
Memory stress



Memory stress

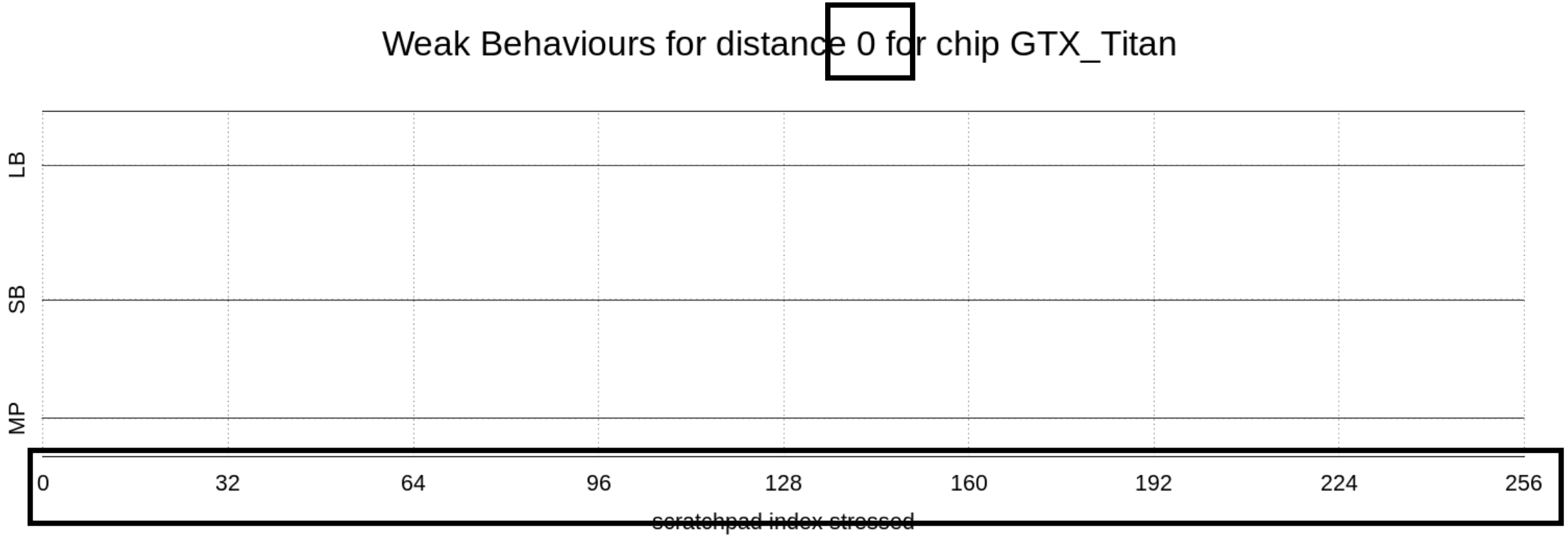
application memory

Distance *D*



Memory stress

Distance *D*

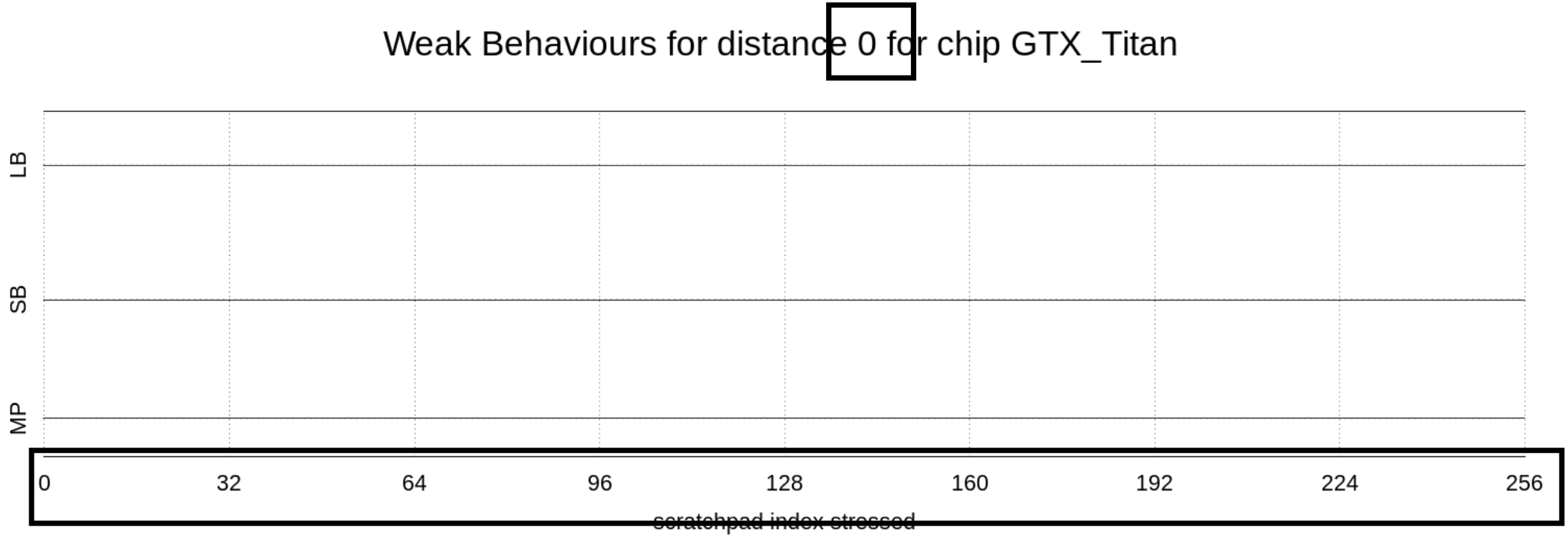


Index *I* stressed

Memory stress

scratchpad memory

Distance *D*

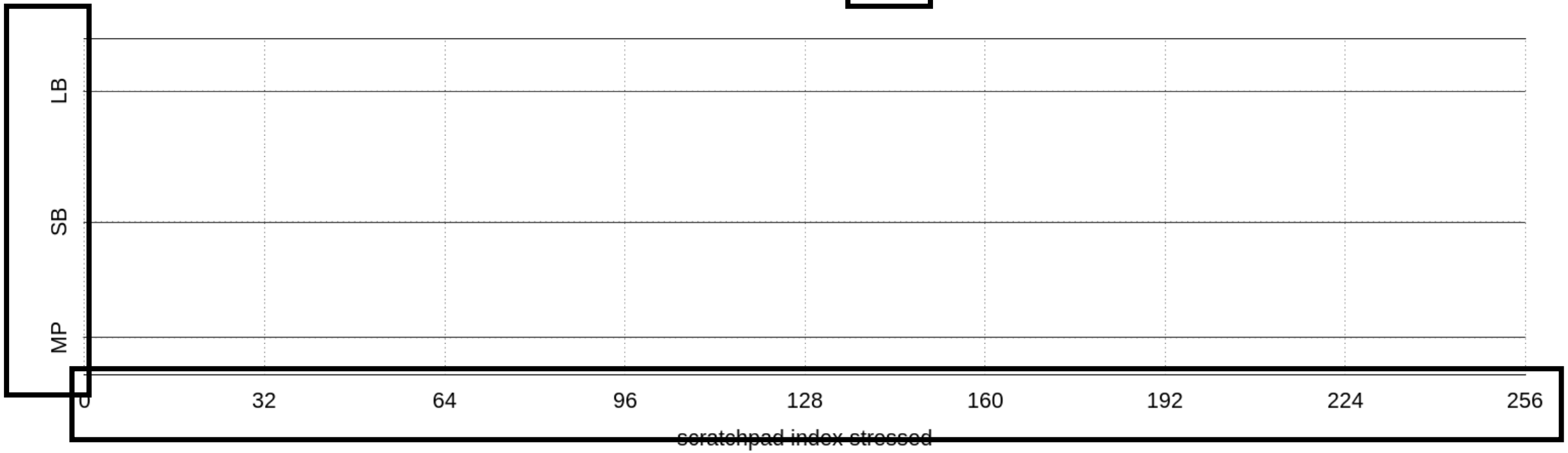


Index *I* stressed

Memory stress

Distance *D*

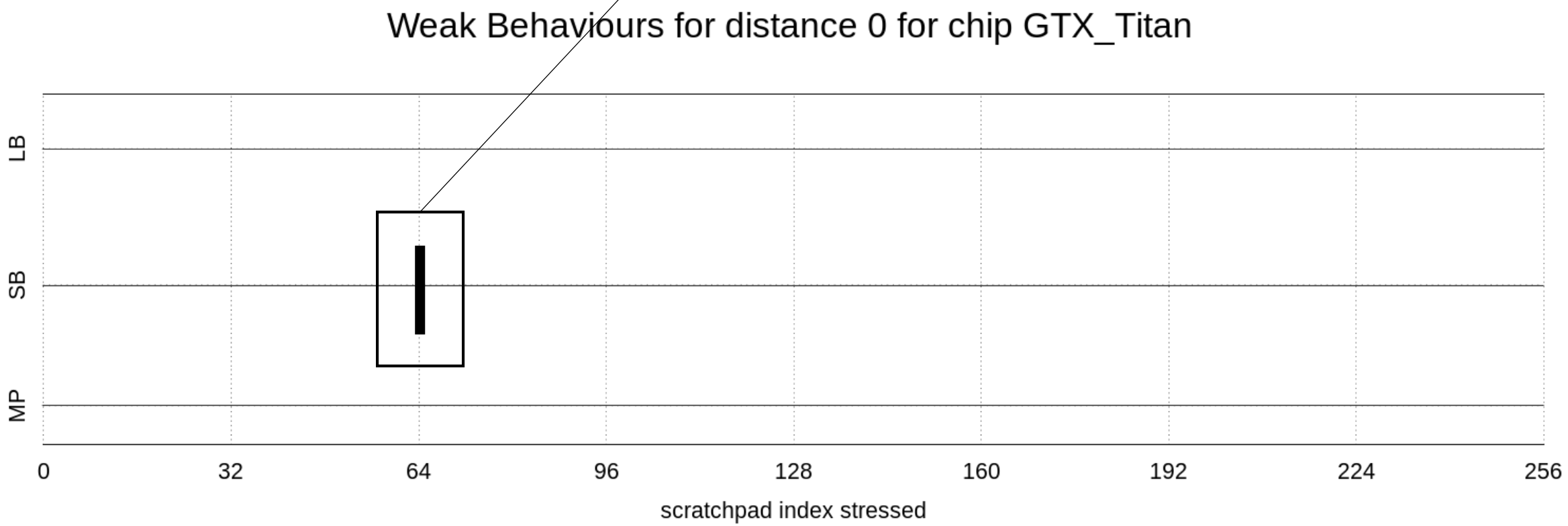
Litmus test



Index *I* stressed

Memory stress

Vertical bar represents the magnitude of weak behaviors observed

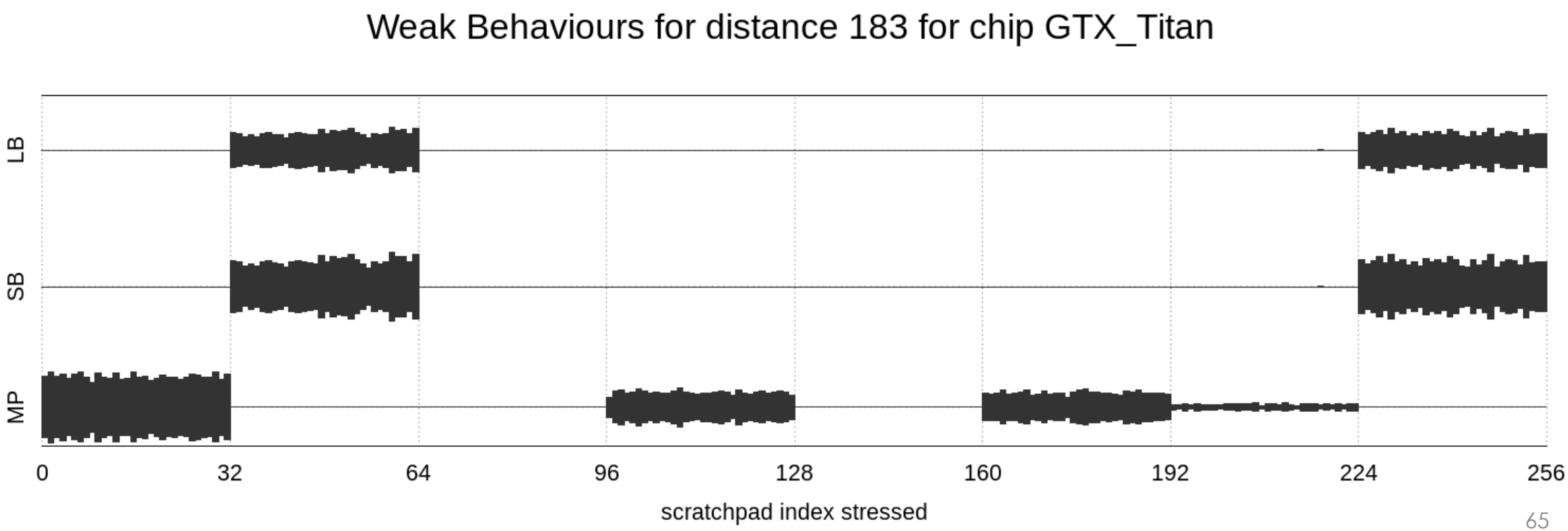


Memory stress

- Visualization samples

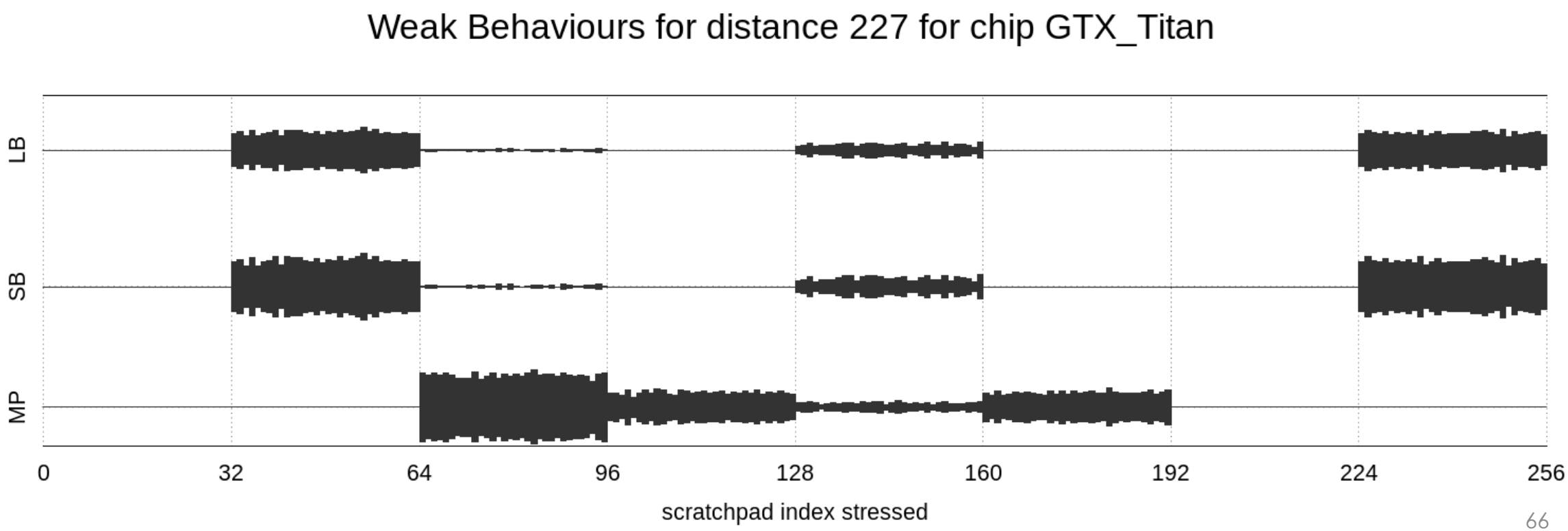
Memory stress

- Visualization samples



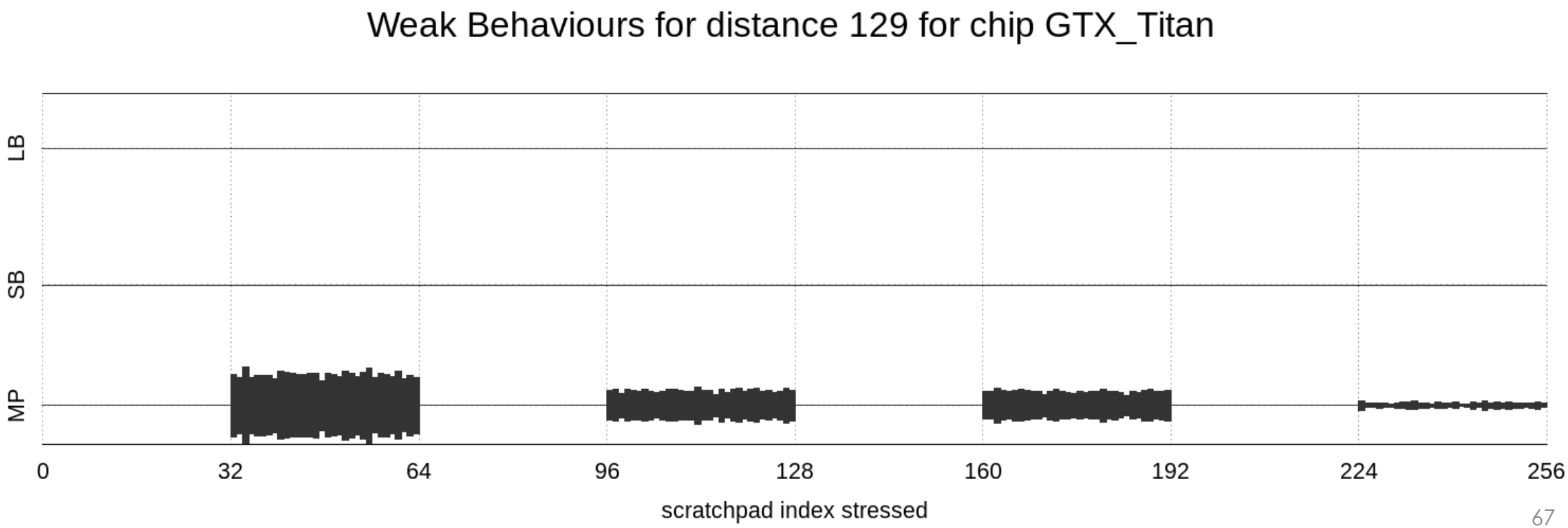
Memory stress

- Visualization samples



Memory stress

- Visualization samples



Memory stress

- What does this tell us?

Memory stress

- What does this tell us?
- *To reveal weak behaviors we only need to stress 1 in every 32 locations**
- We call a contiguous region of 32 elements a ***patch***

*64 for some chips

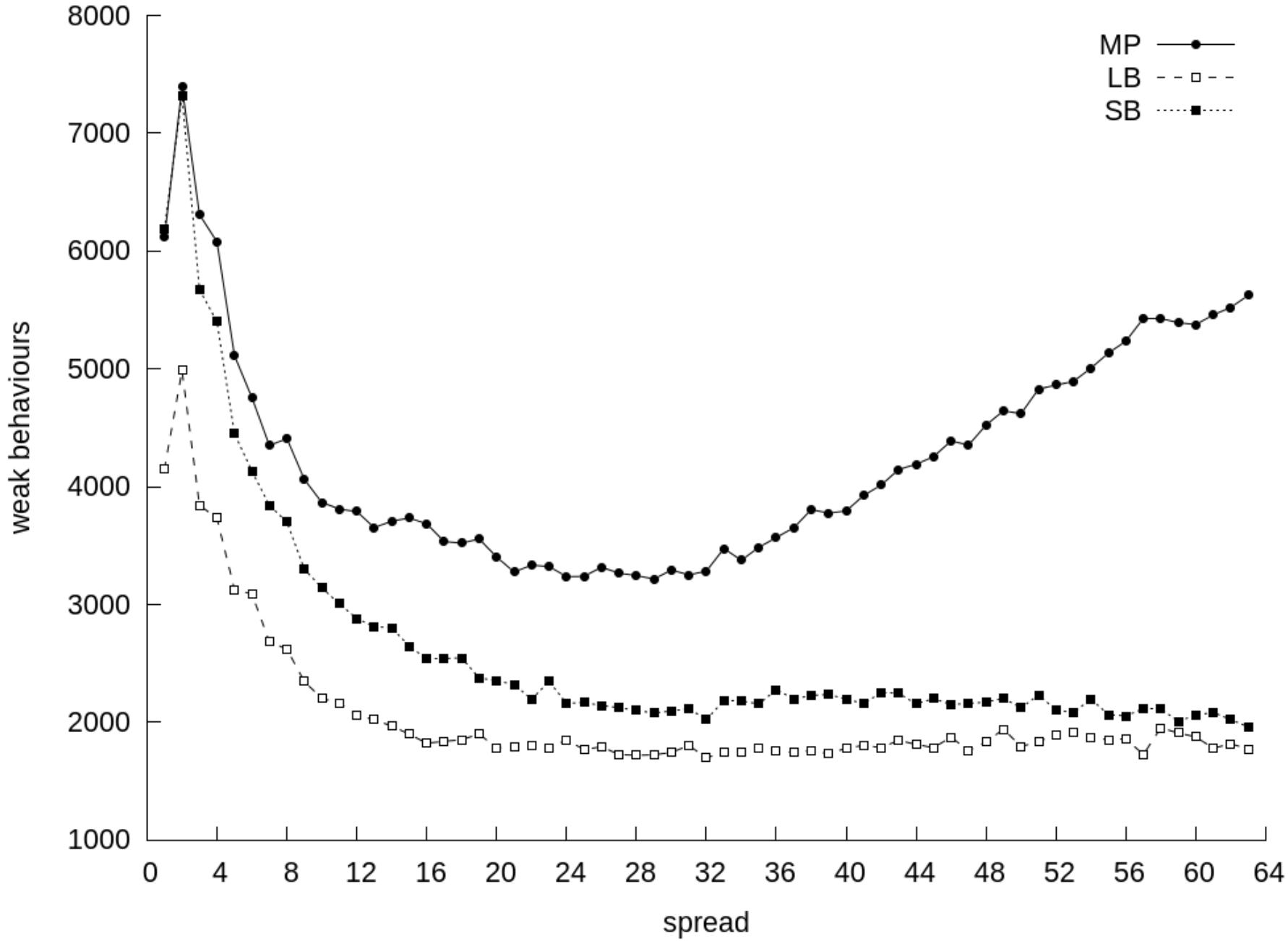
Memory stress

- How many patches can we effectively stress?

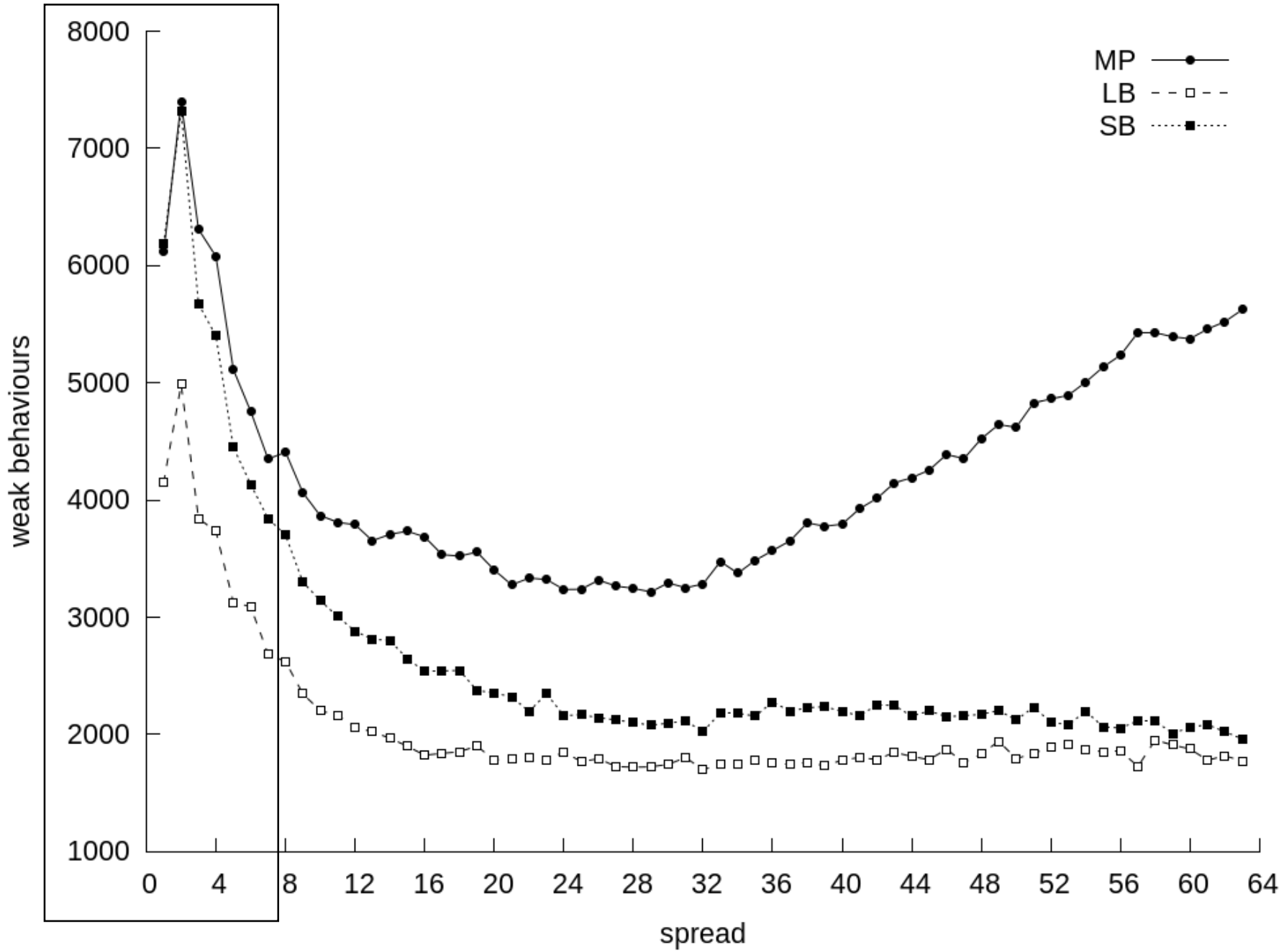
- If *D* is unknown (as in applications), we would like to stress as many disjoint patches as possible

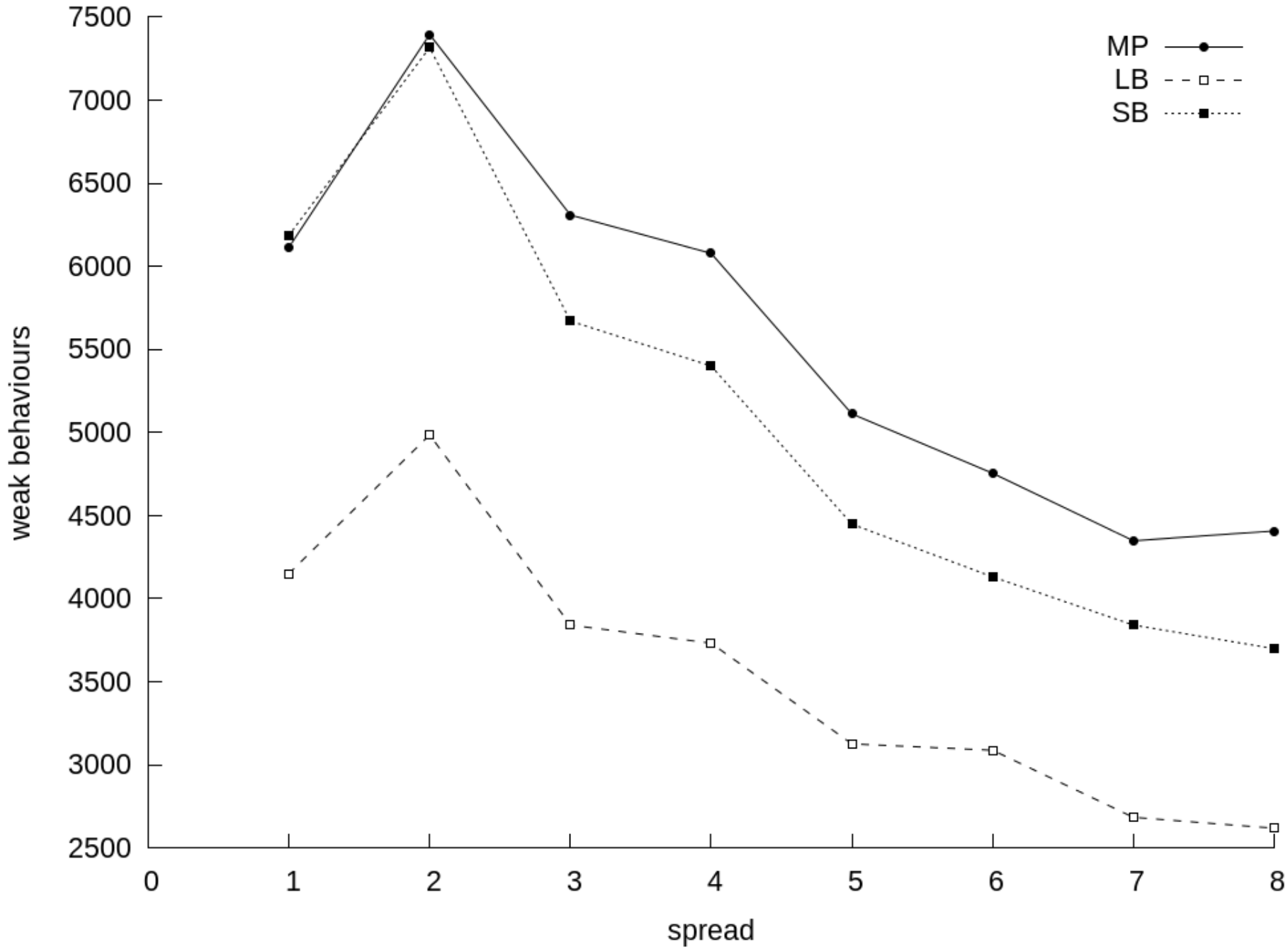
Memory stress

- Scratchpad has size of 64 patches
- We try stressing a randomly selected *n* patches for values 1 – 64 for *n*

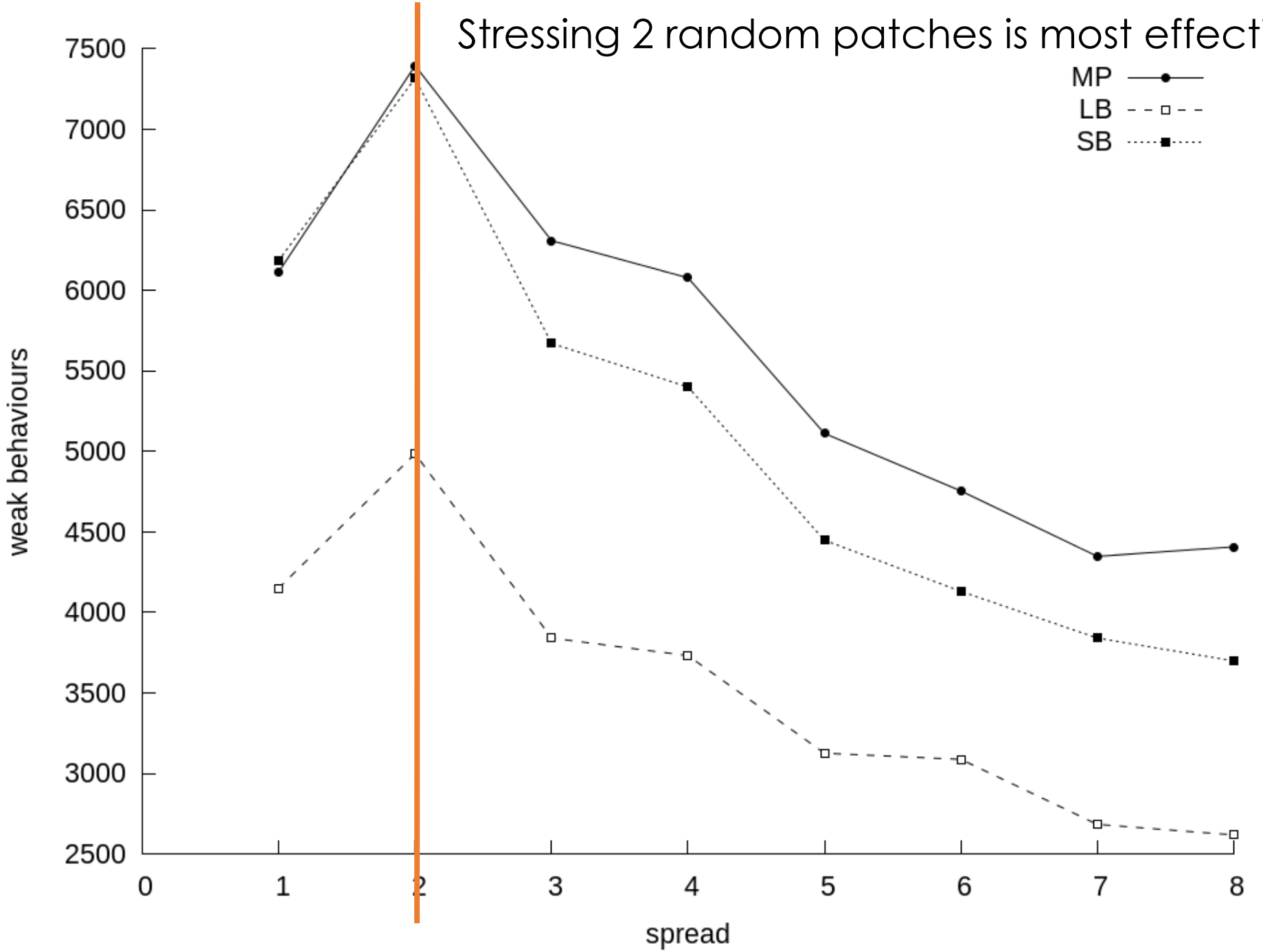

8000
7000
6000
5000
4000
3000
2000
1000
weak behaviours
0
4
8
12
16
20
24
28
32
36
40
44
48
52
56
60
64
spread
MP
LB
SB

Zoom in
on first 8




7500
7000
6500
6000
5500
5000
4500
4000
3500
3000
2500
weak behaviours
0
1
2
3
4
5
6
7
8
spread
MP
LB
SB

Stressing 2 random patches is most effective



Memory stress

- Now we have a memory stressing strategy
  - Stress two random patches in the scratchpad

Roadmap

- Background
- Testing environment
- Environment evaluation
- Fence placement

Applications

- Observed weak memory issues in 8 applications
- 7 Nvidia chips (across 3 architectures)
- Run applications for 1 hour

Applications

CHIP	No-stress	Stress
980		
K5200		
Titan		
K20c		
770		
C2075		
C2050		

Applications

CHIP	No-stress	Stress
980	0	
K5200	1	
Titan	0	
K20c	0	
770	1	
C2075	0	
C2050	0	

Applications

CHIP	No-stress	Stress
980	0	7
K5200	1	8
Titan	0	8
K20c	0	8
770	1	8
C2075	0	8
C2050	0	8

Some results:

- We provided empirical confirmation of **3** bugs reported in prior work
- We discovered unreported weak memory bugs in **2** applications

Roadmap

- Background
- Testing environment
- Environment evaluation
- Fence placement

Fence insertion

- Start with conservative fence placement
- Attempt to remove fences
- Use our testing framework as an unsound oracle
- If errors are observed, put fence back

Fence insertion

App	Conservative Fences	Reduced Fences
cbe-ht	10	
cbe-dot	4	
ct-octree	33	
tpo-tm	28	
sdk-red	6	
cub-scan	51	
ls-bh	90	

Fence insertion

App	Conservative Fences	Reduced Fences
cbe-ht	10	1
cbe-dot	4	1
ct-octree	33	1
tpo-tm	28	1
sdk-red	6	1
cub-scan	51	2
ls-bh	90	4

Fence insertion

- Median overhead of reduced fences:
  - **4% energy**
  - **Less than 3% runtime**

- Median overhead of consv. Fences
  - **63% energy**
  - **59% runtime**

Conclusion

- Stress/fuzz testing can be used to test applications for weak memory bugs
- Stress is best tuned on micro benchmarks
- Fences in applications can be costly, but can be reduced with high confidence using stress/fuzz testing

Tyler Sorensen
https://www.doc.ic.ac.uk/~tsorensen/

Alastair F. Donaldson
http://multicore.doc.ic.ac.uk/