JÜLICH
FORSCHUNGSZENTRUM

JÜLICH
APPLICATION LAB
NVIDIA

Halloc: Fast malloc()/free() for GPUs

Andrew V. Adinetz
adinetz@gmail.com

Dirk Pleiter
d.pleiter@fz-juelich.de

Jülich Supercomputing Centre
Forschungszentrum Jülich

Outline

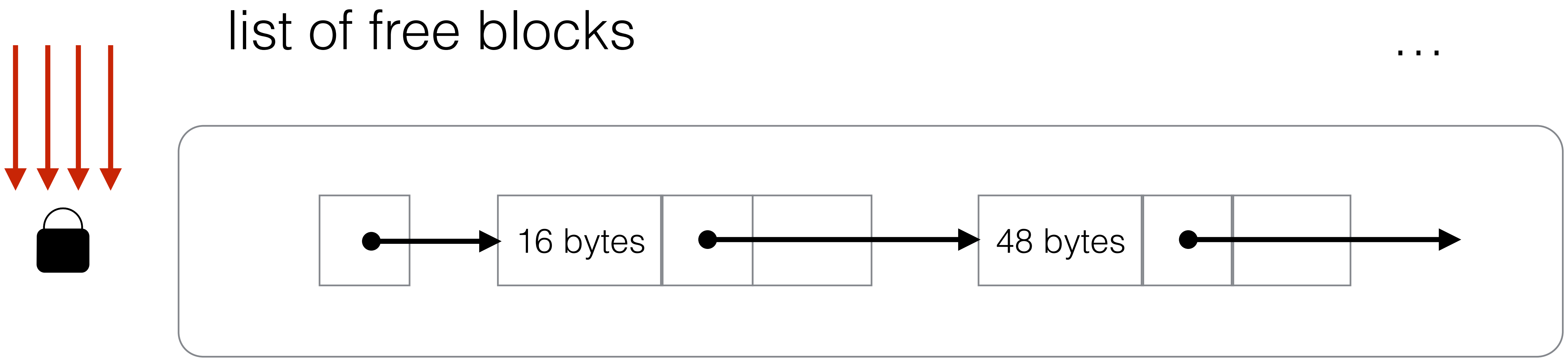
- Motivation
- Halloc's main idea
- Aspects of halloc design
- Performance benchmarks

Outline

- **Motivation**
- Halloc's main idea
- Aspects of halloc design
- Performance benchmarks

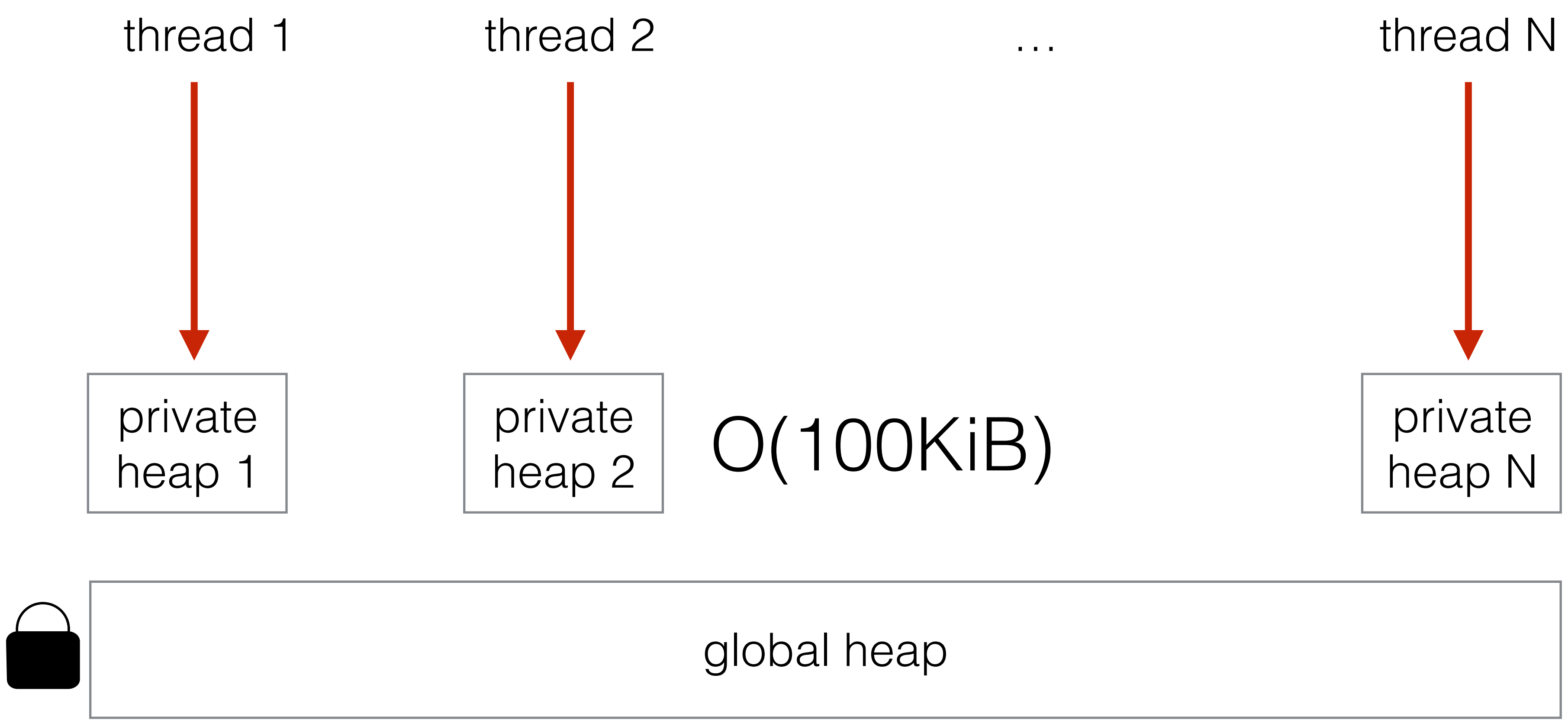
CPU Dynamic Memory Allocation

- taken for granted
  - malloc() / free()



- lock: 1 thread at a time
- O(10) CPU threads: bad
- O(10^4) GPU threads: unacceptable

Private Heap



- small malloc() in private heap

jemalloc
tcmalloc
Hoard allocator

- free() in different private heap?
- $O(10^4)$ GPU threads?

(GPU) Allocator Requirements

- Memory usage
  - fragmentation, overhead
- Performance
  - O(100 M) operations/s
- Scalability
  - O(10 K) threads
- Correctness
- Robustness
  - acceptable performance across different use cases

Outline

- Motivation
- **Halloc's main idea**
- Aspects of halloc design
- Performance benchmarks

Bitmap Heap

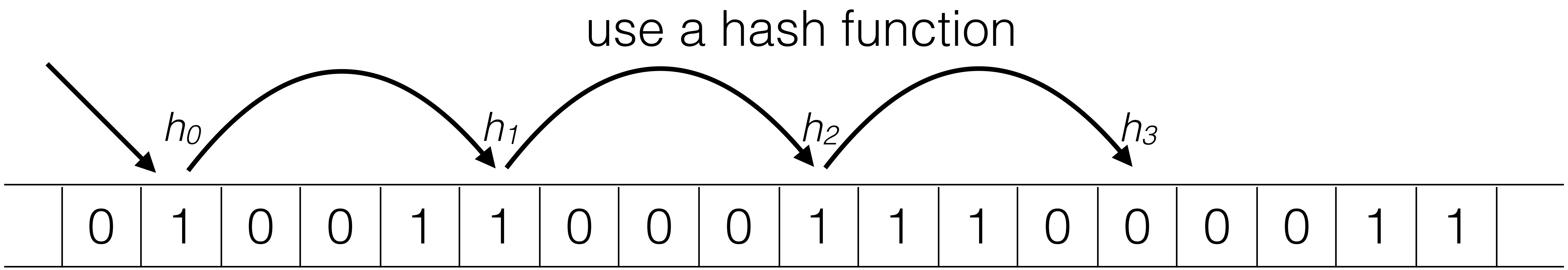
allocate:
```
atomicOr(&word, 1 << sh)
```

free:
```
atomicAnd(&word, ~(1 << sh))
```

0	1	0	0	1	1	0	0	0	1	1	1	0	0	0	0	1	1

- 1 bit/chunk or block
- simple and scalable
- how to get the block to allocate?

Getting Free Block



$$\begin{cases} h(0,c) = T \cdot c \bmod N \\ h(i,c) = (h(0,c) + s \cdot i) \bmod N \end{cases}$$

c — allocation counter

T — counter multiplier

s — hash step

N — number of blocks

- correct (visits all blocks)
- fast and scalable (if < 85% blocks allocated)
- special-purpose allocator
- c incremented atomically

Halloc Hash Function in Detail

#chunks in block (1–8)

allocation counter
(per size)

initial chunk to try

$$\begin{cases} h(c,0) = b \cdot (c/K \cdot KT + c \bmod K) \bmod N \\ h(c,i) = (h(c,0) + (i+1) \cdot ((i+1)bS \bmod s)) \bmod N \end{cases}$$

subsequent chunks to try

#chunks in slab
(multiple of b)

Halloc Hash Function in Detail

#chunks in block (1–8)

allocation counter
(per size)

K=1–8 — better coalescing
for small blocks

initial chunk to try

$$\begin{cases} h(c,0) = b \cdot (c/K \cdot KT + c \bmod K) \bmod N \\ h(c,i) = (h(c,0) + (i+1) \cdot ((i+1)bS \bmod s)) \bmod N \end{cases}$$

subsequent chunks to try

#chunks in slab
(multiple of *b*)

Halloc Hash Function in Detail

#chunks in block (1–8)

T is prime (7, 11, 13)
reduces collisions between threads

allocation counter
(per size)

K=1–8 — better coalescing
for small blocks

initial chunk to try

$$\begin{cases} h(c,0) = b \cdot (\overline{c/K \cdot KT} + c \bmod K) \bmod N \\ h(c,i) = (h(c,0) + (i+1) \cdot ((i+1)bS \bmod s)) \bmod N \end{cases}$$

subsequent chunks to try

#chunks in slab
(multiple of *b*)

Halloc Hash Function in Detail

#chunks in block (1–8)

T is prime (7, 11, 13)
reduces collisions between threads

allocation counter
(per size)

K=1–8 — better coalescing
for small blocks

initial chunk to try

$$\begin{cases} h(c,0) = b \cdot (\overline{c/K \cdot KT} + c \bmod K) \bmod N \\ h(c,i) = (h(c,0) + \underline{(i+1) \cdot ((i+1)bS \bmod s)}) \bmod N \end{cases}$$

subsequent chunks to try

in practice faster than linear hashing visits
all blocks with right choice of *b*, *S* and *s*

#chunks in slab
(multiple of *b*)

Warp-Aggregated Atomic Increment

```
lid = lane_id();
while(__any(want_inc))
  if(want_inc) {
    mask = __ballot(want_inc);
    leader_lid = __ffs(mask) - 1;
    leader_size_id = __shfl
      (leader_size_id, leader_lid);
    group_mask = __ballot
      (size_id == leader_size_id);
    want_inc = false;
  }

group_change = __popc(group_mask);
if(lid == leader_lid)
  old_counter = atomicAdd
    (&counters[size_id], group_change);
old_counter = __shfl
  (old_counter, leader_lid);

change = __popc(group_mask & ((1 << lid) - 1));
return old_counter + change;
```

multiple counters (different sizes)

Warp-Aggregated Atomic Increment

```
lid = lane_id();
while(__any(want_inc))
  if(want_inc) {
    mask = __ballot(want_inc);
    leader_lid = __ffs(mask) - 1;
    leader_size_id = __shfl
      (leader_size_id, leader_lid);
    group_mask = __ballot
      (size_id == leader_size_id);
    want_inc = false;
  }

group_change = __popc(group_mask);
if(lid == leader_lid)
  old_counter = atomicAdd
    (&counters[size_id], group_change);
old_counter = __shfl
  (old_counter, leader_lid);

change = __popc(group_mask & ((1 << lid) - 1));
return old_counter + change;
```

multiple counters (different sizes)

select the leader …

Mitglied der Helmholtz-Gemeinschaft

Warp-Aggregated Atomic Increment

```
lid = lane_id();
while(__any(want_inc))
  if(want_inc) {
    mask = __ballot(want_inc);
    leader_lid = __ffs(mask) - 1;
    leader_size_id = __shfl
      (leader_size_id, leader_lid);
    group_mask = __ballot
      (size_id == leader_size_id);
    want_inc = false;
  }

group_change = __popc(group_mask);
if(lid == leader_lid)
  old_counter = atomicAdd
    (&counters[size_id], group_change);
old_counter = __shfl
  (old_counter, leader_lid);

change = __popc(group_mask & ((1 << lid) - 1));
return old_counter + change;
```

multiple counters (different sizes)

select the leader …

… only for threads with the same size_id

Mitglied der Helmholtz-Gemeinschaft

Warp-Aggregated Atomic Increment

```
lid = lane_id();
while(__any(want_inc))
  if(want_inc) {
    mask = __ballot(want_inc);
    leader_lid = __ffs(mask) - 1;
    leader_size_id = __shfl
      (leader_size_id, leader_lid);
    group_mask = __ballot
      (size_id == leader_size_id);
    want_inc = false;
  }

group_change = __popc(group_mask);
if(lid == leader_lid)
  old_counter = atomicAdd
    (&counters[size_id], group_change);
old_counter = __shfl
  (old_counter, leader_lid);

change = __popc(group_mask & ((1 << lid) - 1));
return old_counter + change;
```

multiple counters (different sizes)

select the leader …

… only for threads with the same size_id

leaders increment the counters …

Warp-Aggregated Atomic Increment

```
lid = lane_id();
while(__any(want_inc))
  if(want_inc) {
    mask = __ballot(want_inc);
    leader_lid = __ffs(mask) - 1;
    leader_size_id = __shfl
      (leader_size_id, leader_lid);
    group_mask = __ballot
      (size_id == leader_size_id);
    want_inc = false;
  }

group_change = __popc(group_mask);
if(lid == leader_lid)
  old_counter = atomicAdd
    (&counters[size_id], group_change);
old_counter = __shfl
  (old_counter, leader_lid);

change = __popc(group_mask & ((1 << lid) - 1));
return old_counter + change;
```

multiple counters (different sizes)

select the leader …

… only for threads with the same size_id

leaders increment the counters …

… and broadcast values to threads of their groups

Warp-Aggregated Atomic Increment

multiple counters (different sizes)

```
lid = lane_id();
while(__any(want_inc))
  if(want_inc) {
    mask = __ballot(want_inc);
    leader_lid = __ffs(mask) - 1;
    leader_size_id = __shfl
      (leader_size_id, leader_lid);
    group_mask = __ballot
      (size_id == leader_size_id);
    want_inc = false;
  }

group_change = __popc(group_mask);
if(lid == leader_lid)
  old_counter = atomicAdd
    (&counters[size_id], group_change);
old_counter = __shfl
  (old_counter, leader_lid);

change = __popc(group_mask & ((1 << lid) - 1));
return old_counter + change;
```

select the leader …

… only for threads with the same size_id

leaders increment the counters …

… and broadcast values to threads of their groups

each thread computes its value

Warp-Aggregated Atomic Increment

```
lid = lane_id();
while(__any(want_inc))
  if(want_inc) {
    mask = __ballot(want_inc);
    leader_lid = __ffs(mask) - 1;
    leader_size_id = __shfl
      (leader_size_id, leader_lid);
    group_mask = __ballot
      (size_id == leader_size_id);
    want_inc = false;
  }

group_change = __popc(group_mask);
if(lid == leader_lid)
  old_counter = atomicAdd
    (&counters[size_id], group_change);
old_counter = __shfl
  (old_counter, leader_lid);

change = __popc(group_mask & ((1 << lid) - 1));
return old_counter + change;
```

multiple counters (different sizes)

select the leader …

… only for threads with the same size_id

Up to 32x less atomics

leaders increment the counters …

… and broadcast values to threads of their groups

each thread computes its value

Outline

- Motivation
- Halloc’s main idea
- **Aspects of halloc design**
- Performance benchmarks

Halloc Design

size	alloc_ctr	head
16	123	
24	12	
32	1025000	
48	0	NONE
64	0	NONE
96	0	NONE
…	…	…

>= 3 KiB:
CUDA's malloc()

Halloc Design

size	alloc_ctr	head
16	123	
24	12	
32	1025000	
48	0	NONE
64	0	NONE
96	0	NONE
…	…	…

…

…

bitmap	
alloc_sizes	
memory	
slab_ctr	1234
block_sz	32
chunk_sz	16
…	…

>= 3 KiB:
CUDA's malloc()

slab: 2–8 MiB
"from OS" (= preallocated)
specialized

Halloc Design

size	alloc_ctr	head
16	123	
24	12	
32	1025000	
48	0	NONE
64	0	NONE
96	0	NONE
…	…	…

…

…

bitmap	
alloc_sizes	
memory	
slab_ctr	1234
block_sz	32
chunk_sz	16
…	…

2–8 MiB

>= 3 KiB:
CUDA's malloc()

slab: 2–8 MiB
"from OS" (= preallocated)
specialized

Halloc Design

size	alloc_ctr	head
16	123	…
24	12	…
32	1025000	
48	0	NONE
64	0	NONE
96	0	NONE
…	…	…

bitmap	
alloc_sizes	
memory	
slab_ctr	1234
block_sz	32
chunk_sz	16
…	…

1	1	0	0	1	0	1	1	…

2–8 MiB

>= 3 KiB:
CUDA's malloc()

slab: 2–8 MiB
"from OS" (= preallocated)
specialized

Halloc Design

size	alloc_ctr	head
16	123	…
24	12	…
32	1025000	
48	0	NONE
64	0	NONE
96	0	NONE
…	…	…

bitmap	
alloc_sizes	
memory	
slab_ctr	1234
block_sz	32
chunk_sz	16
…	…

1	1	0	0	1	0	1	1	…

2	0	0	0	**1**	0	**2**	0	…

2–8 MiB

>= 3 KiB:
CUDA's malloc()

slab: 2–8 MiB
"from OS" (= preallocated)
specialized

Chunks

- Without chunks
  - 1 bit = different block sizes
  - incompatible metadata

1	1	0	0	1	0	1	1	0	0

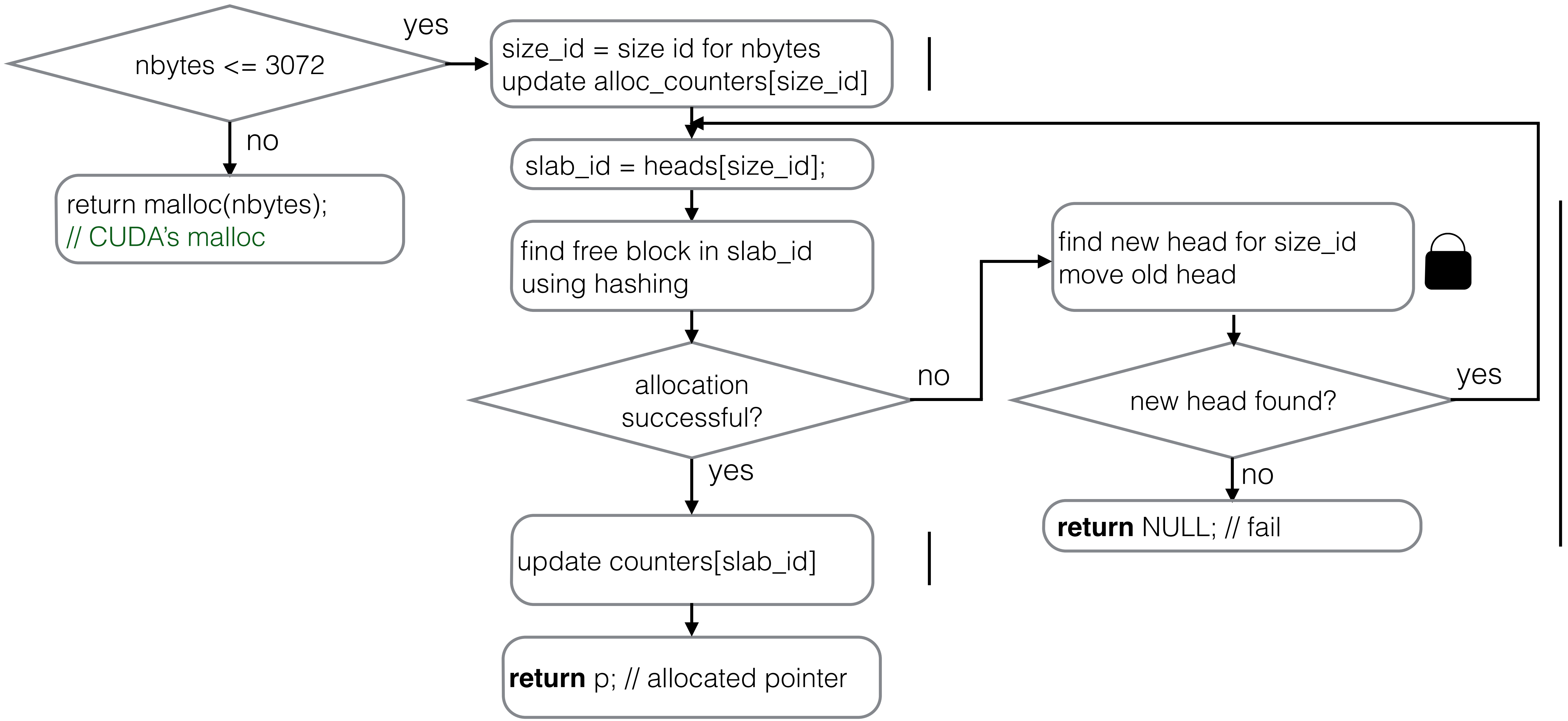
0	1	0	0	1	0	1

- With chunks
  - 1 bit = 1 chunk
  - **block** = returned by malloc()
  - block = several chunks (1x, 2x, 4x, 8x)
  - non-free slabs can switch within chunk

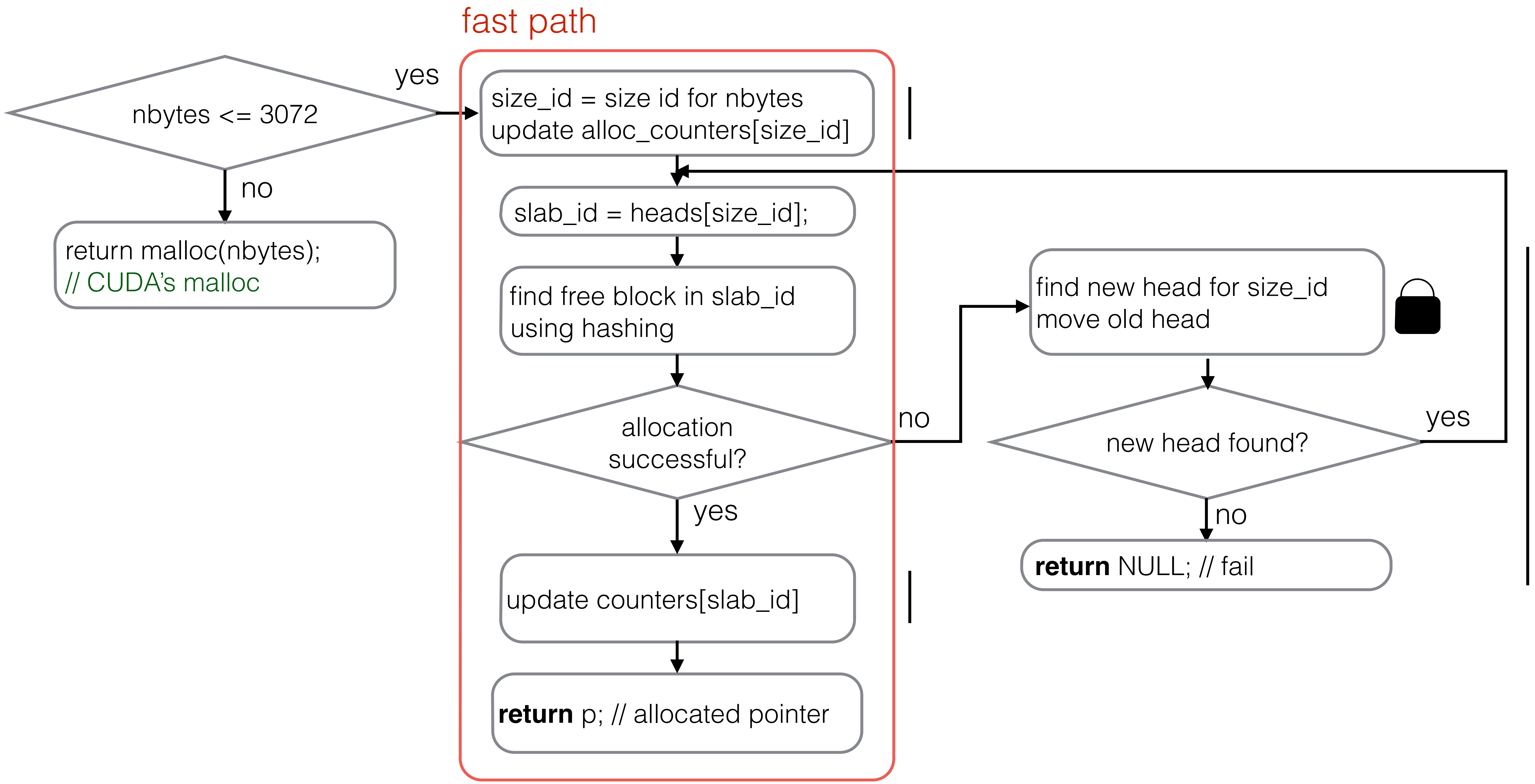
1	1	0	0	1	0	1	1	0	0

1	1	0	0	1	0	1	1	0	0

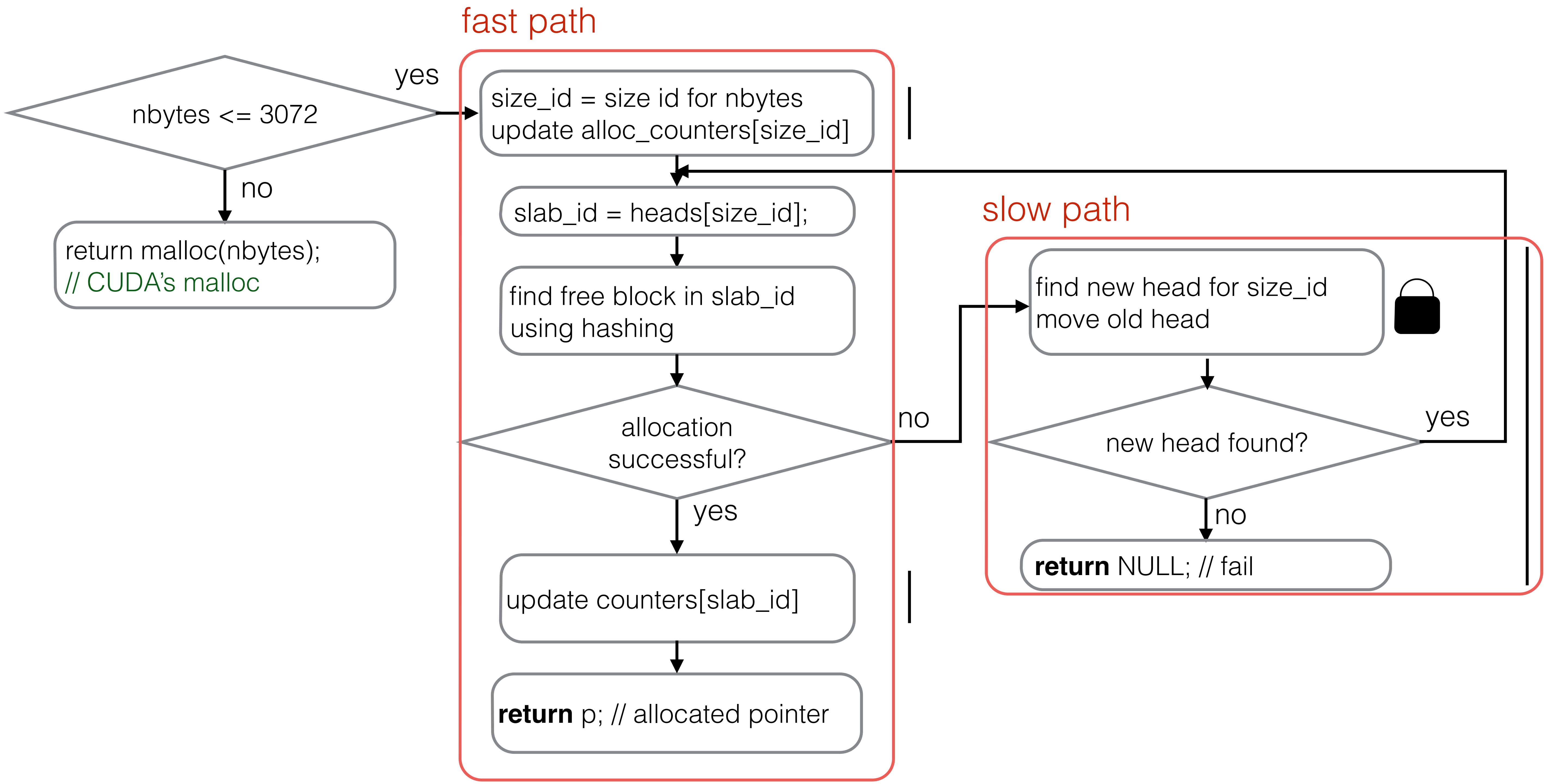
void *hamalloc(size_t nbytes)



void *hamalloc(size_t nbytes)



void *hamalloc(size_t nbytes)

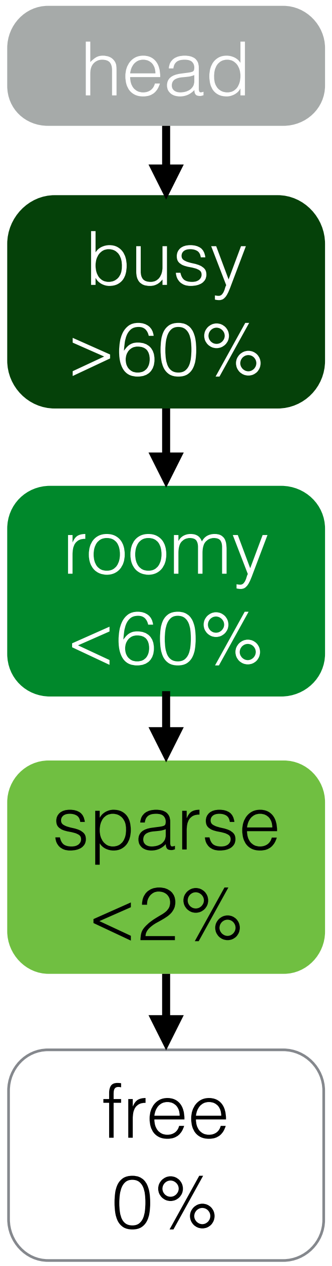


Head Replacement

- Can affect performance
  - 1.7 Gmalloc/s x 32 B = 50.66 GiB/s
  - 83.5% x 4 MiB / 50.66 GiB/s = **64 μs** – slab consumption time
  - head replacement: O(10 μs)
  - other threads have to wait
- Single-threaded code, lots of memory accesses
  - GPUs not optimized for that
- Effective slab usage
  - avoid fragmentation
  - avoid filled-up slabs

Slab Classes

based on slab fill ratio (thresholds adjustable)



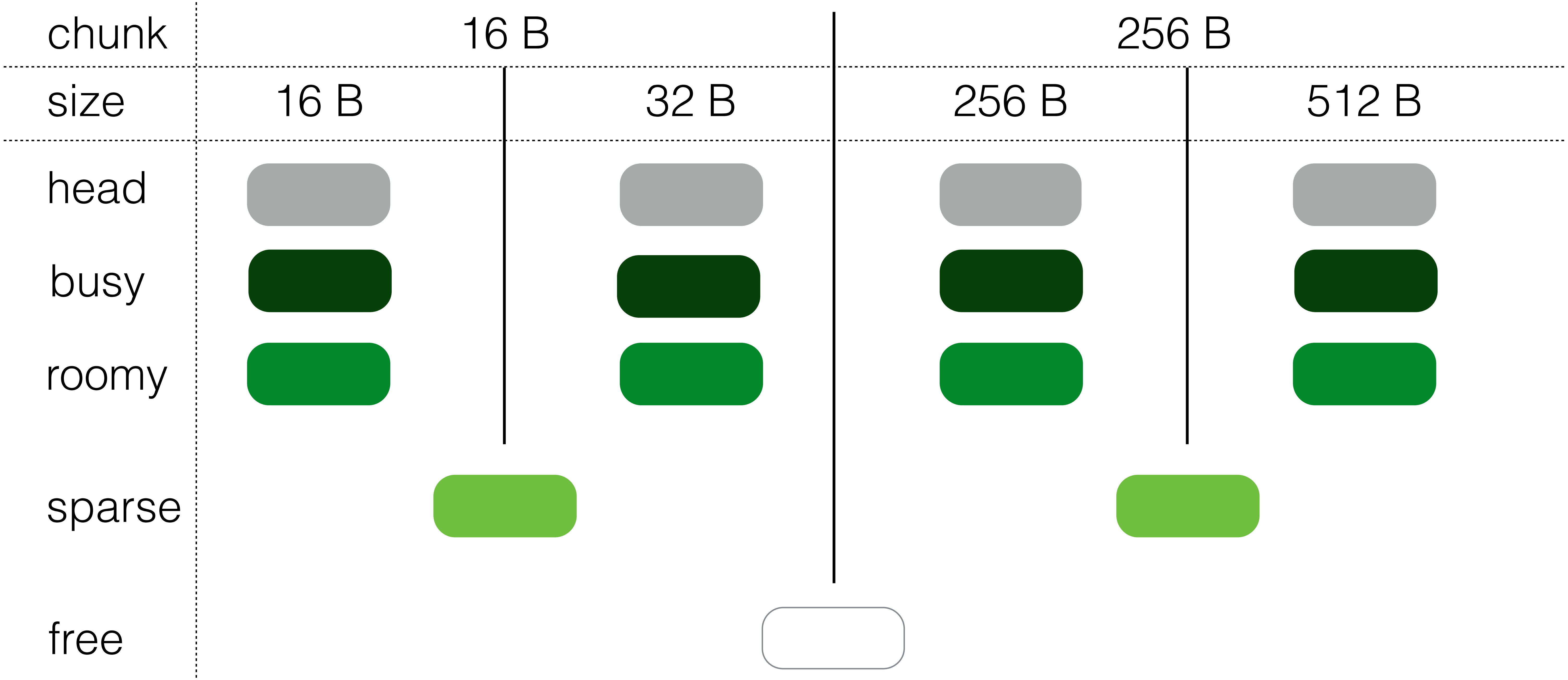
only heads used for allocation

normally not used in head search
(except when no other blocks)

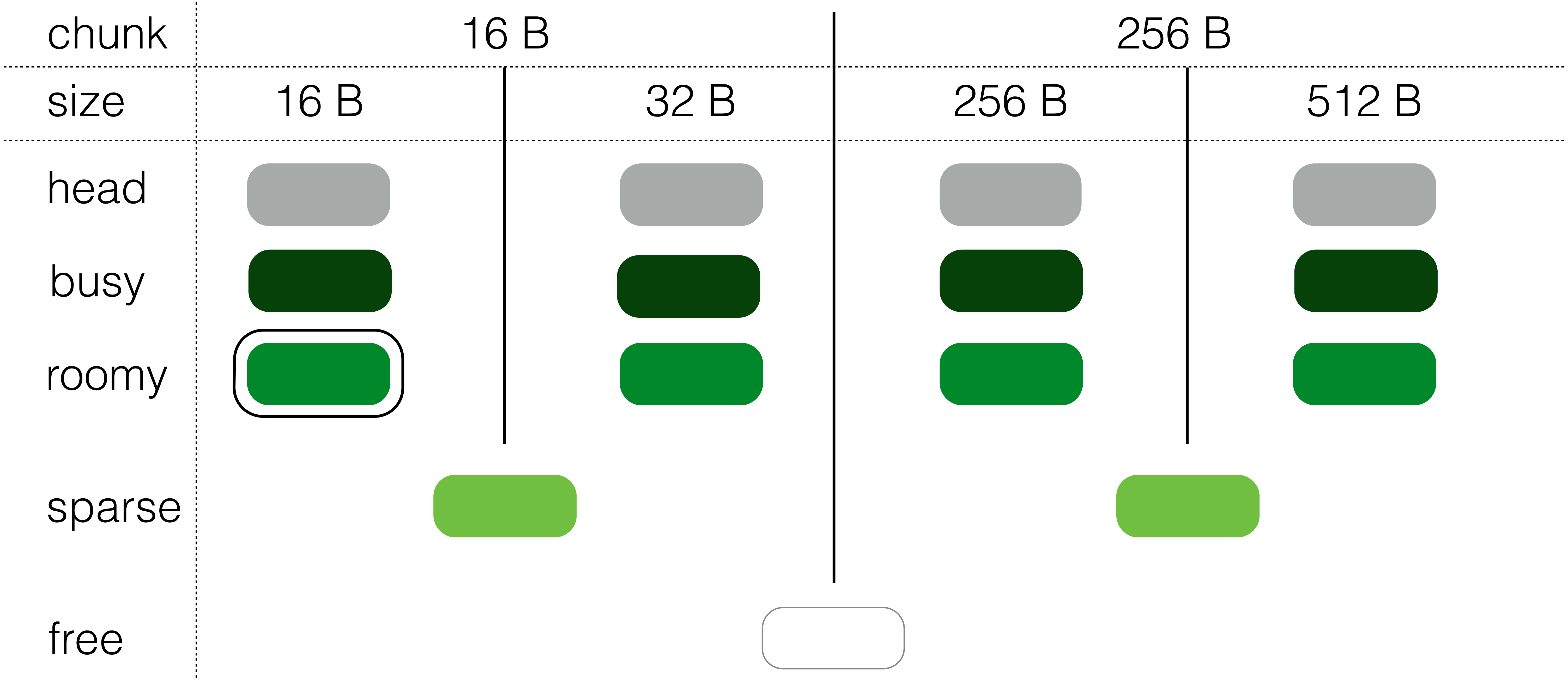
switch between block sizes (within same chunk)

can switch between chunk sizes

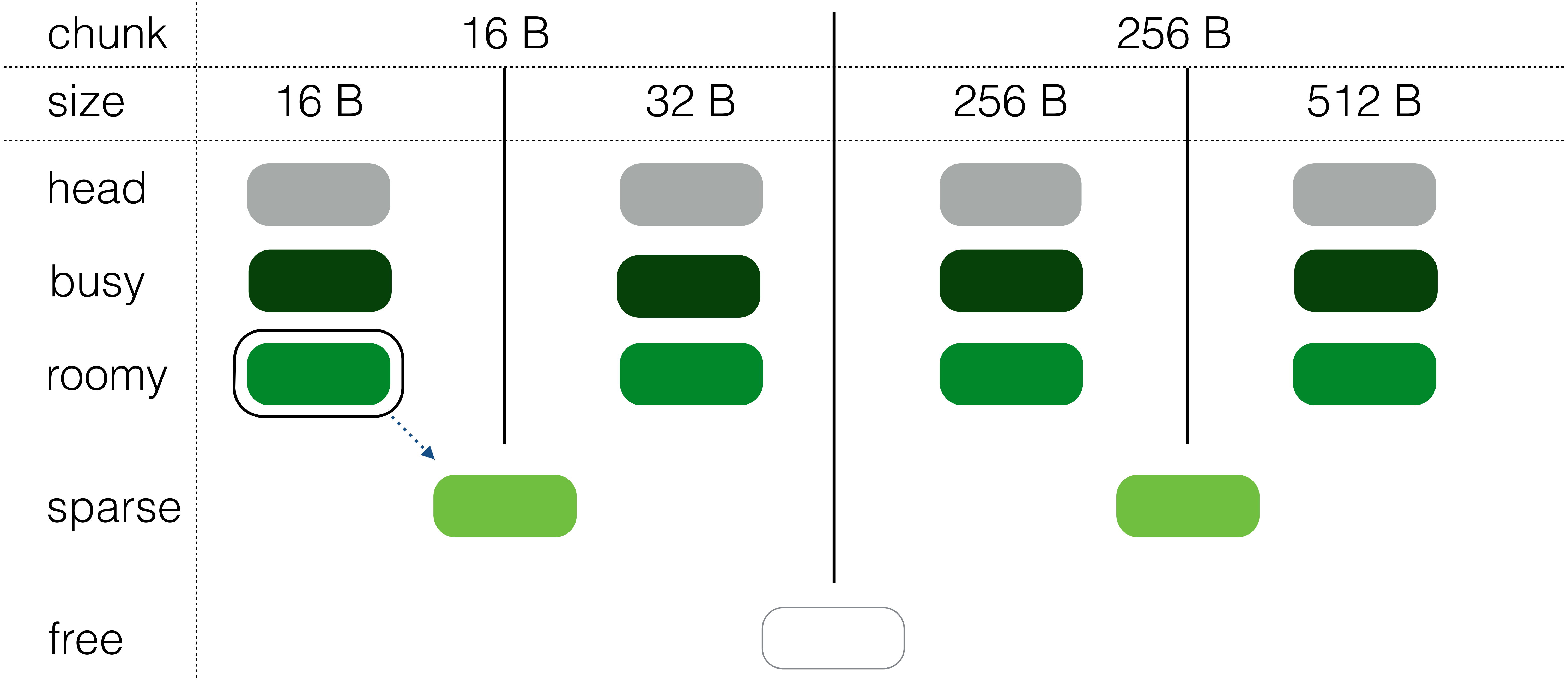
Slab Search Order



Slab Search Order

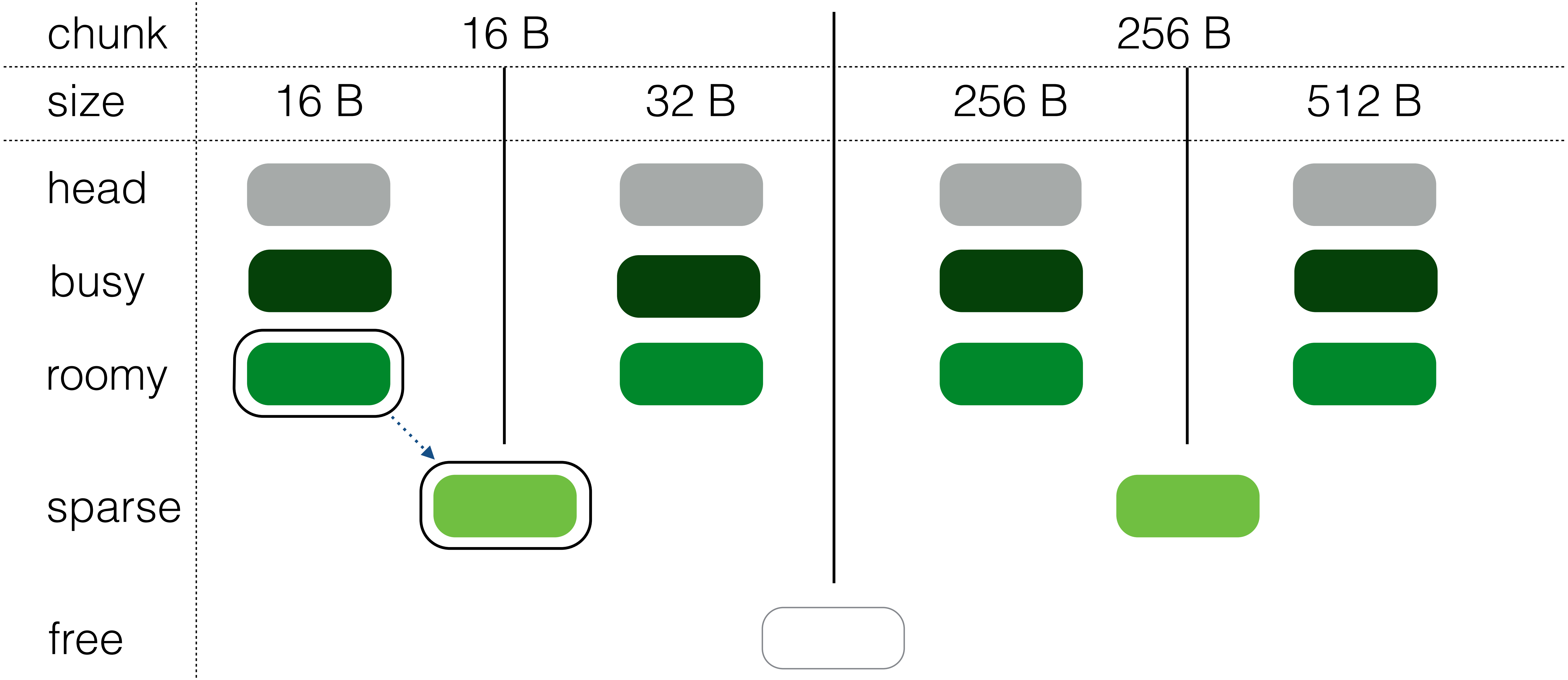


Slab Search Order

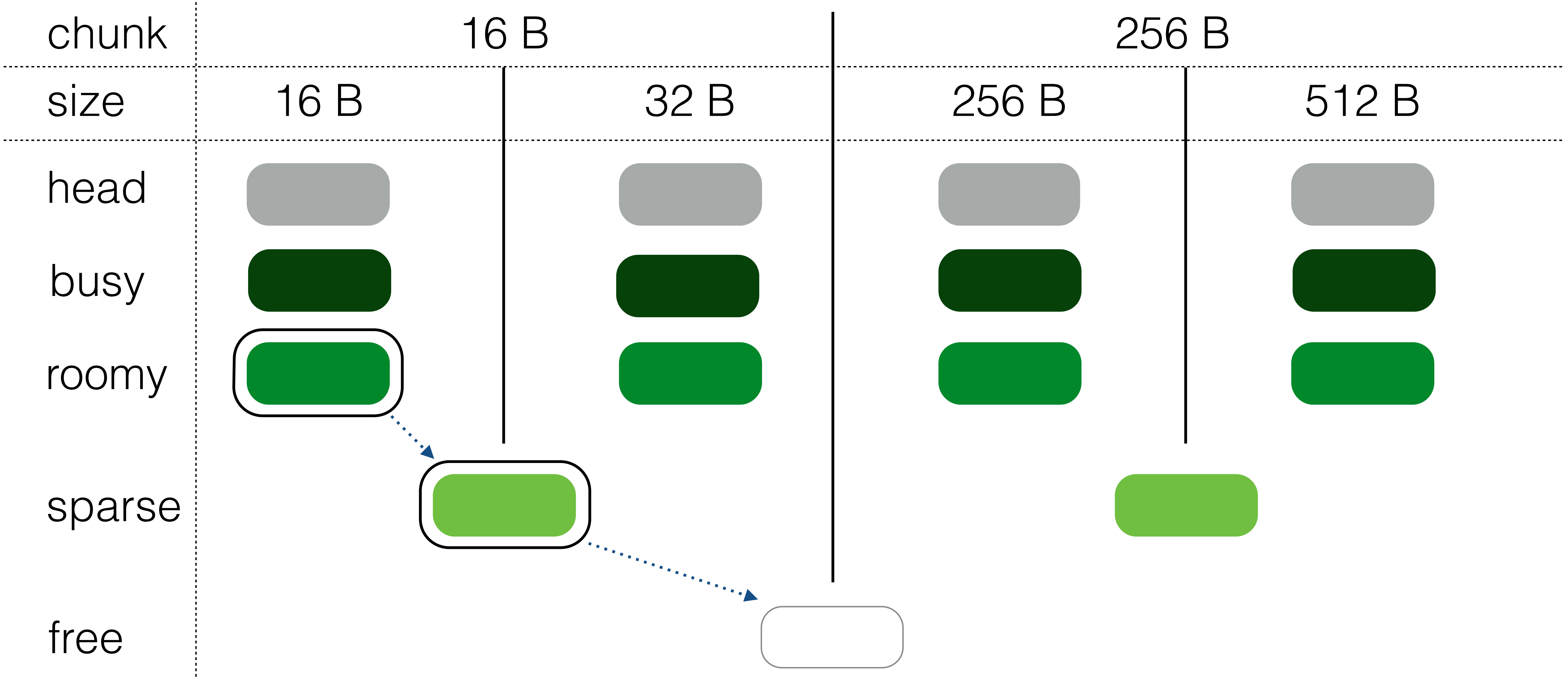


JÜLICH
FORSCHUNGSZENTRUM

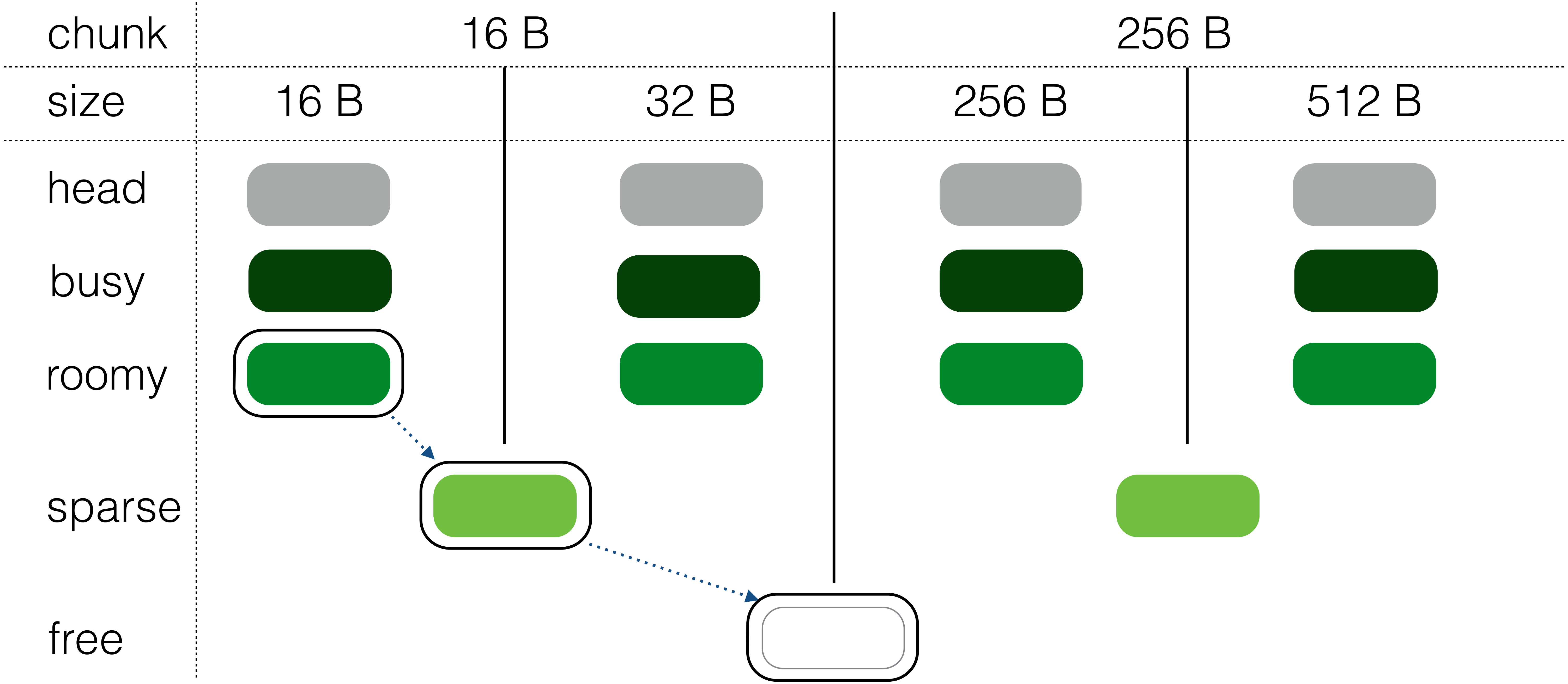
Slab Search Order



Slab Search Order

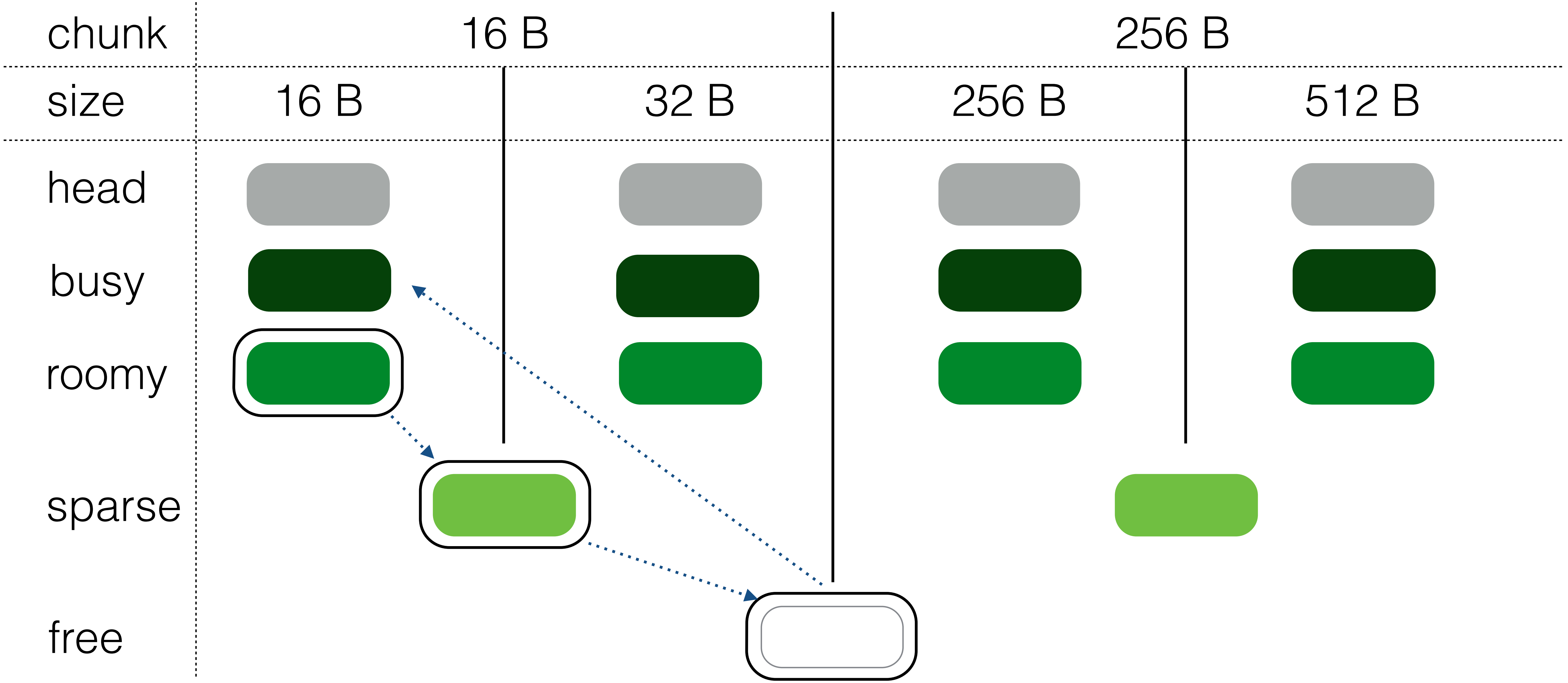


Slab Search Order

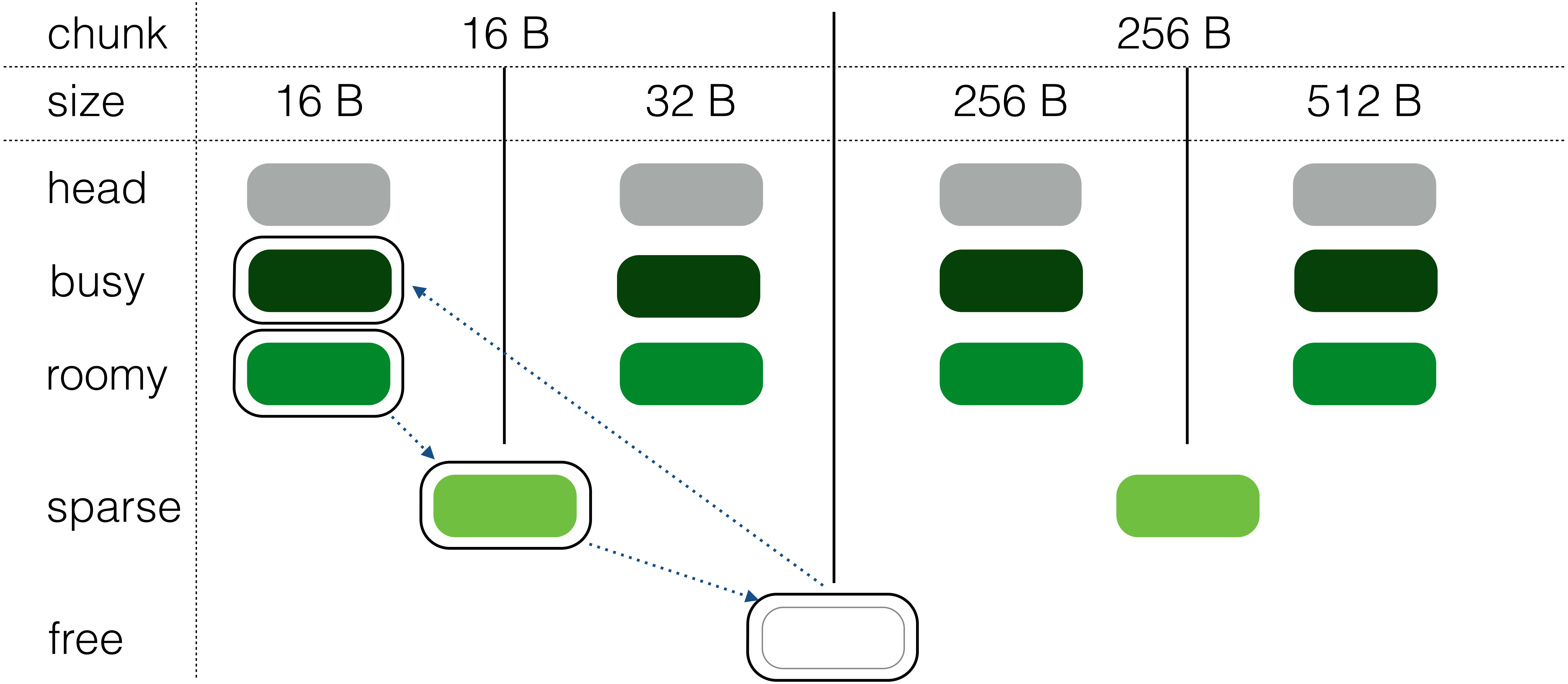


JÜLICH FORSCHUNGSZENTRUM

Slab Search Order



Slab Search Order



Slab Sets

	0	1	0	0	1	1	0	0	0	1	1	1	0	0	0	0	1	1	

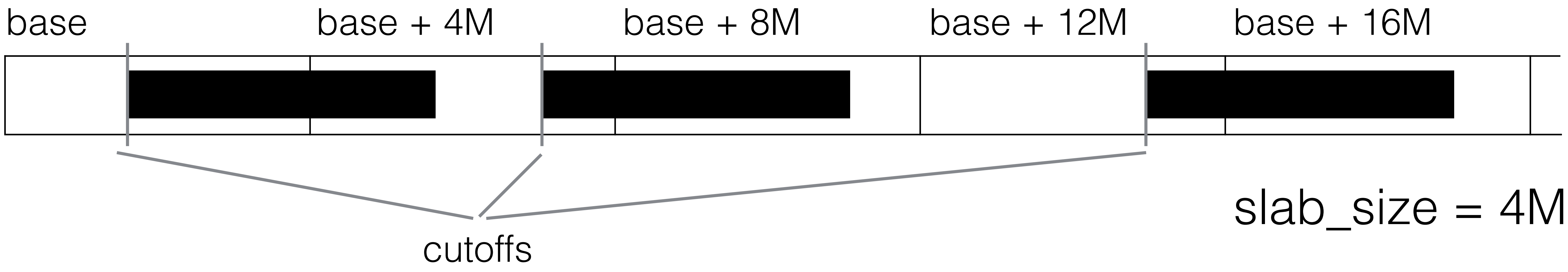
- Bitmaps
  - add(), remove(): atomicOr, atomicAnd
  - get_any(): scan the bit array, try to remove if 1
    - O(100) slabs
    - only small number of words to check
- Bitmap is an “upper bound”
  - can contain false positives
  - slab always locked before checking
- Set of free slabs is “exact”

Head Replacement Tricks

- While hashing
  - periodically check counter
- Start early
  - replace when slab counter steps over threshold (> 83.5 %)
  - even if allocation successful
  - some threads can still allocate from old head
- Overlap replacement with allocation
  - take new head from cache …
  - … so that other threads start allocating …
  - find new head for cache

Slab Grid

How to find a slab to which the pointer belongs?

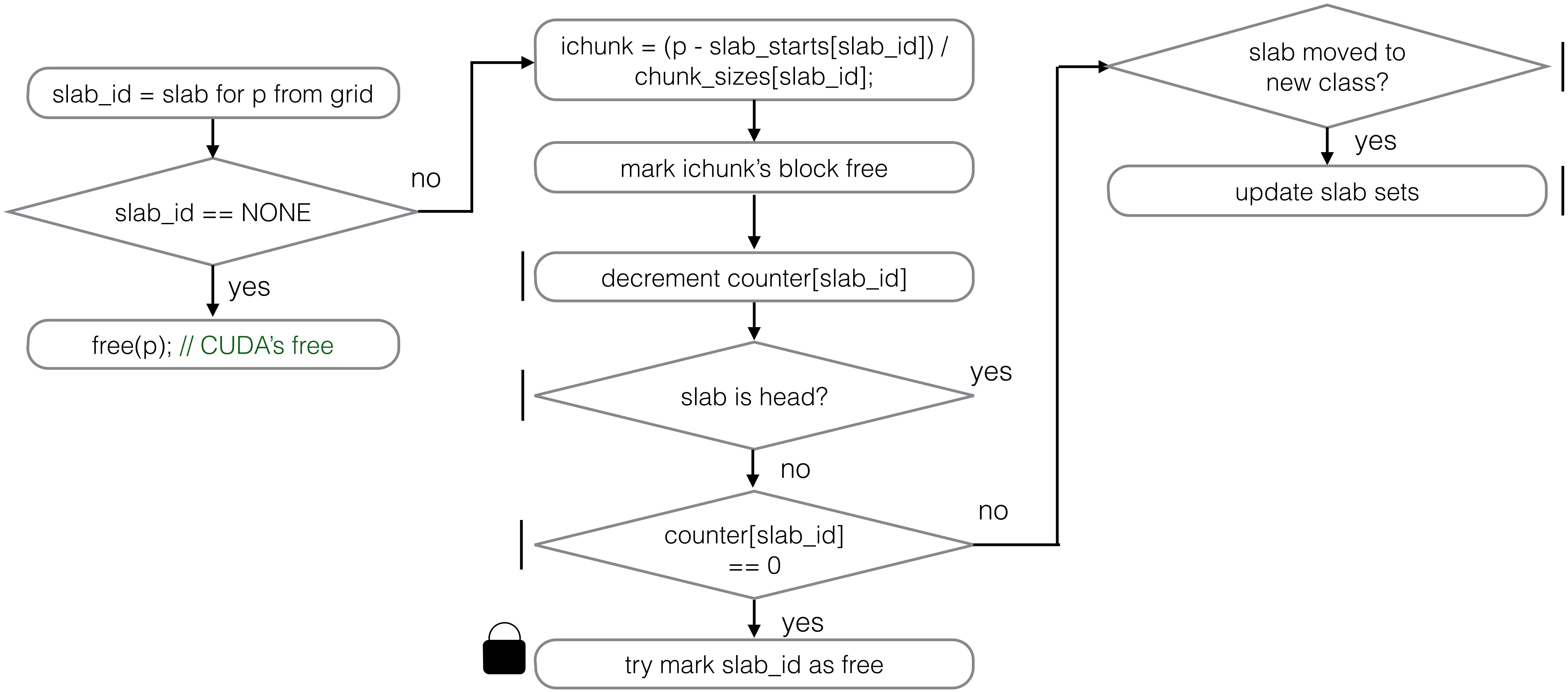



```
icell = (ptr - base) / slab_size;
slab_id = ptr < cutoffs[icell] ?
  first_slabs[icell] : second_slabs[icell];
```

Assumptions:

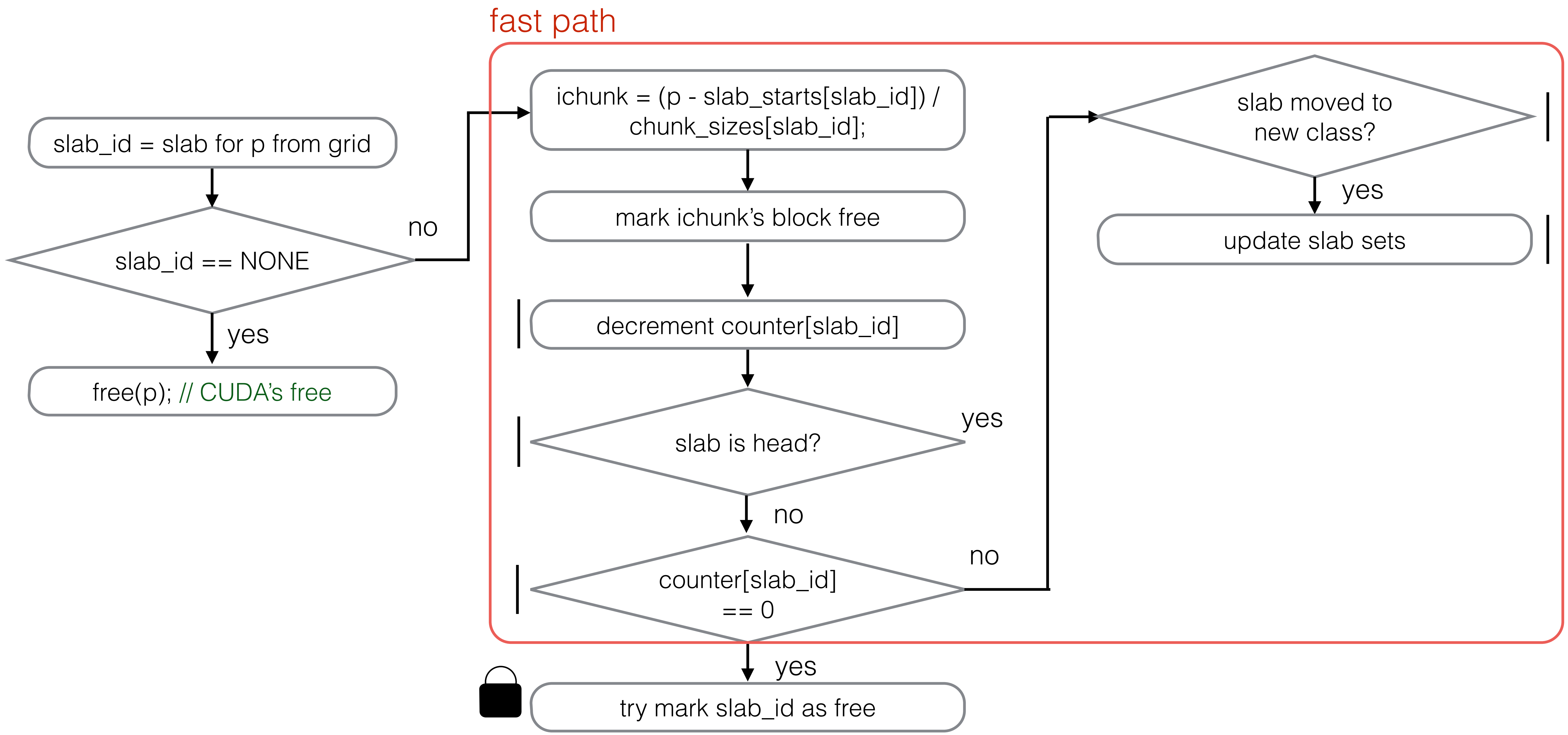
- contiguous GPU addresses
- slabs not perfectly aligned

void hafree(void *p)

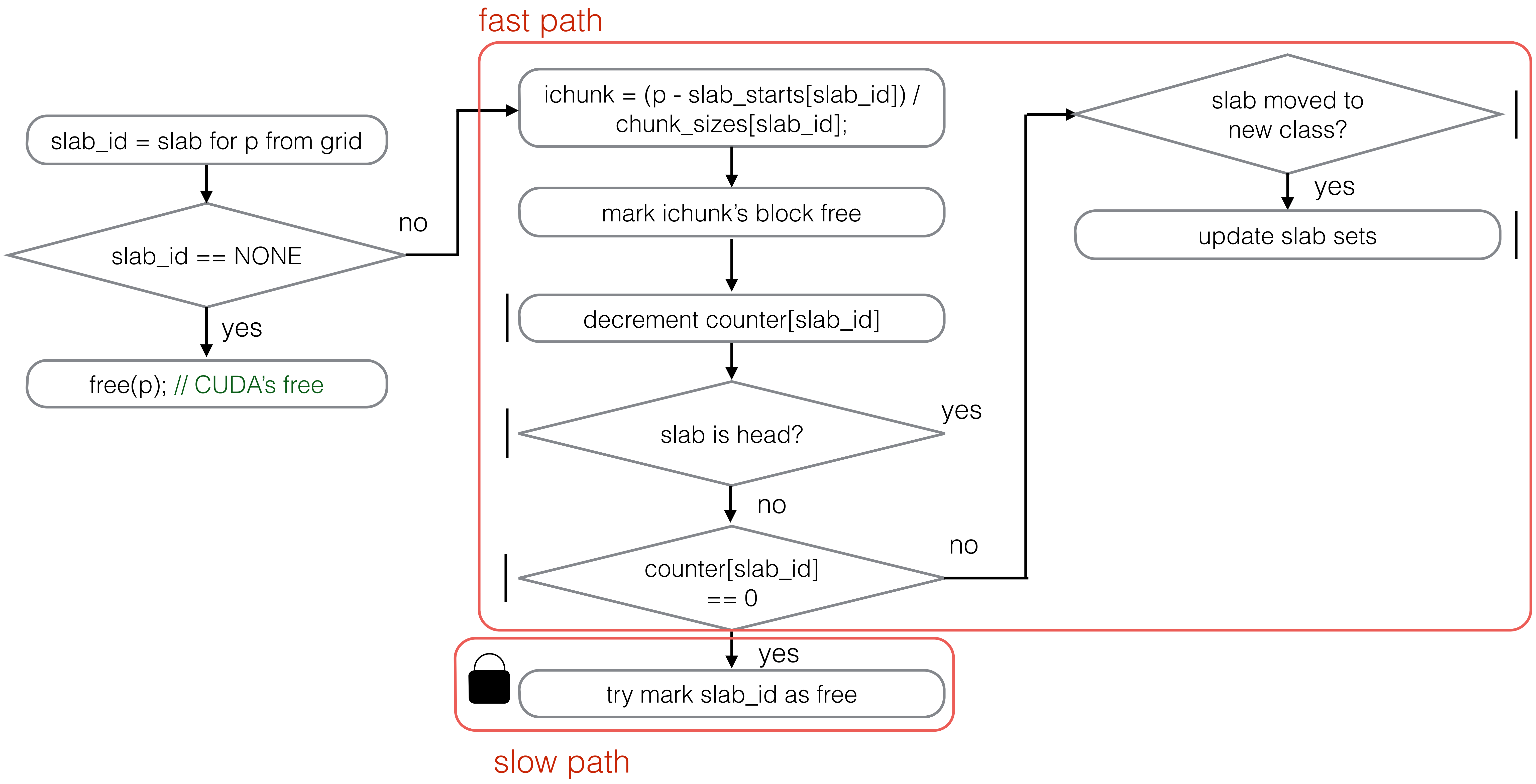


Mitglied der Helmholtz-Gemeinschaft

void hafree(void *p)



void hafree(void *p)



Outline

- Motivation
- Halloc’s main idea
- Aspects of halloc design
- **Performance benchmarks**

Benchmark

- Two-phase test
  - allocate phase / free phase
- Randomized test
  - “real-world behaviour”

Benchmark

- Two-phase test
  – allocate phase / free phase
- Randomized test
  – “real-world behaviour”

```
if(!allocd && rnd() < probab[phase][0]) {
  p = hamalloc(size);
  allocd = true;
} else if(allocd && rnd() < probab[phase][1]) {
  hafree(p);
  allocd = false;
}
phase = 1 - phase;
```

x #threads

Benchmark

- Two-phase test
  - allocate phase / free phase
- Randomized test
  - “real-world behaviour”

	even	odd
allocate	0.95	0.55
free	0.55	0.95

```
if(!allocd && rnd() < probab[phase][0]) {
  p = hamalloc(size);
  allocd = true;
} else if(allocd && rnd() < probab[phase][1]) {
  hafree(p);
  allocd = false;
}
phase = 1 - phase;
```

x #threads

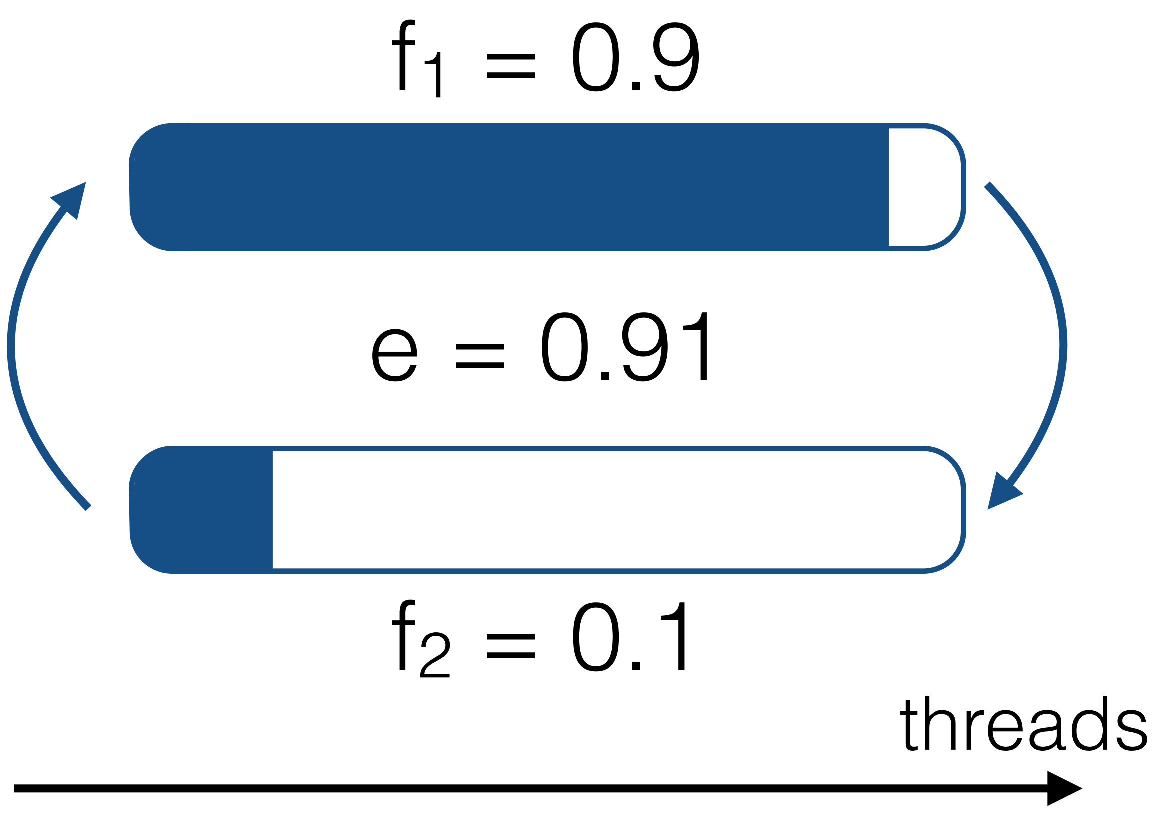
Benchmark

- Two-phase test
  - allocate phase / free phase
- Randomized test
  - "real-world behaviour"

	even	odd
allocate	0.95	0.55
free	0.55	0.95

```
if(!allocd && rnd() < probab[phase][0]) {
  p = hamalloc(size);
  allocd = true;
} else if(allocd && rnd() < probab[phase][1]) {
  hafree(p);
  allocd = false;
}
phase = 1 - phase;
```

x #threads



Benchmark Parameters & Cases

- Spree test
  - 1 iteration inside kernel
  - malloc() even, free() in odd
- Private test
  - 2 * M iterations inside kernel
  - malloc() / free() in single kernel invocation
- Other parameters
  - branch divergence: per-thread / **per-warp** / per-block
  - number of allocations
  - allocation size

Test Setup

JÜLICH
FORSCHUNGSZENTRUM

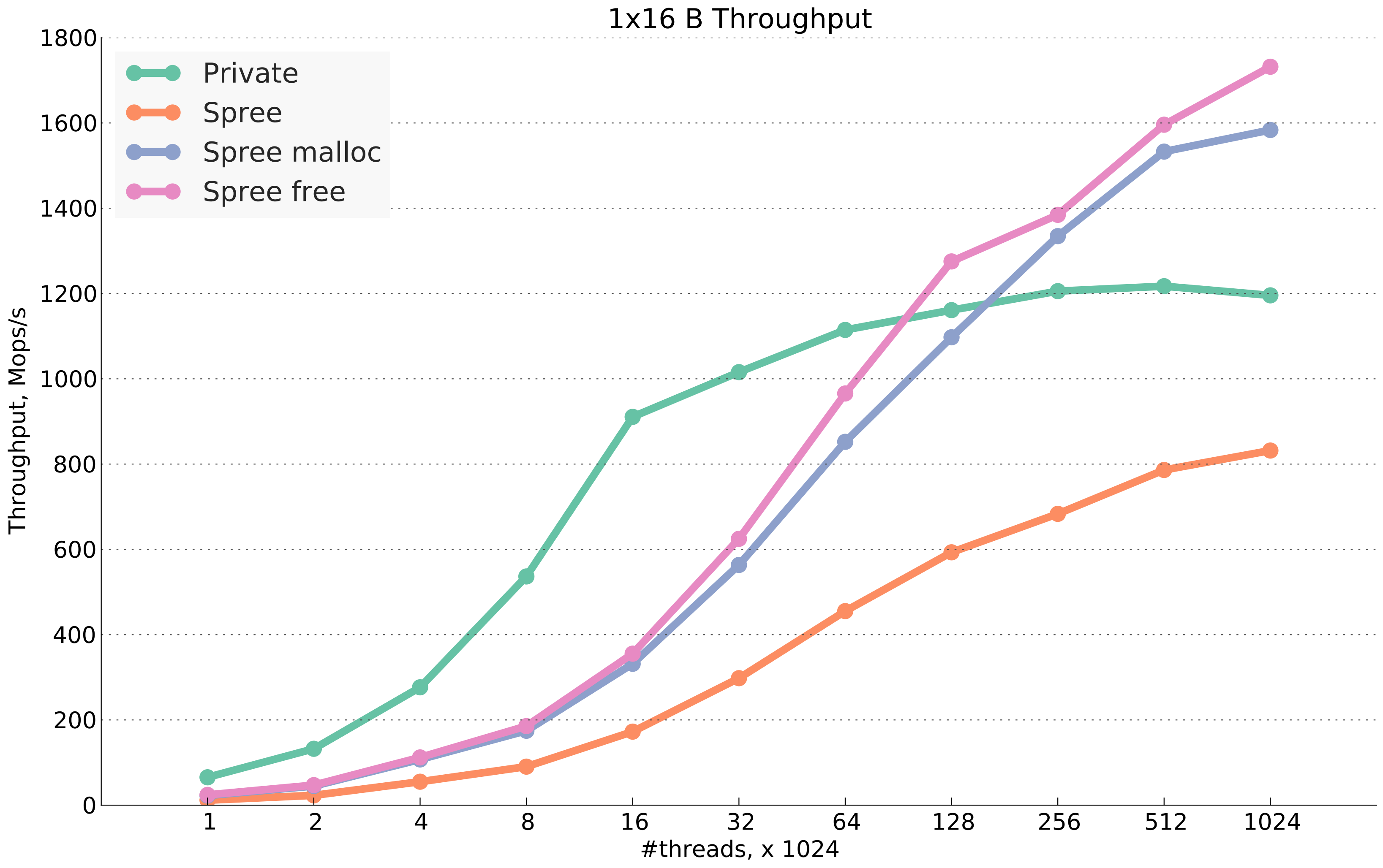
System setup

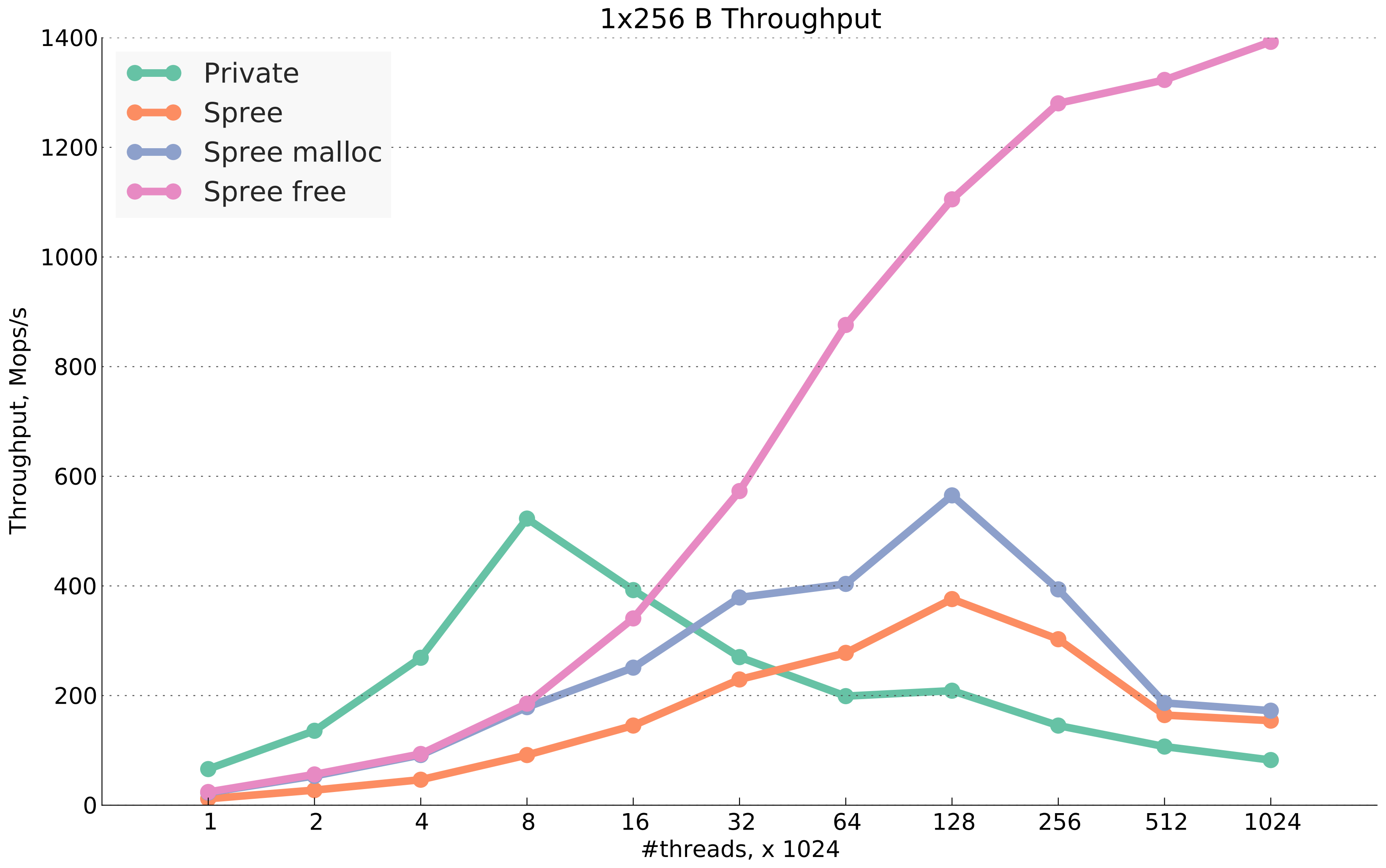
- NVidia Kepler K20X GPU
- CUDA 5.5

Allocator

- 512 MiB memory, 75% for halloc (rest for CUDA)
- slab size: 4 MiB
- max fill ratio (head replacement): 83.5%
- chunks: 16, 24, 256, 384 bytes
- blocks: 1x, 2x, 4x, 8x chunks
- linked as device library

Scalability: Throughput



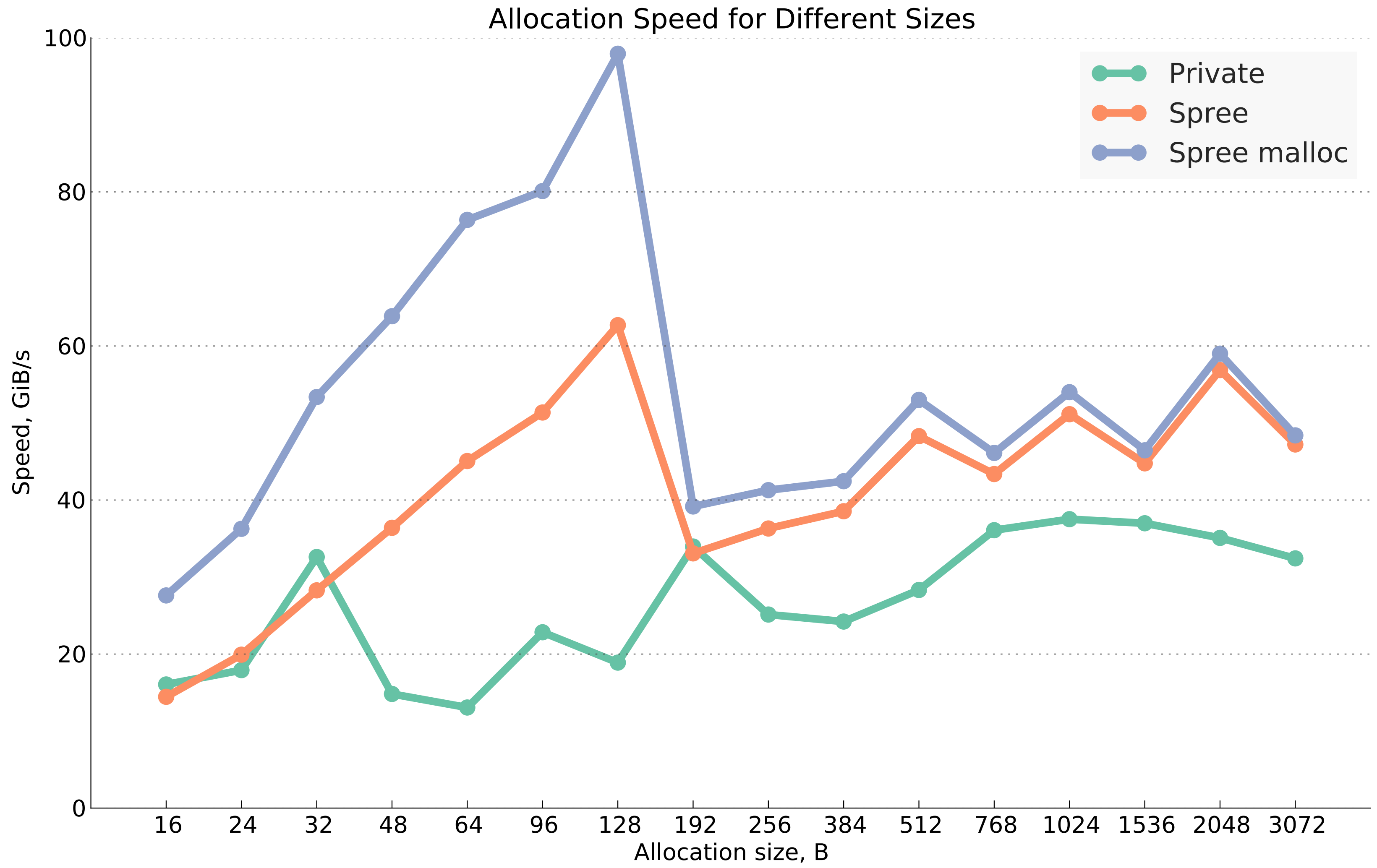


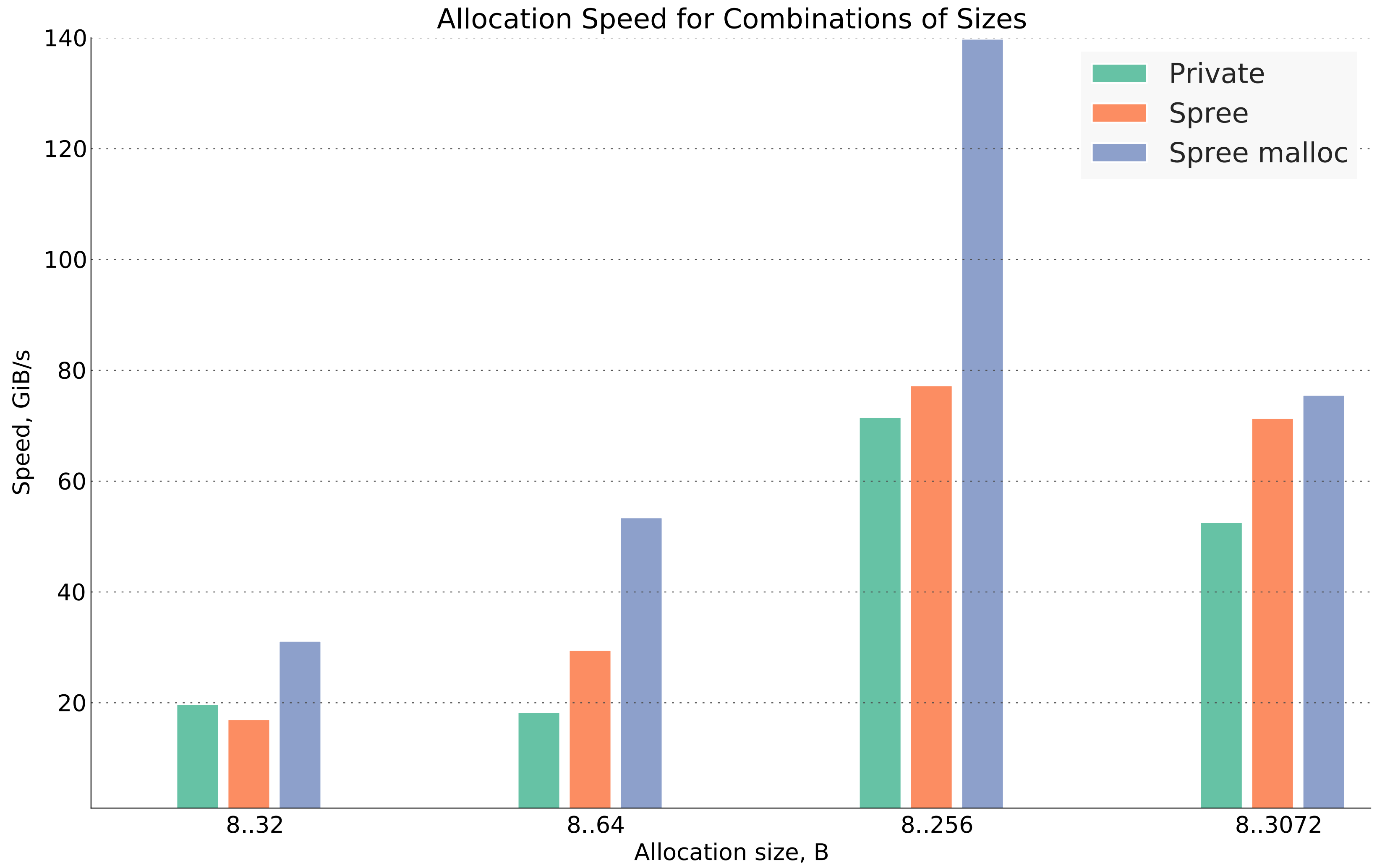
16 B blocks

256 B blocks

max 25000 simultaneous threads

Different Allocation Sizes





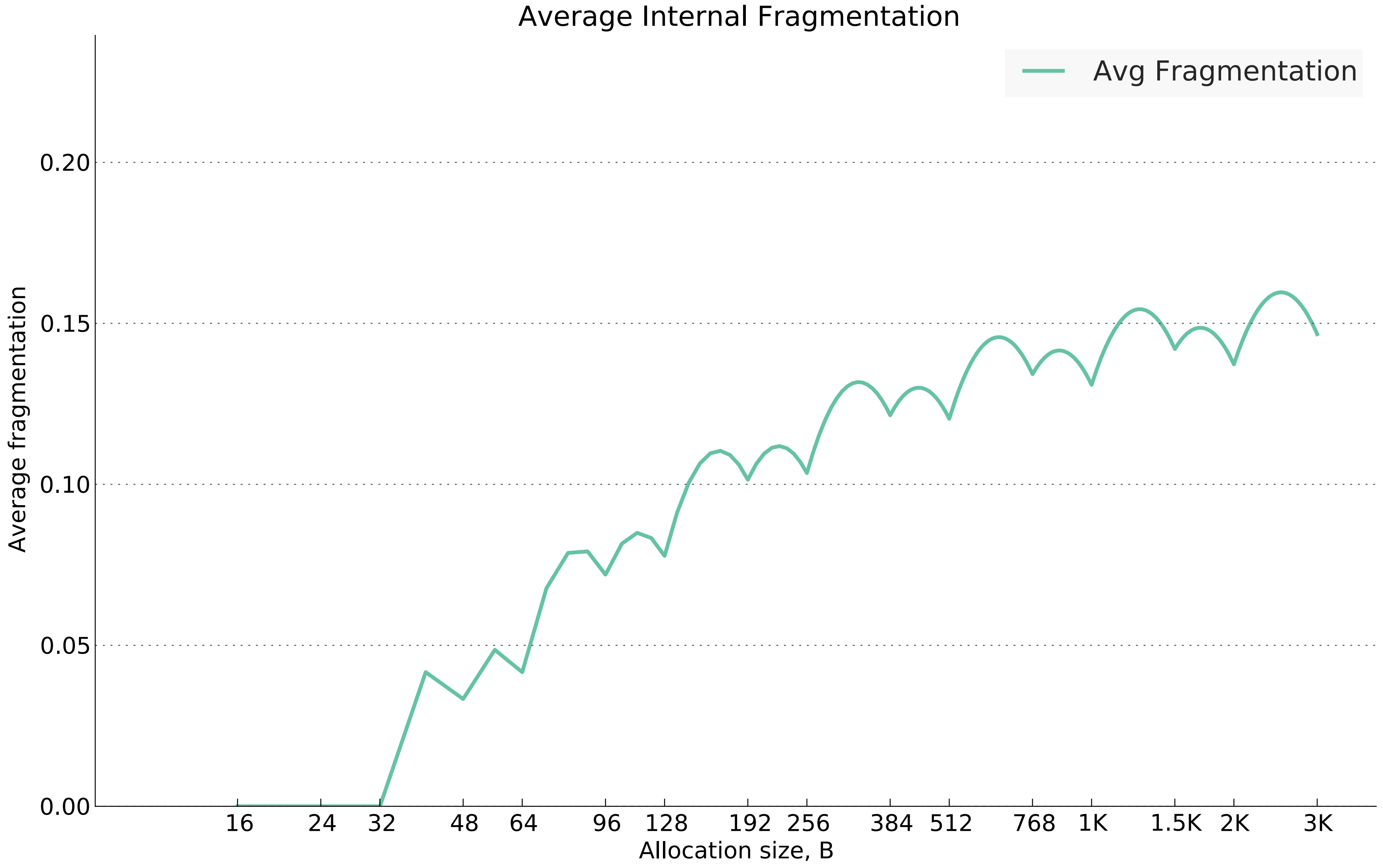
single size

mix of sizes

allocation speed stays around 15–20 GiB/s

Fragmentation

internal



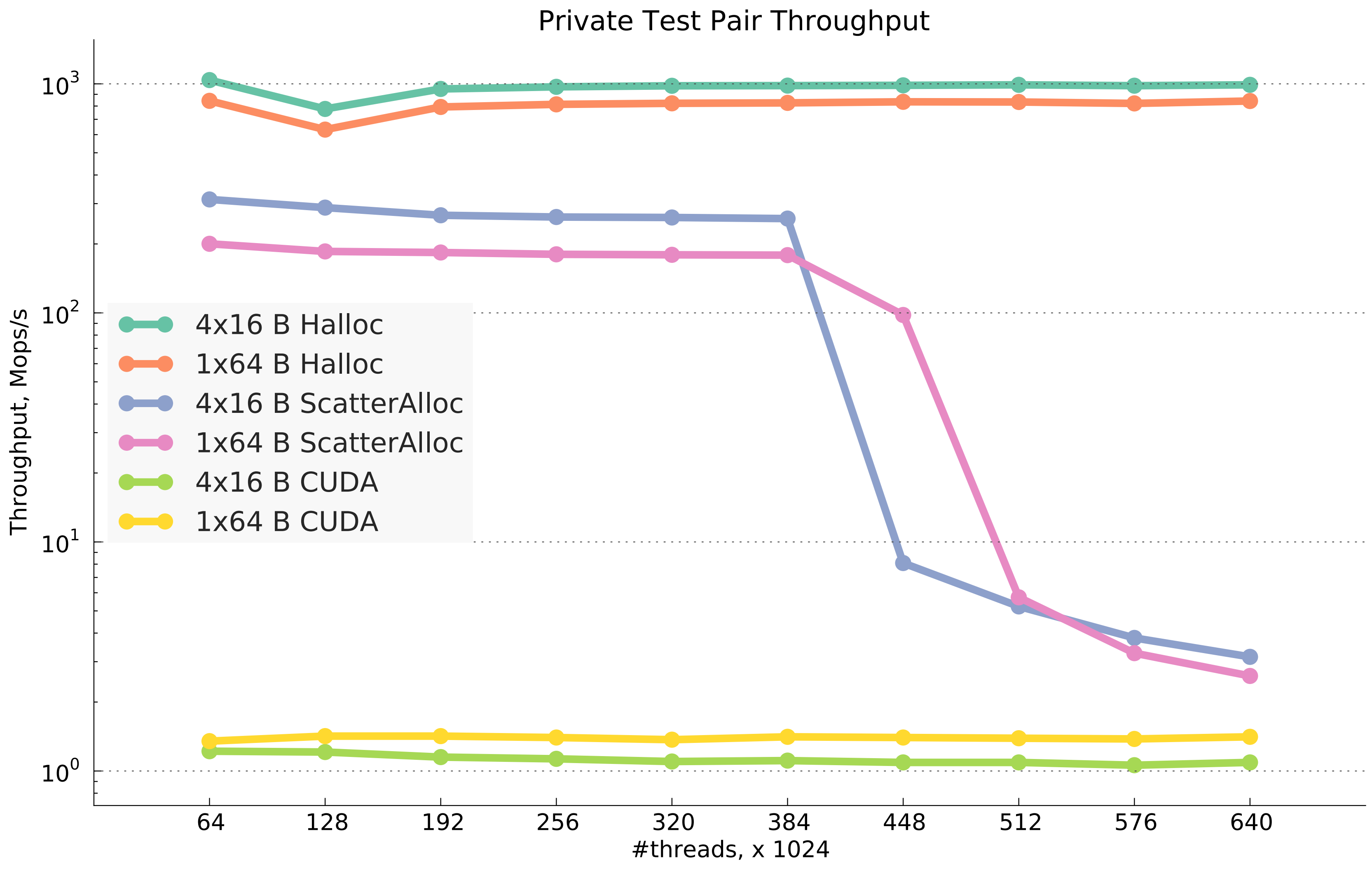
(average, size multiple of 8 B)

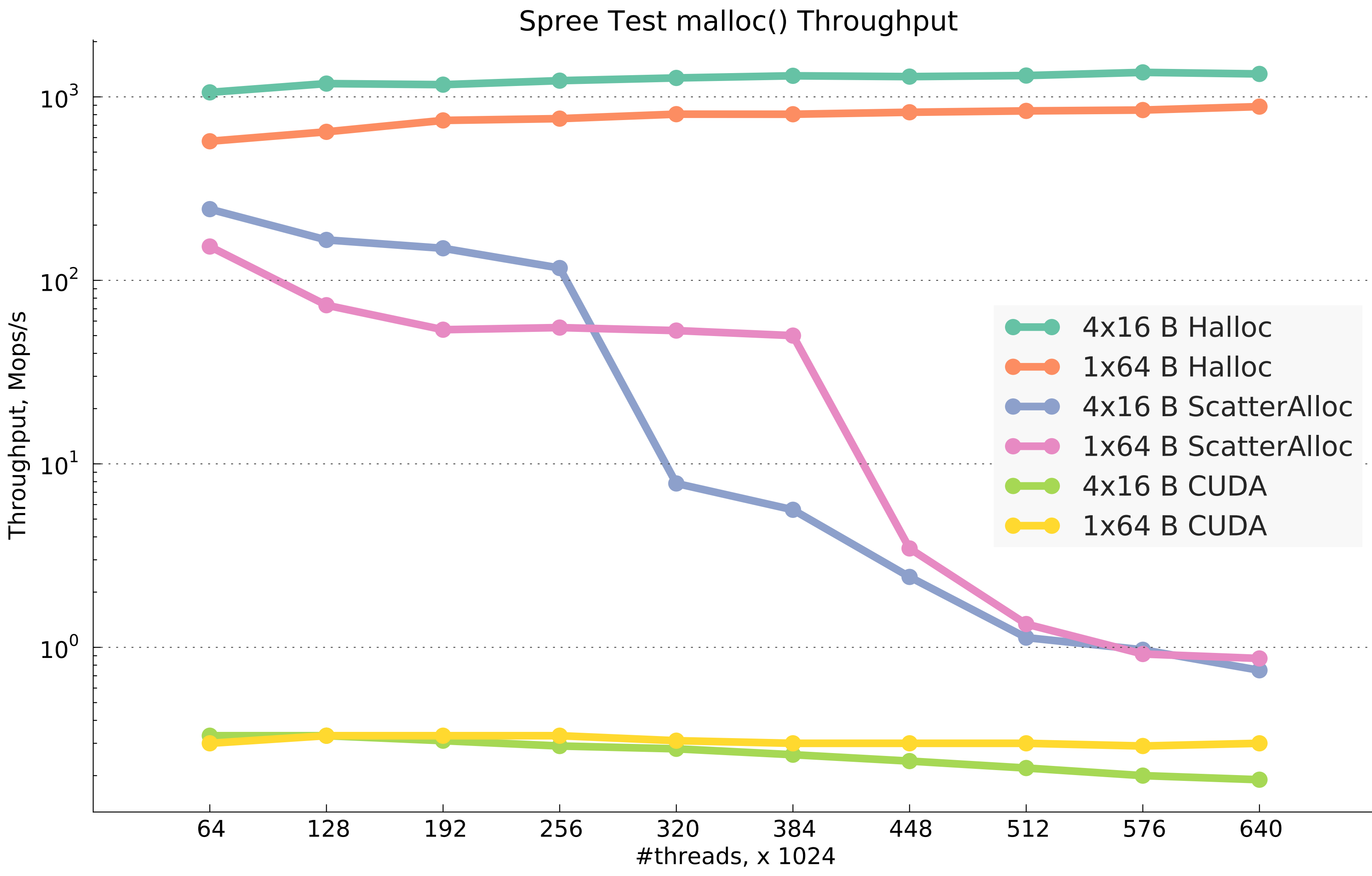
external

- Currently
  - defined by CUDA/halloc split
  - = 75%
- CUDA malloc() slabs:
  - ~ fraction of non-free slabs
- Overhead
  - only ~85% of a slab allocated

Halloc vs ScatterAlloc, CUDA's malloc

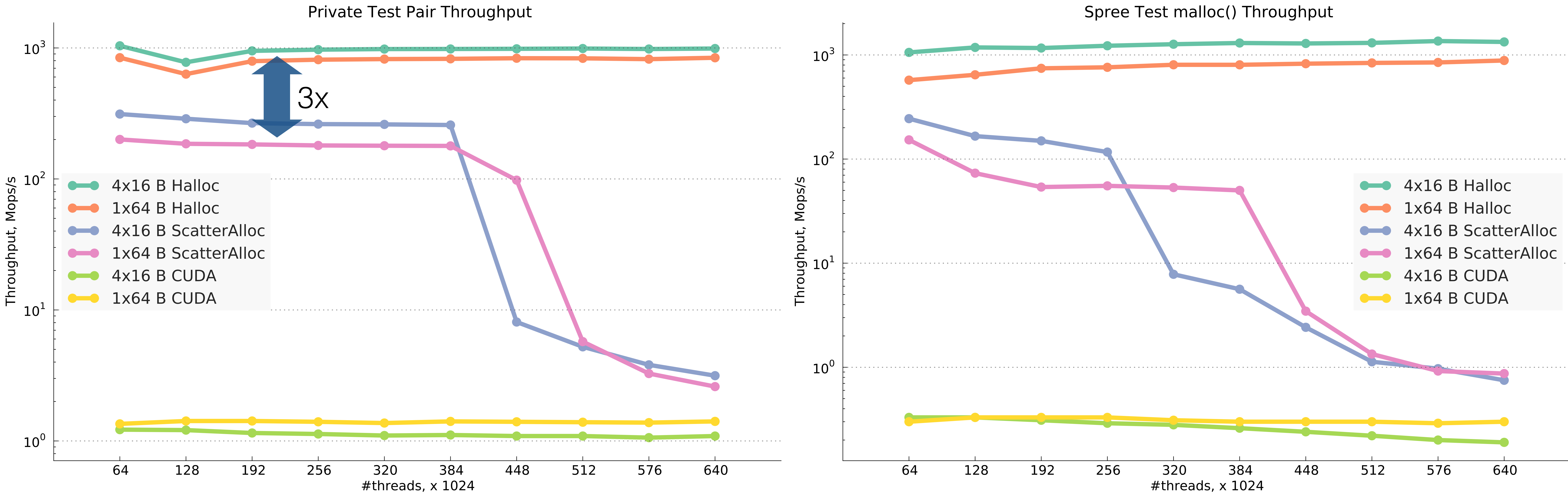
M. Steinberger et al. ScatterAlloc: Massively parallel dynamic memory allocation for the GPU. InPar–2012





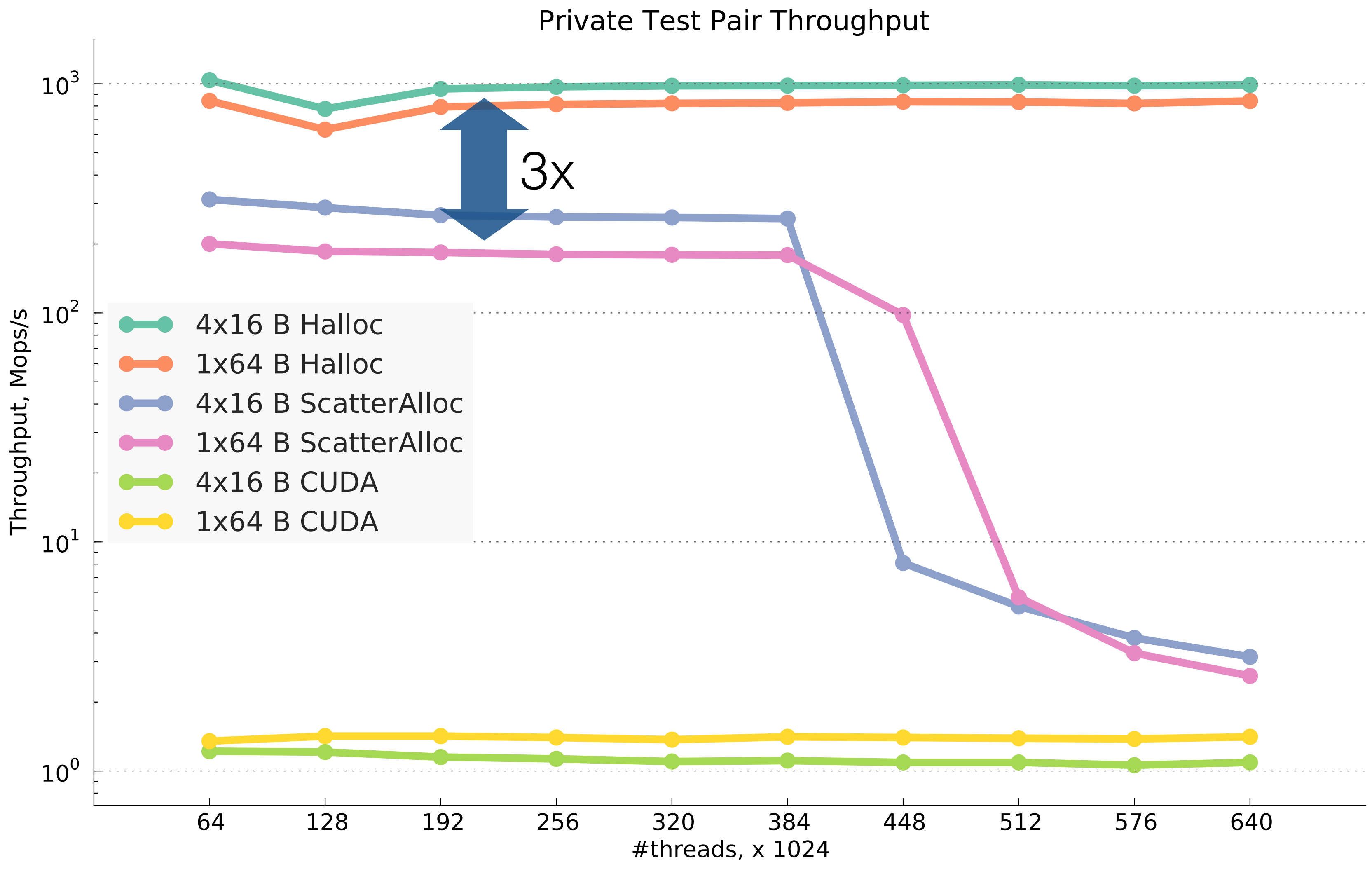
Halloc vs ScatterAlloc, CUDA's malloc

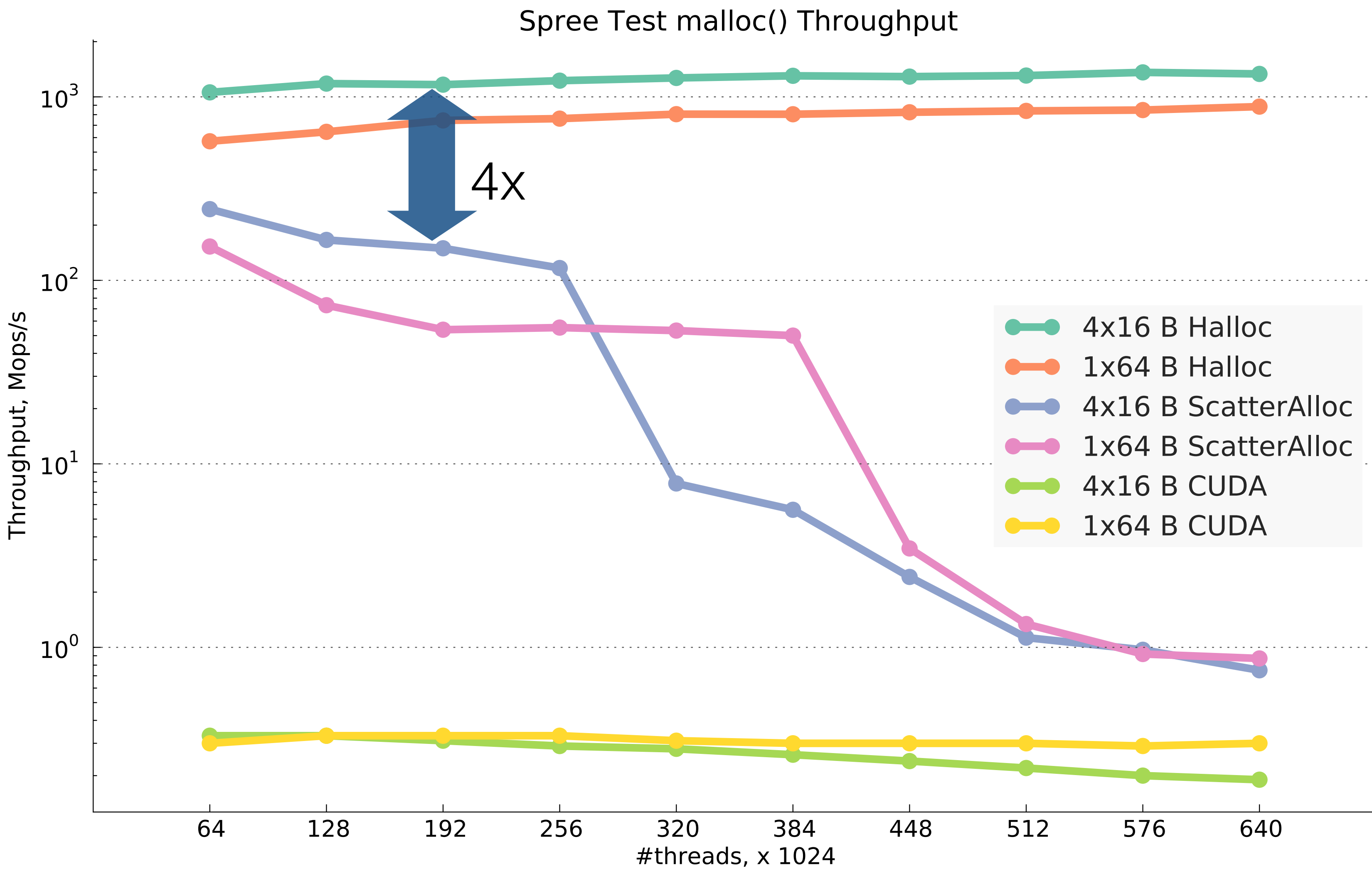
M. Steinberger et al. ScatterAlloc: Massively parallel dynamic memory allocation for the GPU. InPar–2012



Halloc vs ScatterAlloc, CUDA's malloc

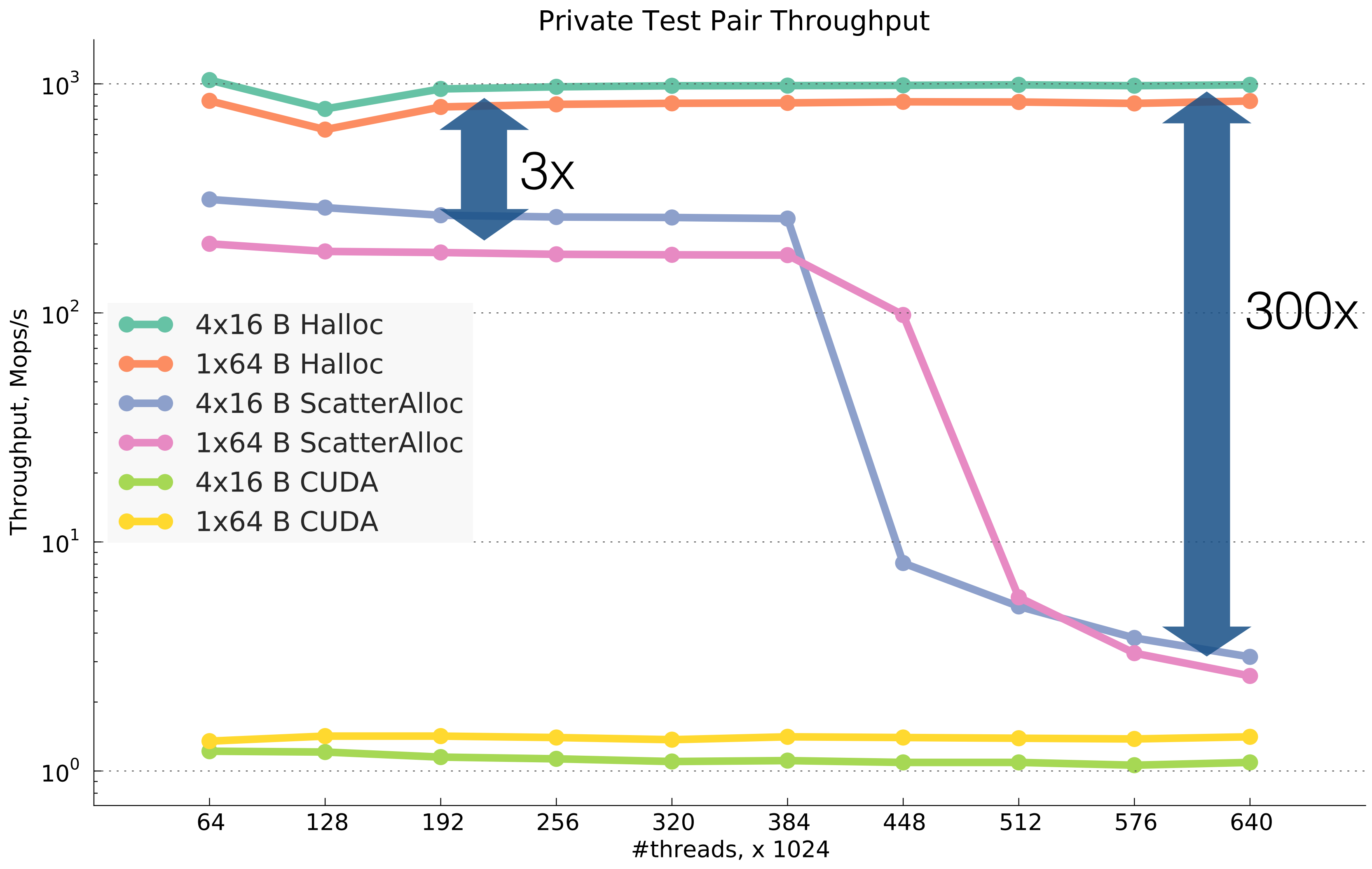
M. Steinberger et al. ScatterAlloc: Massively parallel dynamic memory allocation for the GPU. InPar–2012

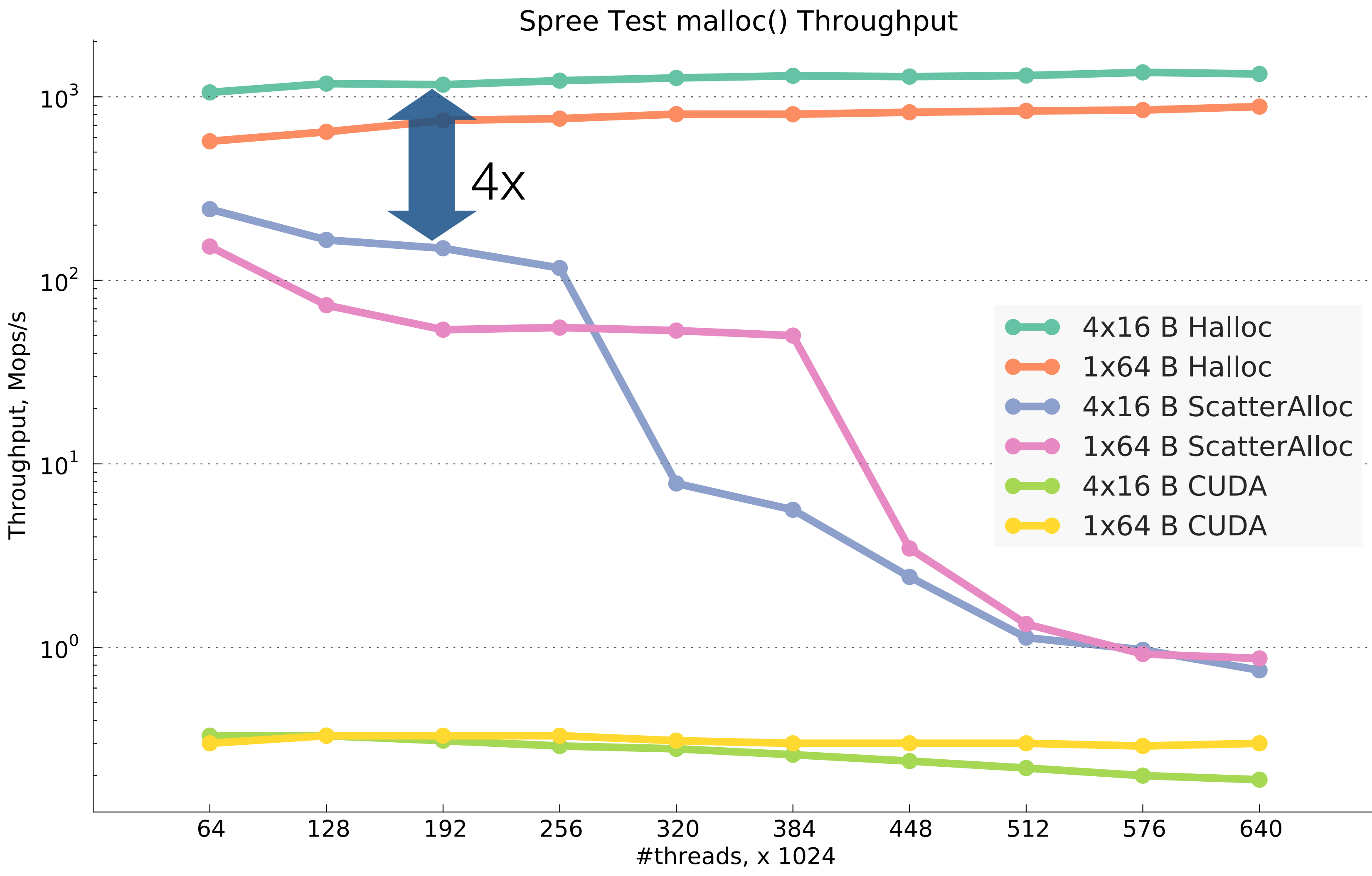




Halloc vs ScatterAlloc, CUDA's malloc

M. Steinberger et al. ScatterAlloc: Massively parallel dynamic memory allocation for the GPU. InPar–2012

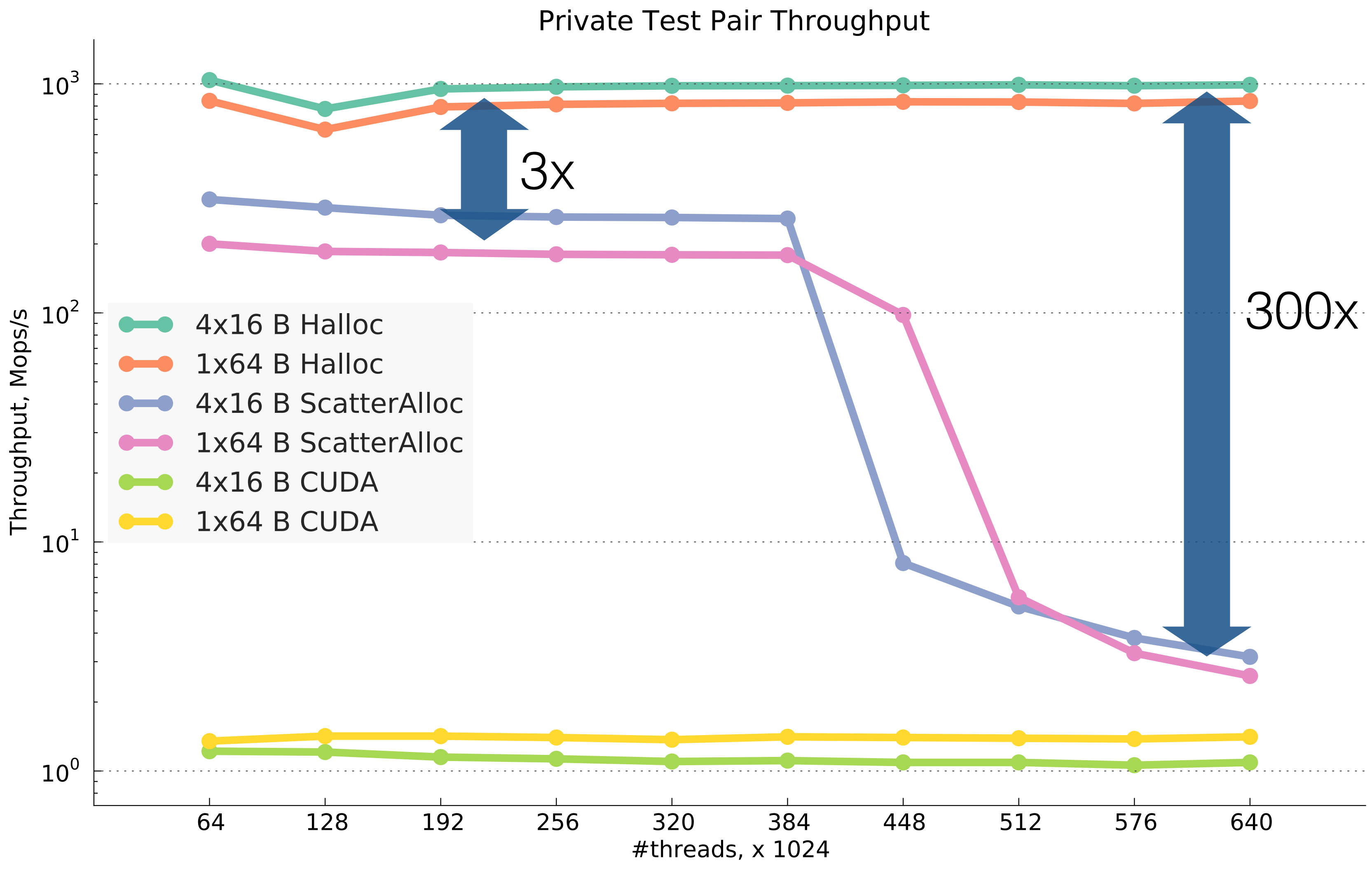


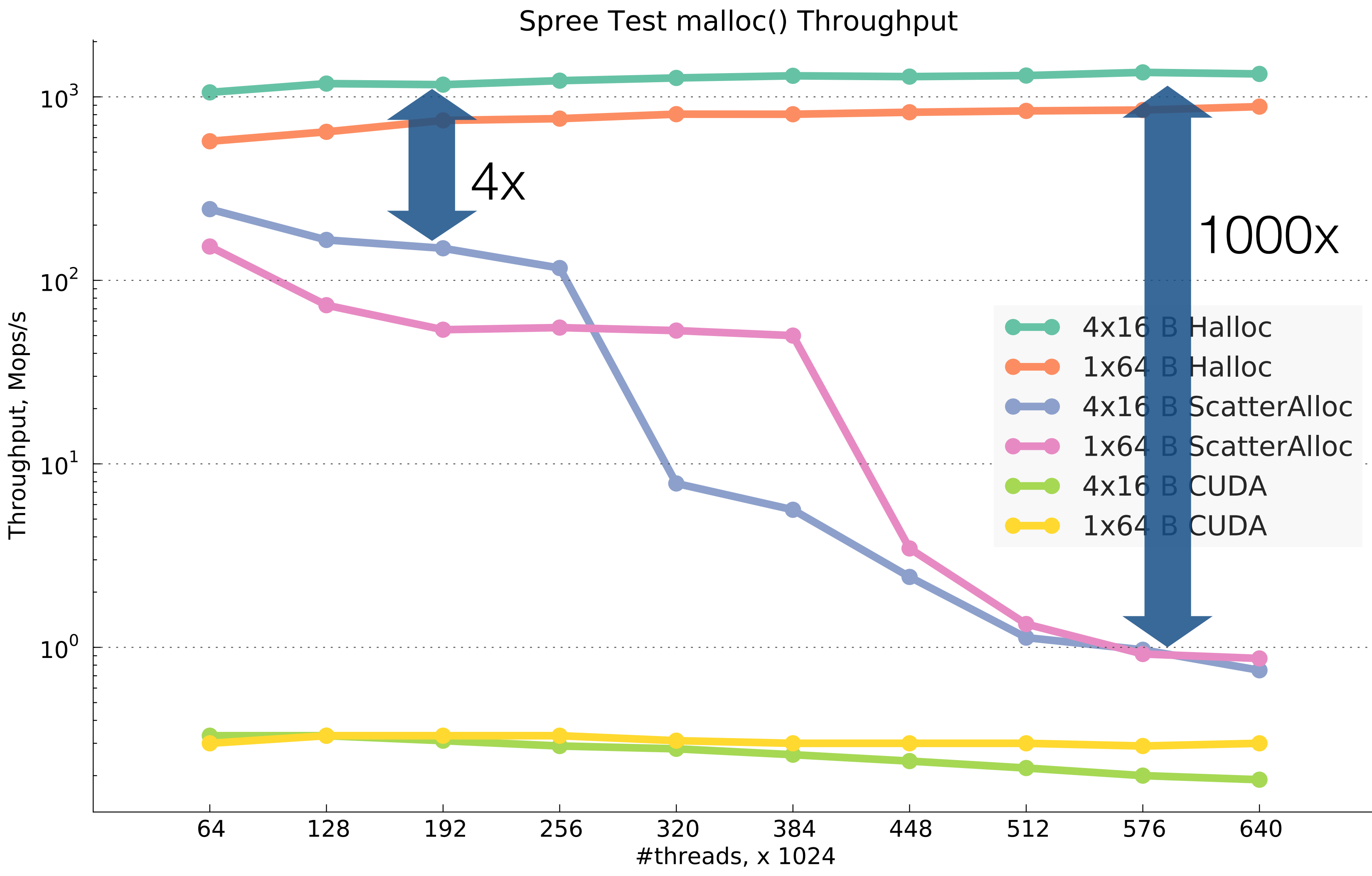


Mitglied der Helmholtz-Gemeinschaft

Halloc vs ScatterAlloc, CUDA's malloc

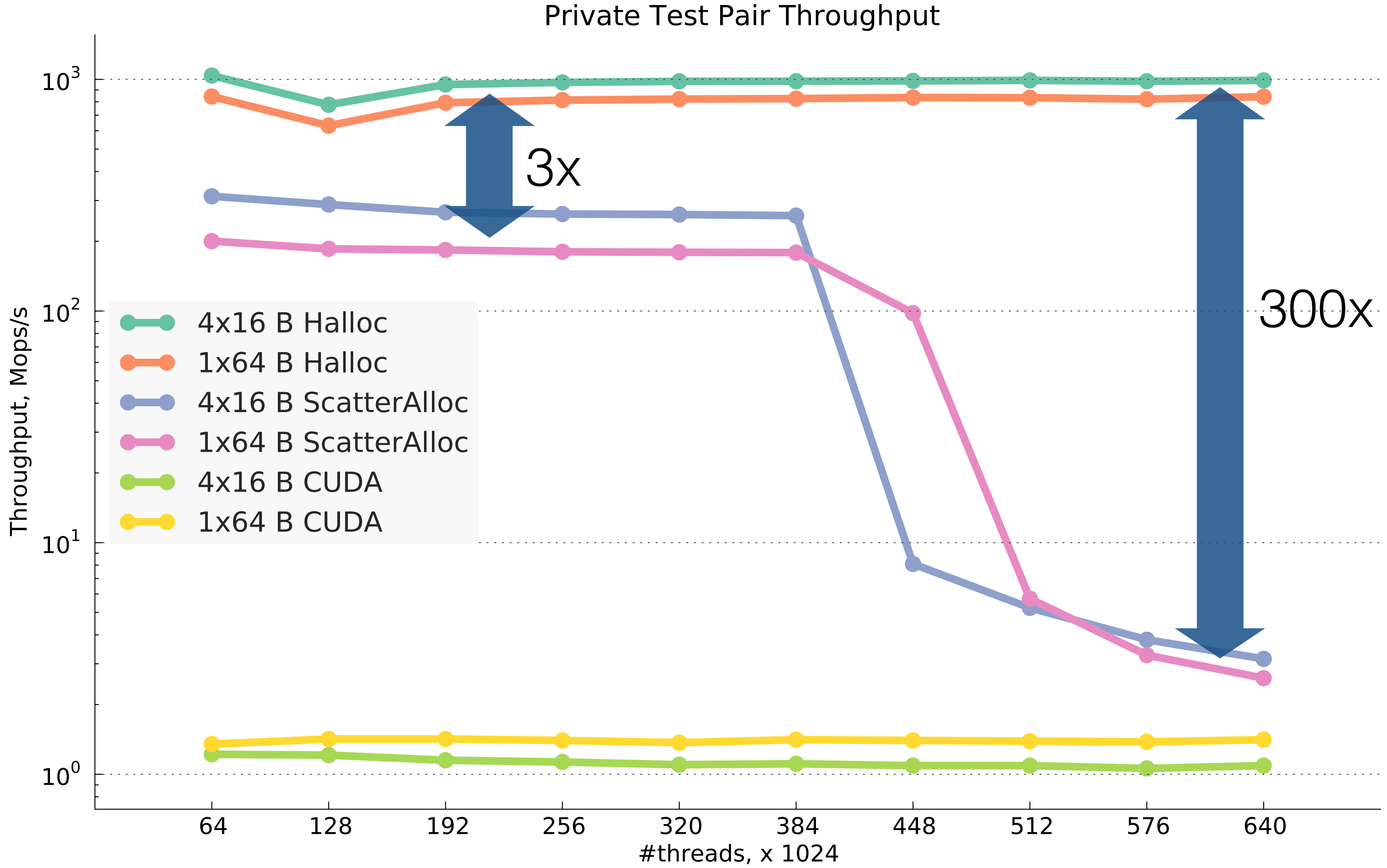
M. Steinberger et al. ScatterAlloc: Massively parallel dynamic memory allocation for the GPU. InPar–2012

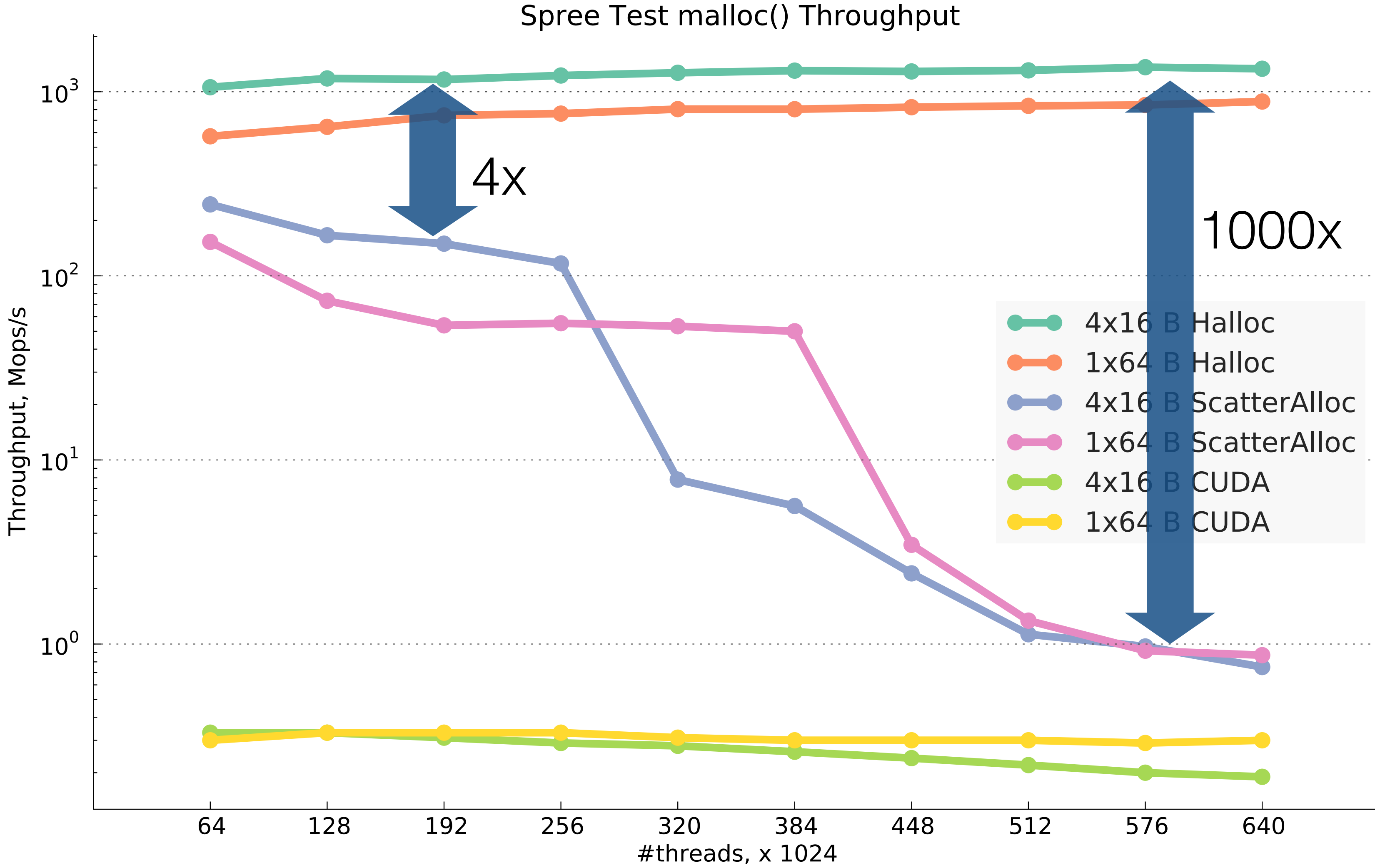




Halloc vs ScatterAlloc, CUDA's malloc

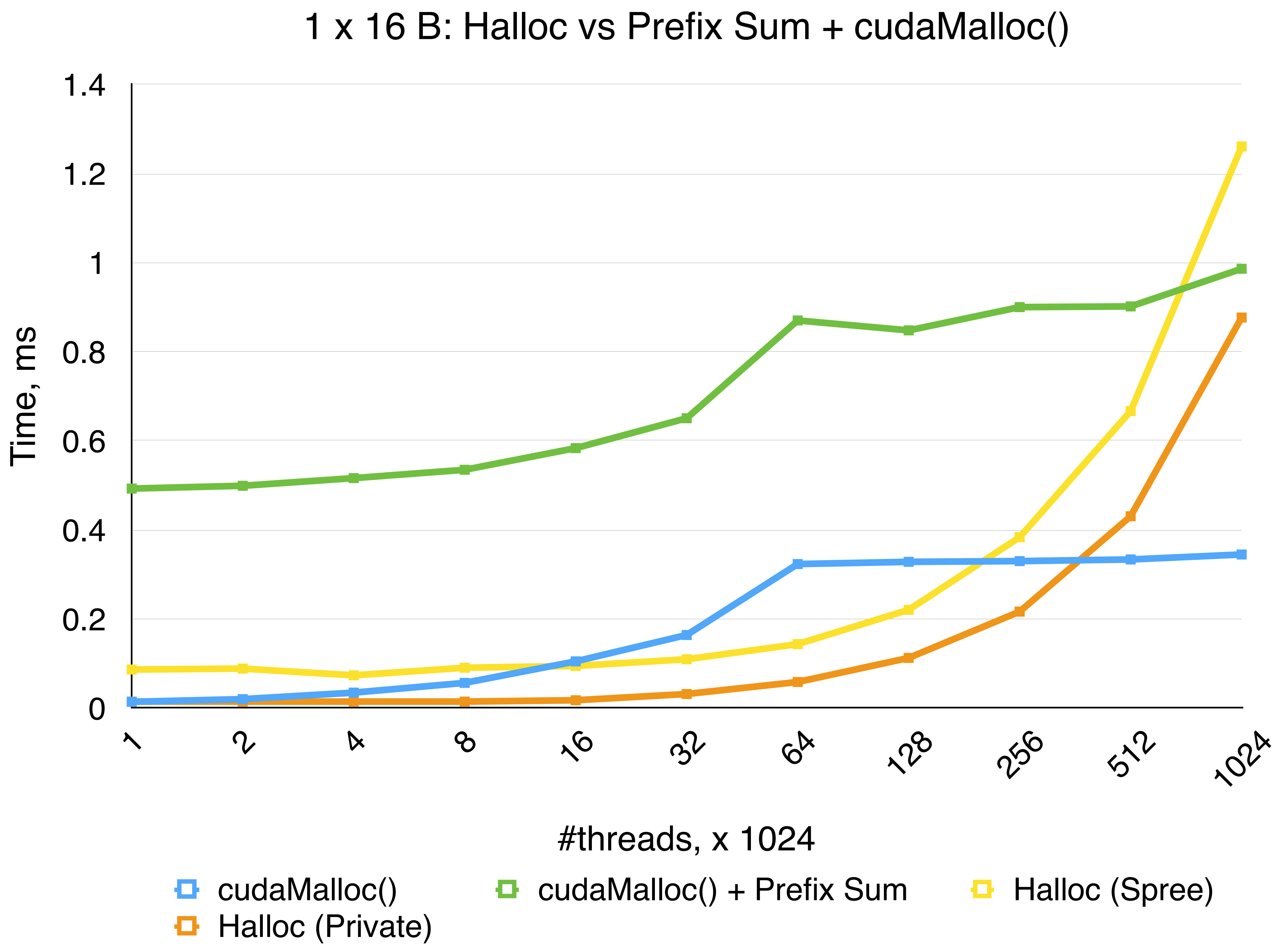
M. Steinberger et al. ScatterAlloc: Massively parallel dynamic memory allocation for the GPU. InPar–2012





- ScatterAlloc "holds": Halloc is 3–4x faster
- ScatterAlloc "doesn't hold": Halloc is 10–1000x faster

Halloc vs Prefix Sum + cudaMalloc()



- Performance depends on:
  - allocation size
  - #allocations
- Productivity:
  - halloc is better
  - no need to split kernel

Using in Your Applications

github.com/canonizer/halloc

- include

```
#include <halloc.h>
```

- initialize (host)

```
ha_init(512 * 1024 * 1024);  // 512 MiB memory
```

- use (GPU)

```
list_t *p = (list_t *)hamalloc(sizeof(list_t));
// … do something …
hafree(p);
```

- compile & link

```
nvcc -arch=sm_35 -O3 -I $(PREFIX)/include -dc myprog.cu -o myprog.o
nvcc -arch=sm_35 -O3 -L $(PREFIX)/lib -lhalloc -o myprog myprog.o
```

Conclusions & Future Work

- Fast and scalable GPU allocator
  - malloc/free-style, no additional limitations
  - 1.7 Gmalloc/s on K20X
- Future work
  - get slabs from CUDA allocator
  - decrease fragmentation
- **Looking for collaborations with application developers!**

Questions?

?

Halloc: github.com/canonizer/halloc

NVidia Application Lab at FZJ: www.fz-juelich.de/ias/jsc/nvlab

twitter: @adinetz

Andrew V. Adinetz adinetz@gmail.com
Dirk Pleiter d.pleiter@fz-juelich.de

Applications wanted!

Explicit Load to L1 on Kepler K40

```
__device__ inline uint ldca(const uint *p) {
  uint res;
  asm("ld.global.ca.u32 %0, [%1];": "=r"(res) : "l"(p));
  return res;
}
```

- Faster than texture or constant cache
  - for Halloc's readonly data
- The entire 128 B line is cached
  - should not be accessed otherwise