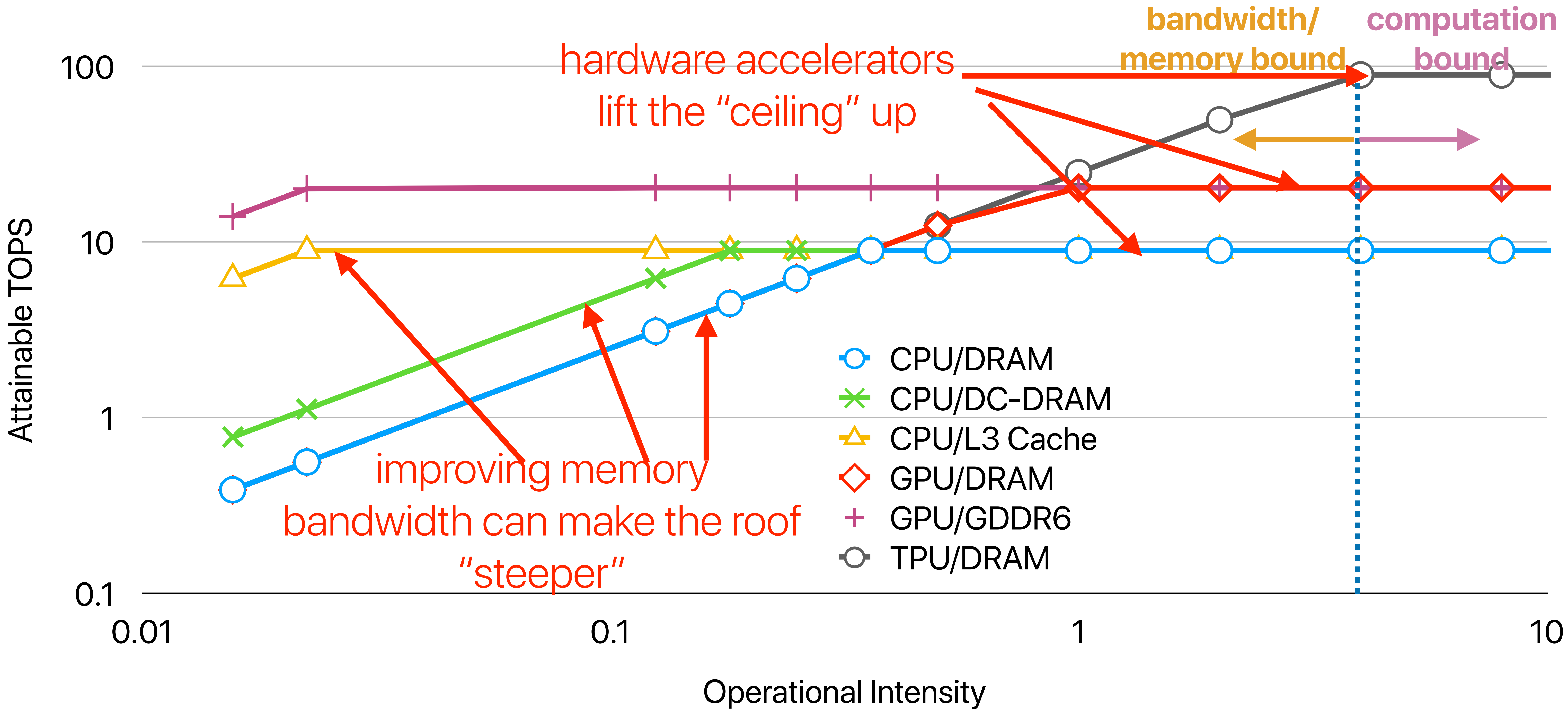
Modern Heterogeneous Computers: (5) Memory Components

Hung-Wei Tseng

Recap: the roofline after using hardware accelerators



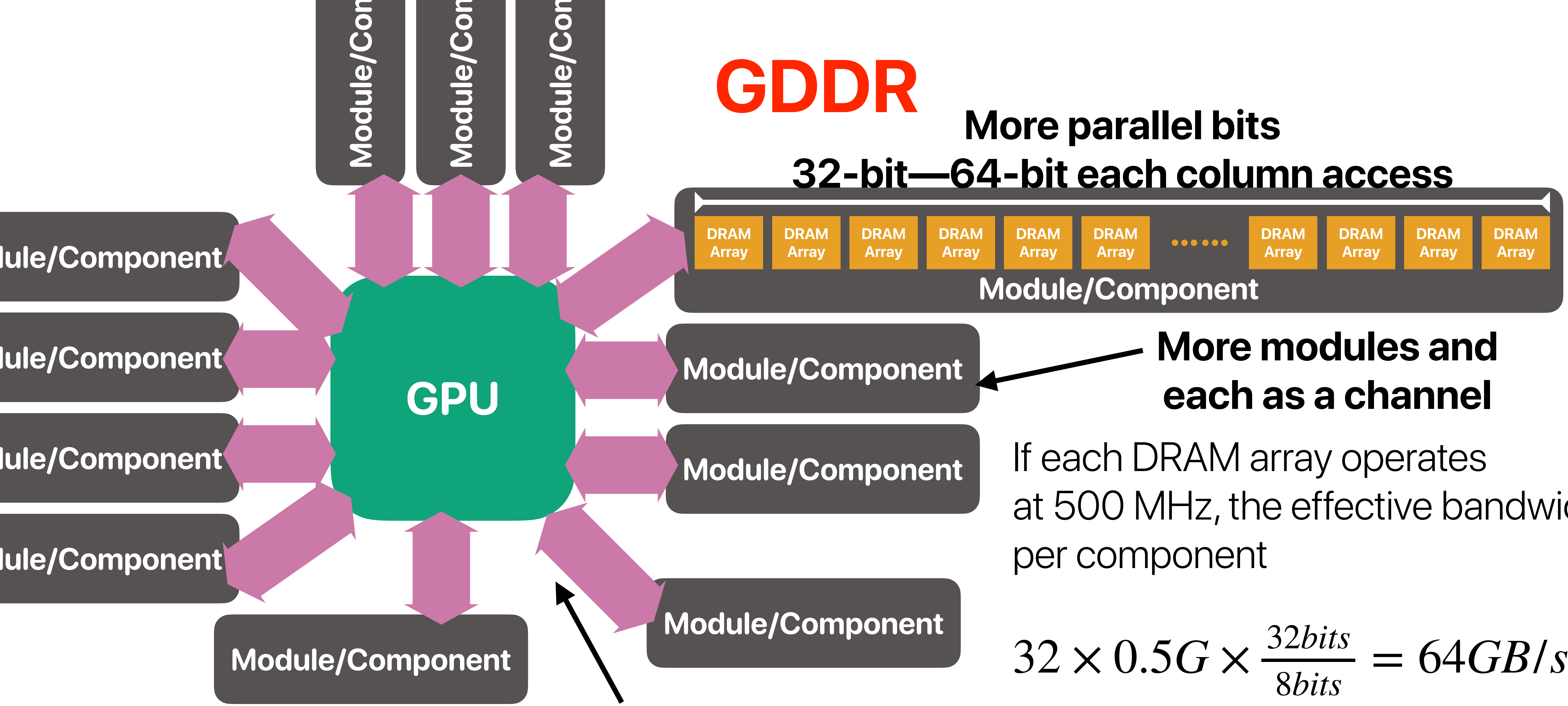
hardware accelerators lift the "ceiling" up

improving memory bandwidth can make the roof "steeper"

Ideas of increasing bandwidth

- More parallel bits
- More banks
- More channels
- Widen the memory-processor bus

GDDR



More parallel bits

32-bit—64-bit each column access

More modules and each as a channel

If each DRAM array operates at 500 MHz, the effective bandwidth per component

$$32 \times 0.5G \times \frac{32bits}{8bits} = 64GB/sec$$

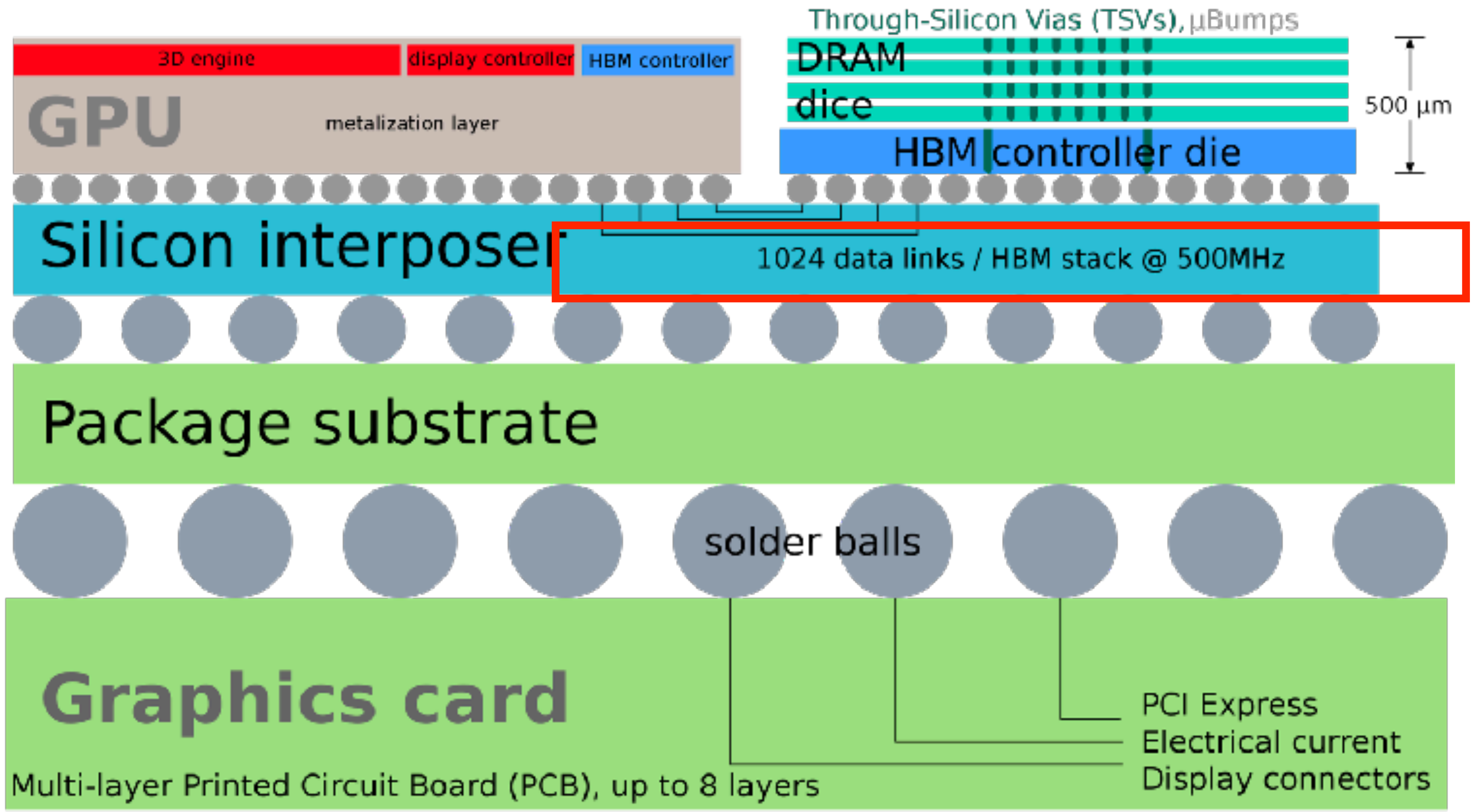
Wider bus: 192-bit — 384-bit bus

If the bus is 256-bit (32B) wide, the memory controller needs to be at

$$\frac{64GB/sec}{32B} = 2GHz$$

If you have 12x modules — 12*64 = 768 GB/sec

Recap: HBM (High Bandwidth Memory)

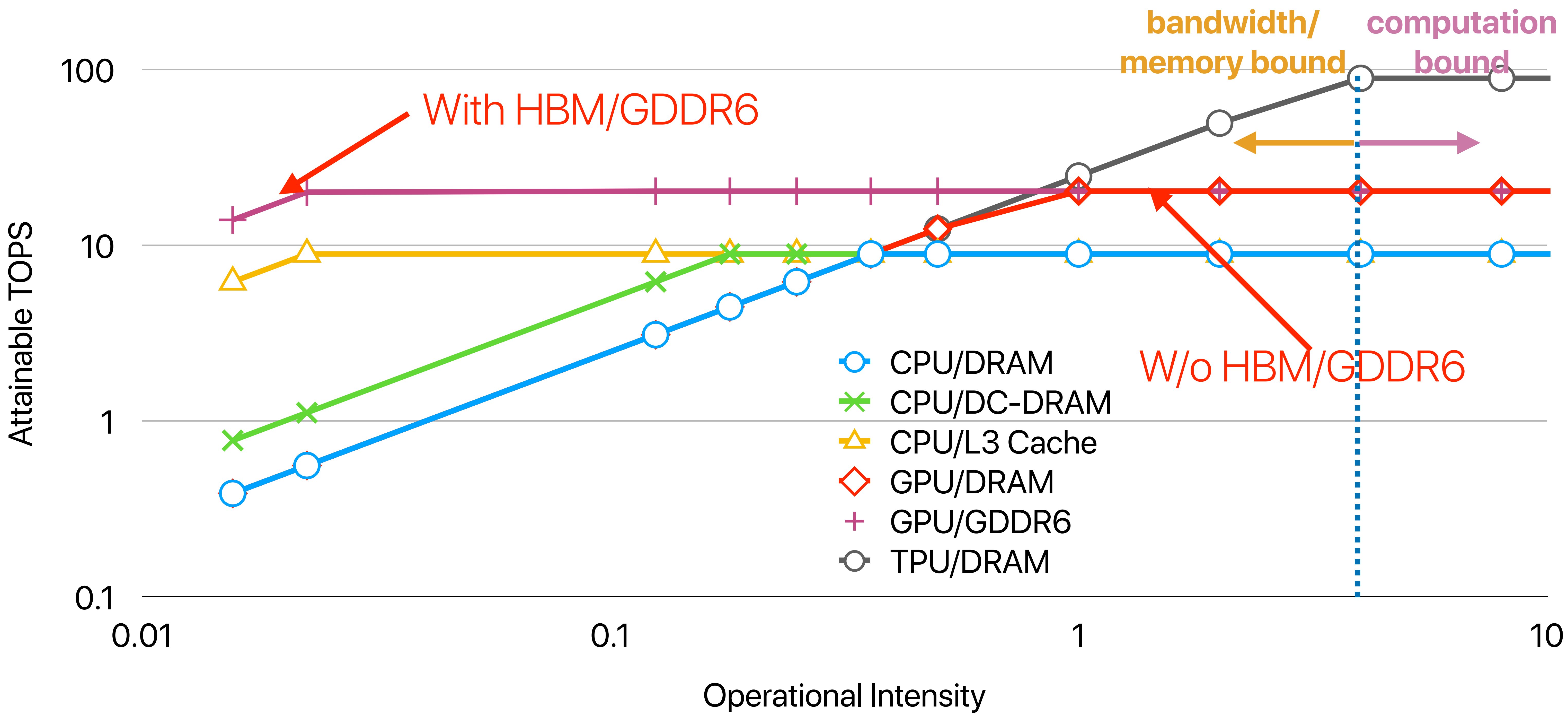


$$0.5G \times \frac{1024bits}{8bits} = 64GB/sec$$

If you have 4x modules — 4*64 = 256 GB/sec

HBM2 increase the clock rate to 2GHz — 1TB/sec

Recap: the roofline after using hardware accelerators



3:00

The program

```
#include <stdio.h>
#include <stdlib.h>
#include <sys/time.h>
#include <time.h>    /* for clock_gettime */
#include <malloc.h>
#include <immintrin.h>

void write_memory_avx(void* array, size_t size) {
  __m256i* varray = (__m256i*) array;

  __m256i vals = _mm256_set1_epi32(1);
  size_t i;
  for (i = 0; i < size / sizeof(__m256i); i++) {
    _mm256_store_si256(&varray[i], vals);
// This will generate the vmovaps instruction.
  }
}
```

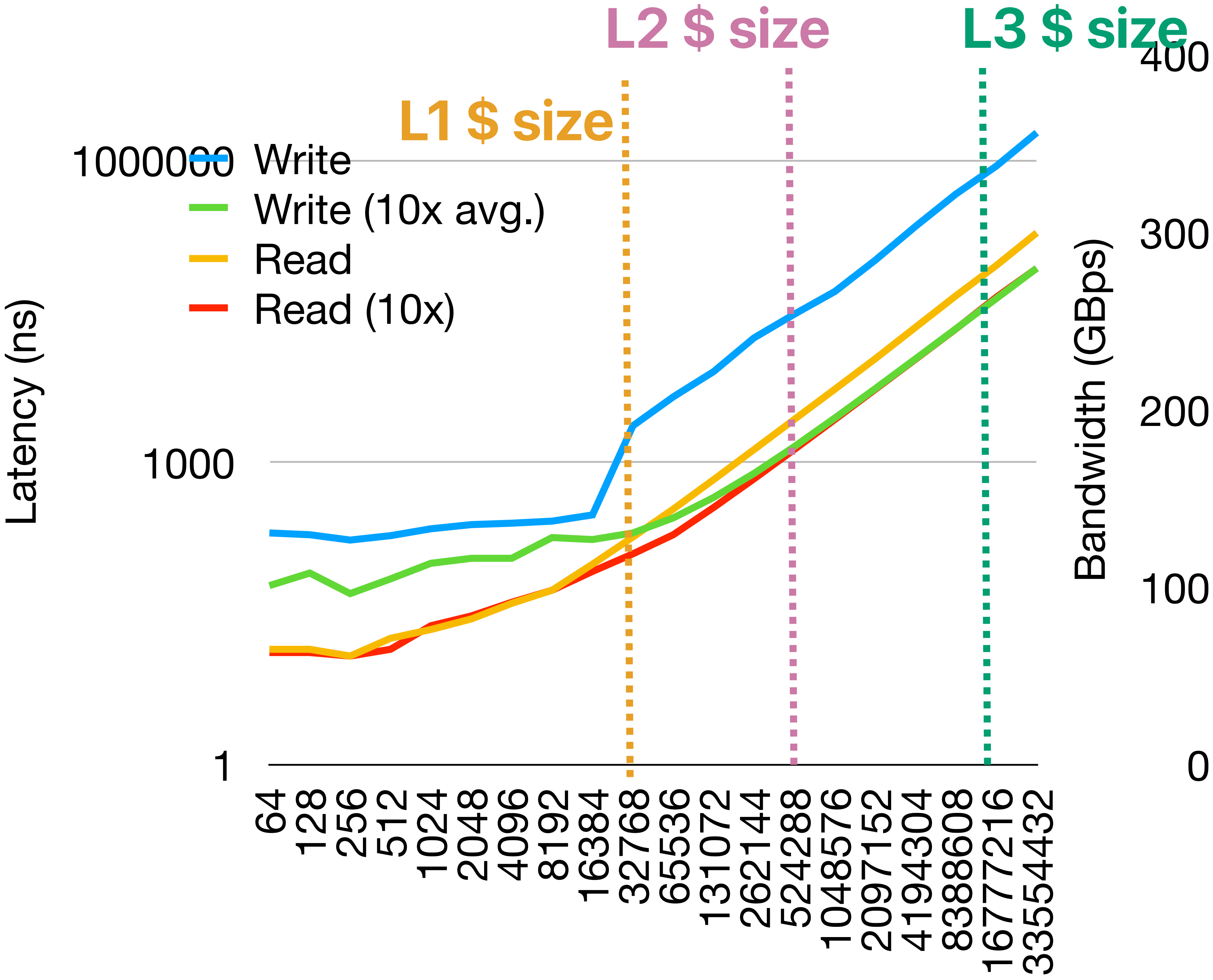
What’s the purpose of using AVX/Vector Instructions here?

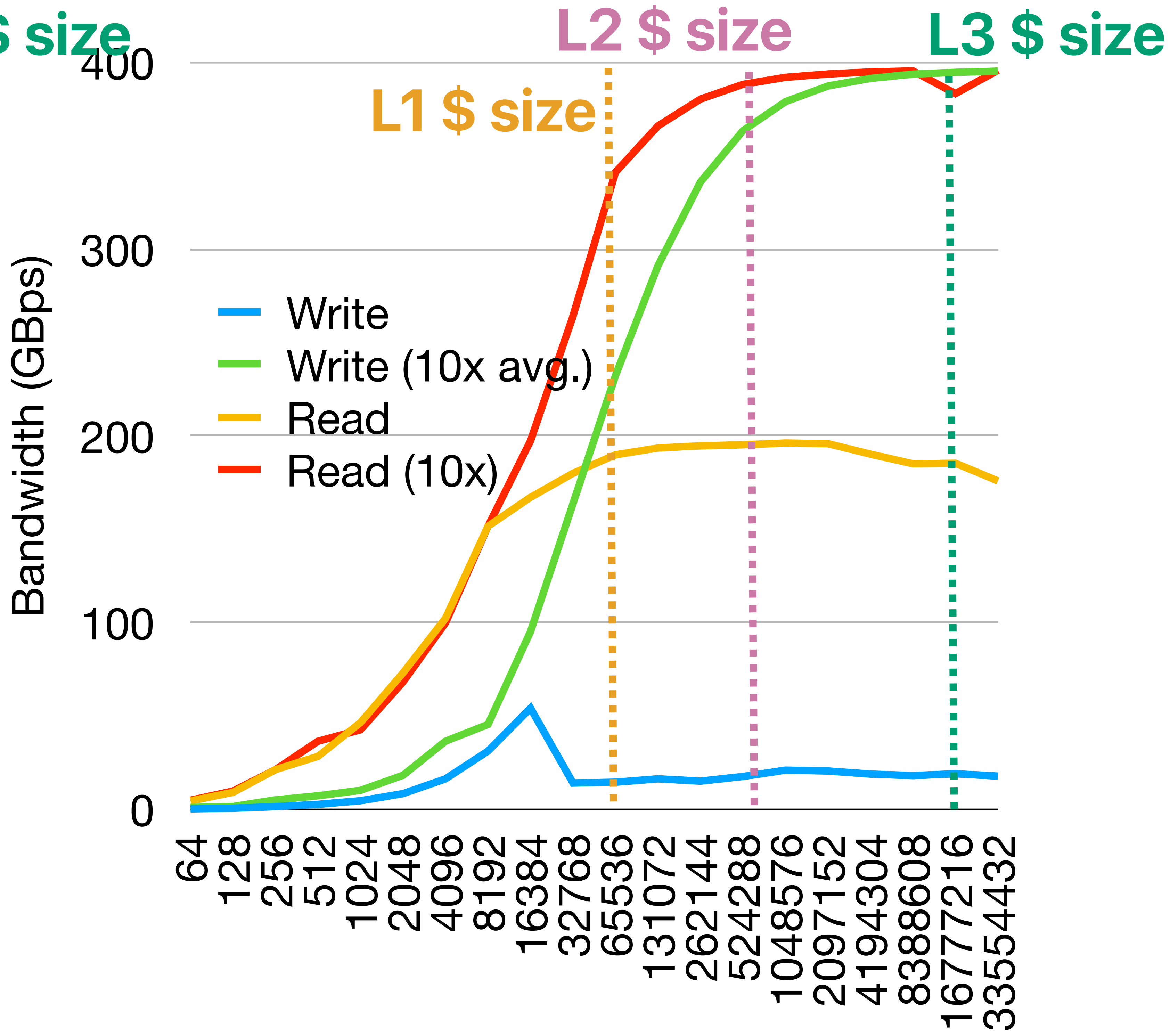
```
int main(int argc, char **argv)
{
    size_t *array;
    size_t size;
    double total_time;
    struct timespec start, end;
    int i;
    size = atoi(argv[1])/sizeof(size_t);
```

What’re the differences between the 1st and the rest runs?

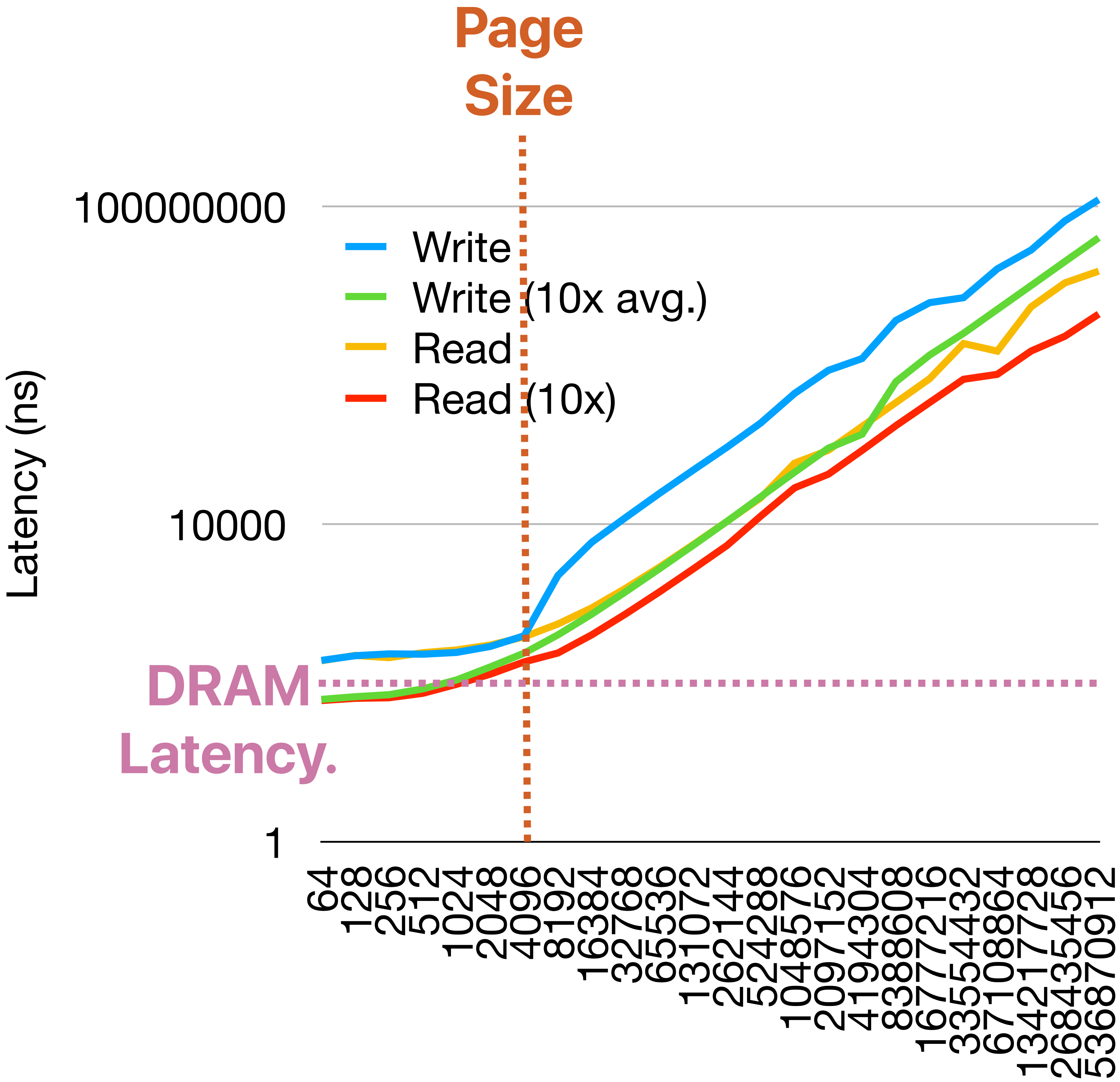
```
    array = (size_t *)memalign(32,sizeof(size_t)*size);

    clock_gettime(CLOCK_MONOTONIC, &start);

    write_memory_avx(array, size);

    clock_gettime(CLOCK_MONOTONIC, &end);

    total_time = ((end.tv_sec * 1000000000.0 +
end.tv_nsec) - (start.tv_sec * 1000000000.0 +
start.tv_nsec));
    fprintf(stderr,"Latency: %.0lf ns, GBps:
%lf\n",total_time, (double)((double)size*sizeof(size_t)/
(total_time)));
    clock_gettime(CLOCK_MONOTONIC, &start);

for(i = 0 ; i< 10; i++)
    write_memory_avx(array, size);

    clock_gettime(CLOCK_MONOTONIC, &end);
    total_time = ((end.tv_sec * 1000000000.0 +
end.tv_nsec) - (start.tv_sec * 1000000000.0 +
start.tv_nsec));
    fprintf(stderr,"Latency (10x average): %.0lf ns, GBps
(10x average): %lf\n",total_time/10, (double)
((double)size*10*sizeof(size_t)/(total_time)));
    return 0;
}
```

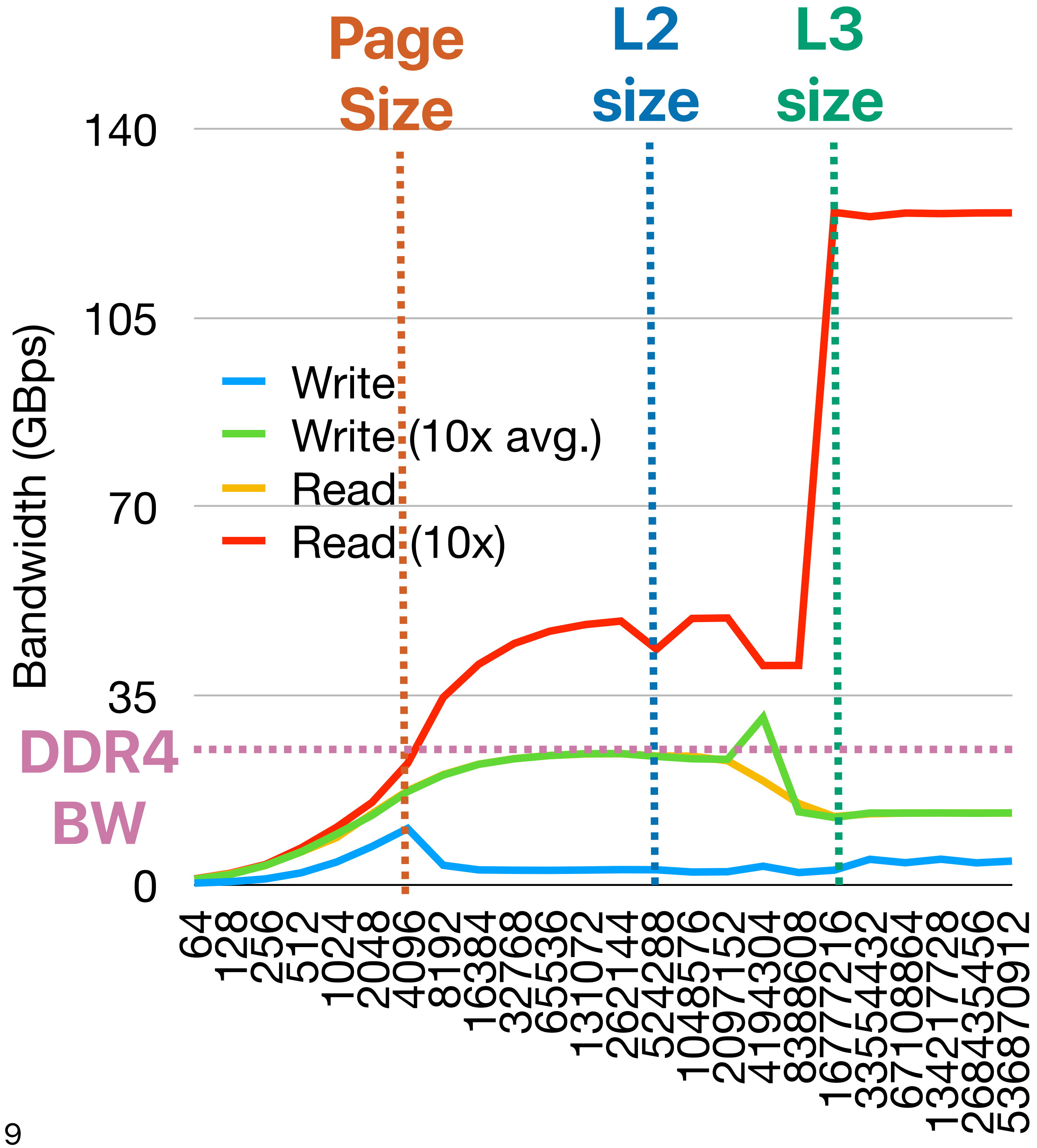
Wrong Performance Chart (on AMD RyZen 5 2600)



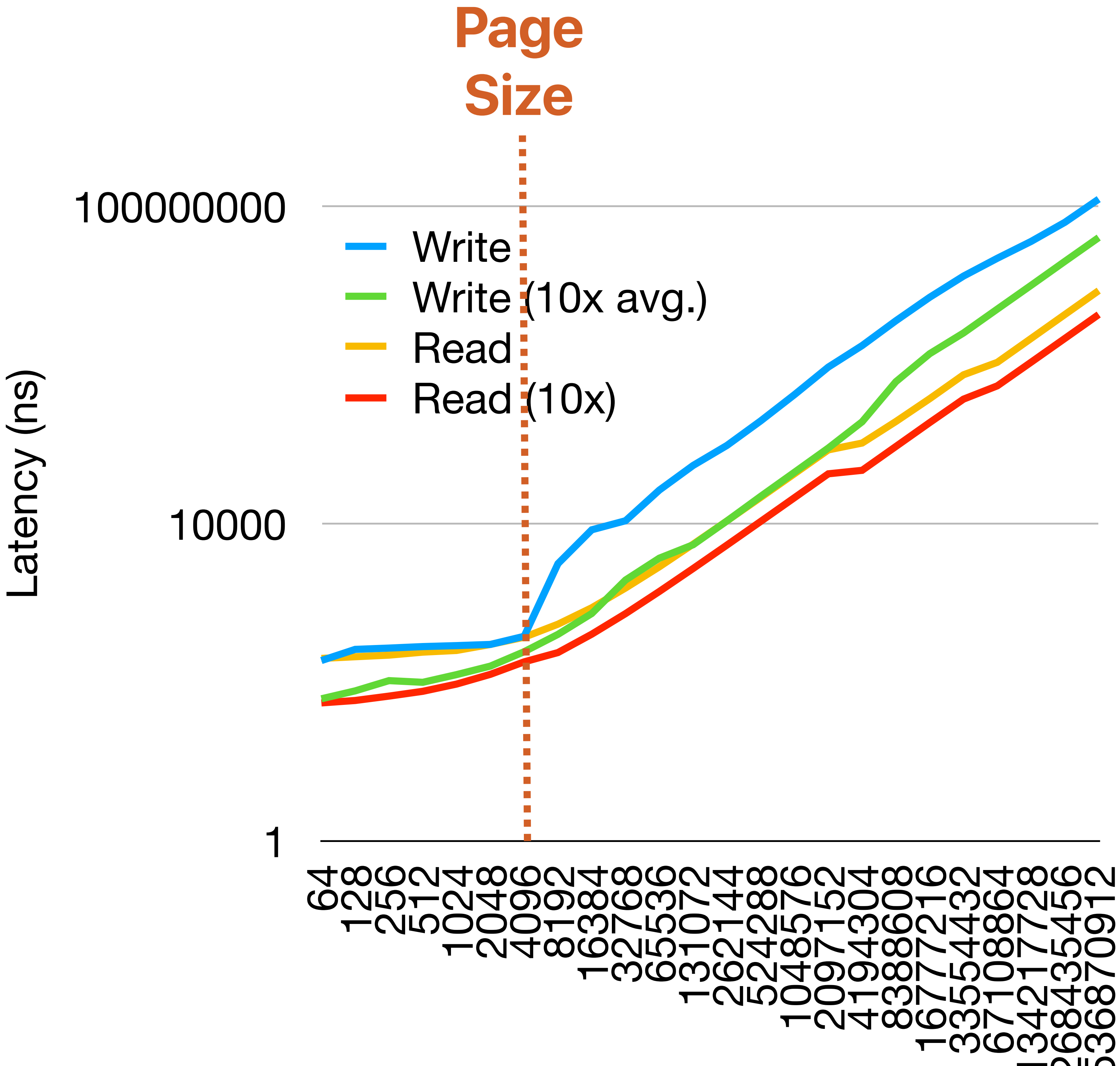


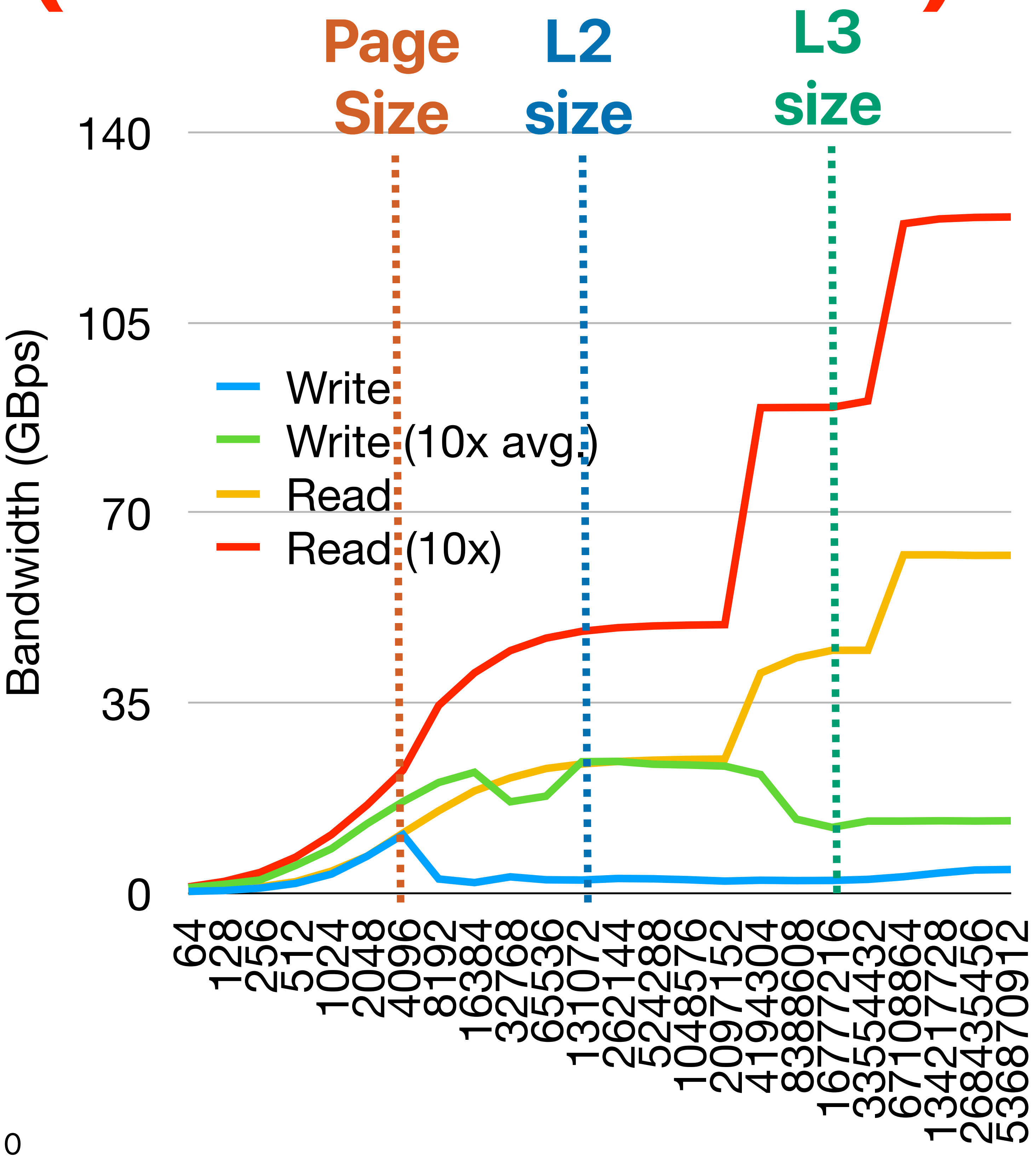
Corrected Performance Chart (on AMD RyZen 5 2600)



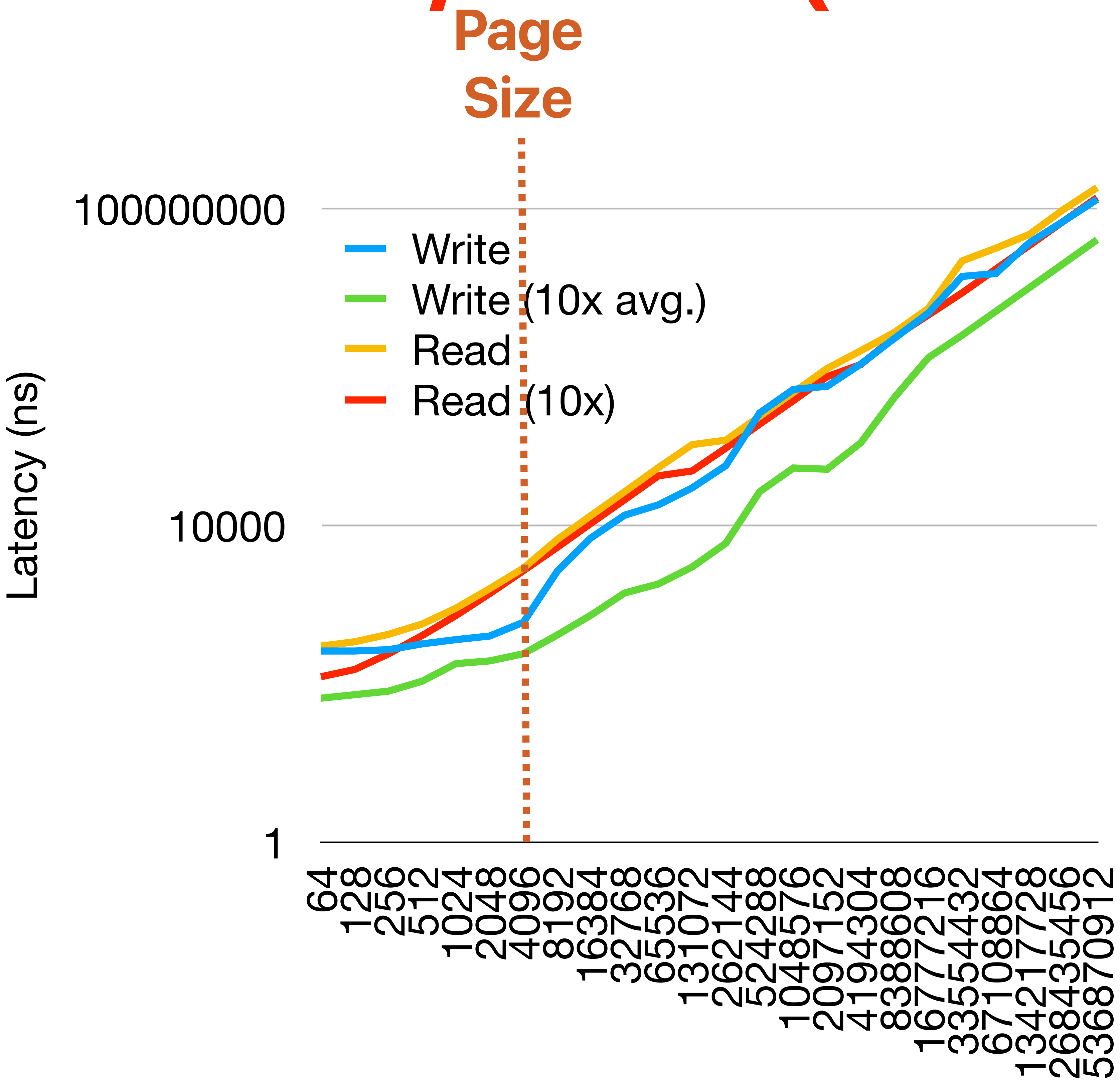


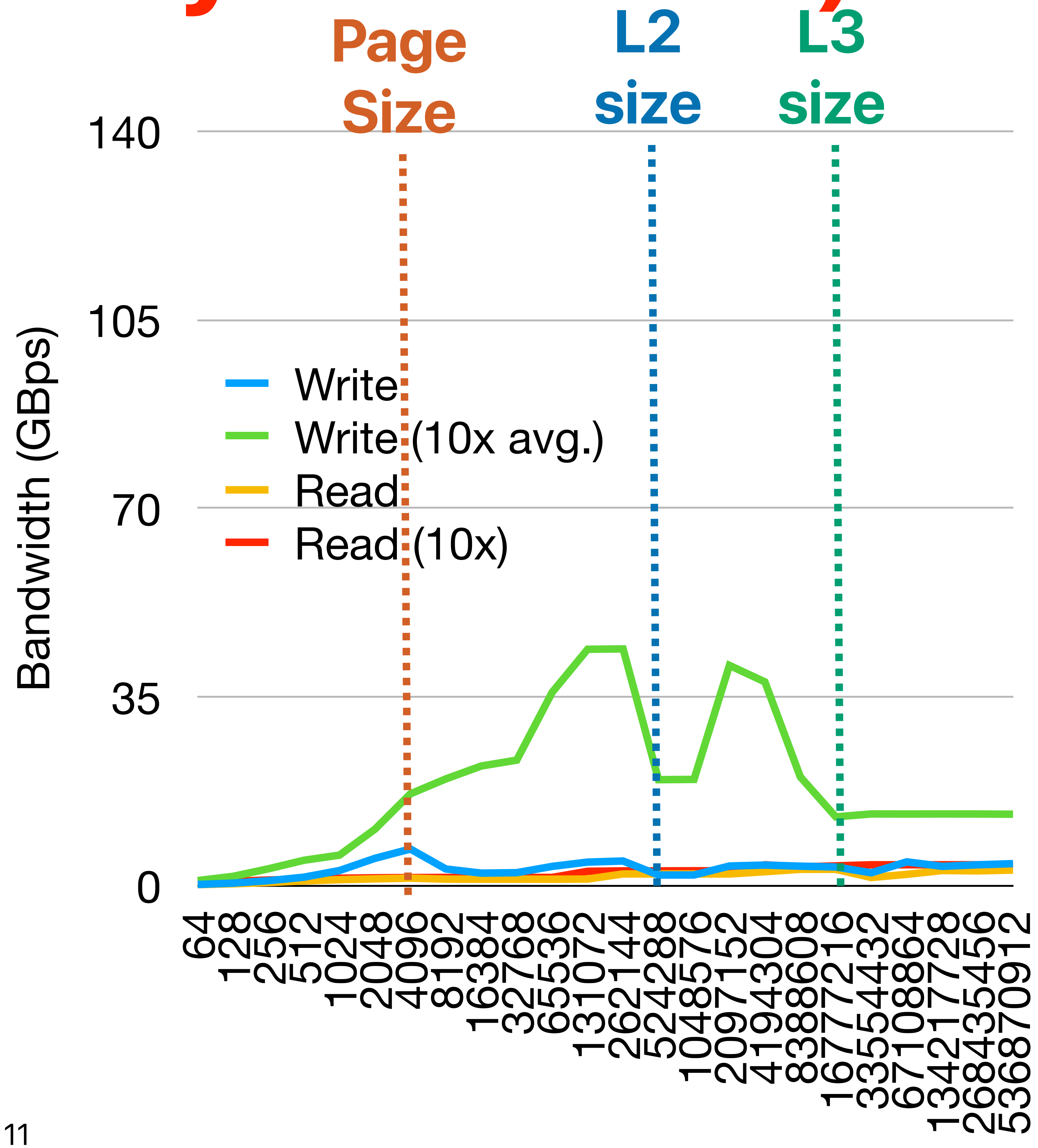
Performance Chart (on Core i5 12500)





W/O AVX (on AMD RyZen 5 2600)





3:00

What are the implications for programmers from the experimental results?

Implications

- The cost of load a word is a lot
  - More than just a load — you need to calculate the effective address
  - That’s why we want AVX to load 256-bit (32B or 4 64-bit words) to load in one instruction
- The cost of a page fault is significant
  - That’s why we see the first write/read is a lot longer
  - Huge page can be helpful

Case: what do you expect the performance of the program looks like and how can you improve it?

3:00

```c
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <sys/time.h>
#include <time.h> /* for clock_gettime */
#include <malloc.h>

void write_memory_loop(void* array, size_t size) {
  size_t* carray = (size_t*) array;
  size_t i;
  for (i = 0; i < size / sizeof(size_t); i++) {
    carray[i] = 1;
  }
}


int main(int argc, char **argv)
{
    size_t *array;
    size_t *dest;
    size_t size;
    double total_time;
```

```c
    struct timespec start, end;
    struct timeval time_start, time_end;
    size = atoi(argv[1])/sizeof(size_t);
    array = (size_t *)malloc(sizeof(size_t)*size);
    dest = (size_t *)malloc(sizeof(size_t)*size);
    clock_gettime(CLOCK_MONOTONIC, &start);

    write_memory_loop(array, size);

    clock_gettime(CLOCK_MONOTONIC, &start);

    memcpy(dest, array, size*sizeof(size_t));
    clock_gettime(CLOCK_MONOTONIC, &end);
   total_time = ((end.tv_sec * 1000000000.0 +
end.tv_nsec) - (start.tv_sec * 1000000000.0 +
start.tv_nsec));
    fprintf(stderr,"Latency: %.0lf ns, GBps: %lf,
%llu\n",total_time, (double)((double)size*sizeof(size_t)/
(total_time)), dest[rand()%size]);
    return 0;
}
```

https://github.com/hungweitseng/EE277/tree/main/demo/memory

The program

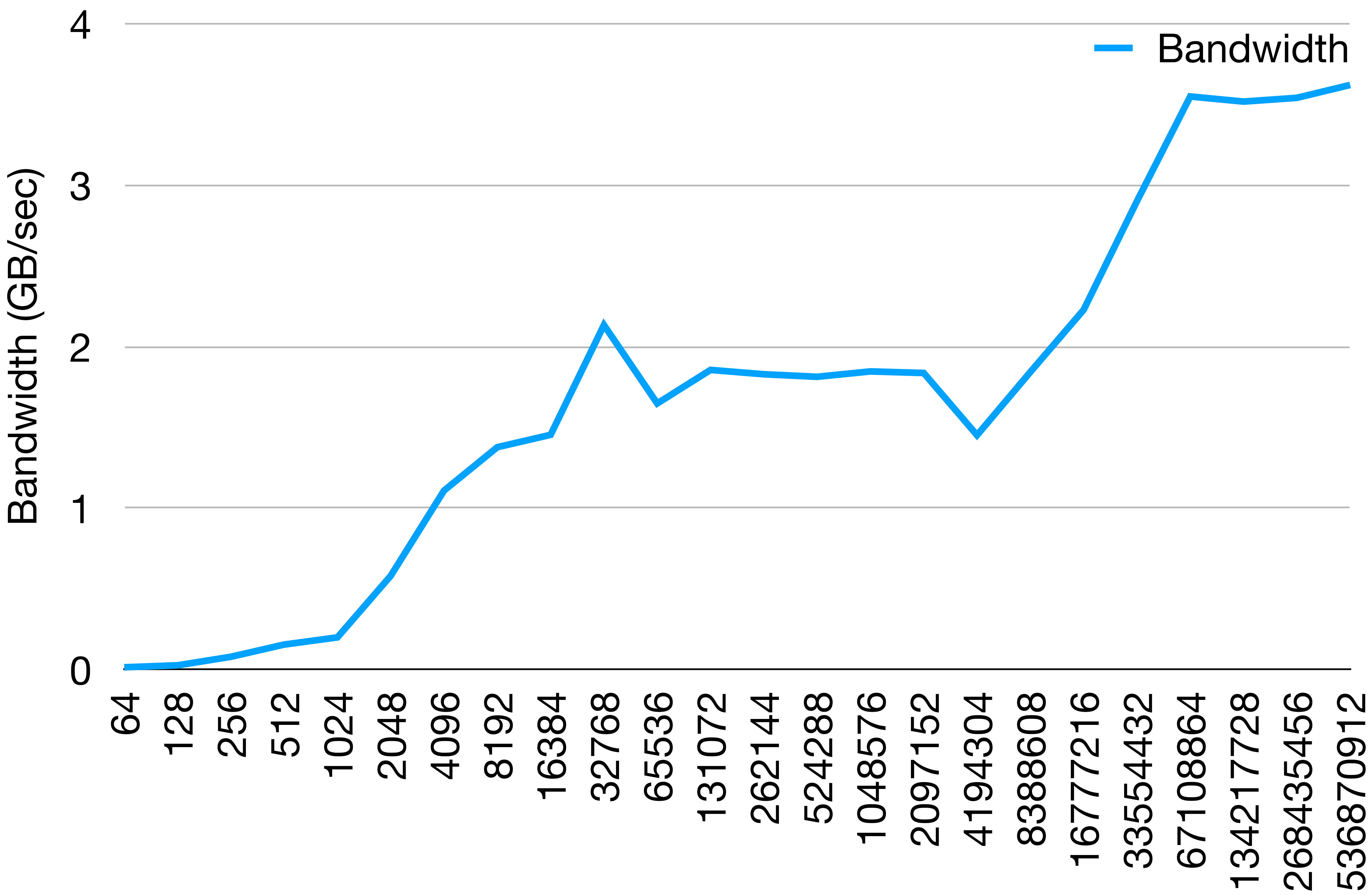
```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <sys/time.h>
#include <time.h>    /* for clock_gettime */
#include <malloc.h>

void write_memory_loop(void* array, size_t size) {
  size_t* carray = (size_t*) array;
  size_t i;
  for (i = 0; i < size / sizeof(size_t); i++) {
    carray[i] = 1;
  }
}


int main(int argc, char **argv)
{
    size_t *array;
    size_t *dest;
    size_t size;
    double total_time;
    struct timespec start, end;
    struct timeval time_start, time_end;
    size = atoi(argv[1])/sizeof(size_t);
    array = (size_t *)malloc(sizeof(size_t)*size);
```

```
    dest = (size_t *)malloc(sizeof(size_t)*size);
    clock_gettime(CLOCK_MONOTONIC, &start);    /* mark
start time */
    #ifdef SIMD
    write_memory_avx(array, size);
    #else
    write_memory_loop(array, size);
    #endif
    clock_gettime(CLOCK_MONOTONIC, &start);    /* mark
start time */
    memcpy(dest, array, size*sizeof(size_t));
    clock_gettime(CLOCK_MONOTONIC, &end);    /* mark
start time */
    total_time = ((end.tv_sec * 1000000000.0 +
end.tv_nsec) - (start.tv_sec * 1000000000.0 +
start.tv_nsec));
    fprintf(stderr,"Latency: %.0lf ns, GBps: %lf,
%llu\n",total_time, (double)((double)size*sizeof(size_t)/
(total_time)), dest[rand()%size]);
    return 0;
}
```

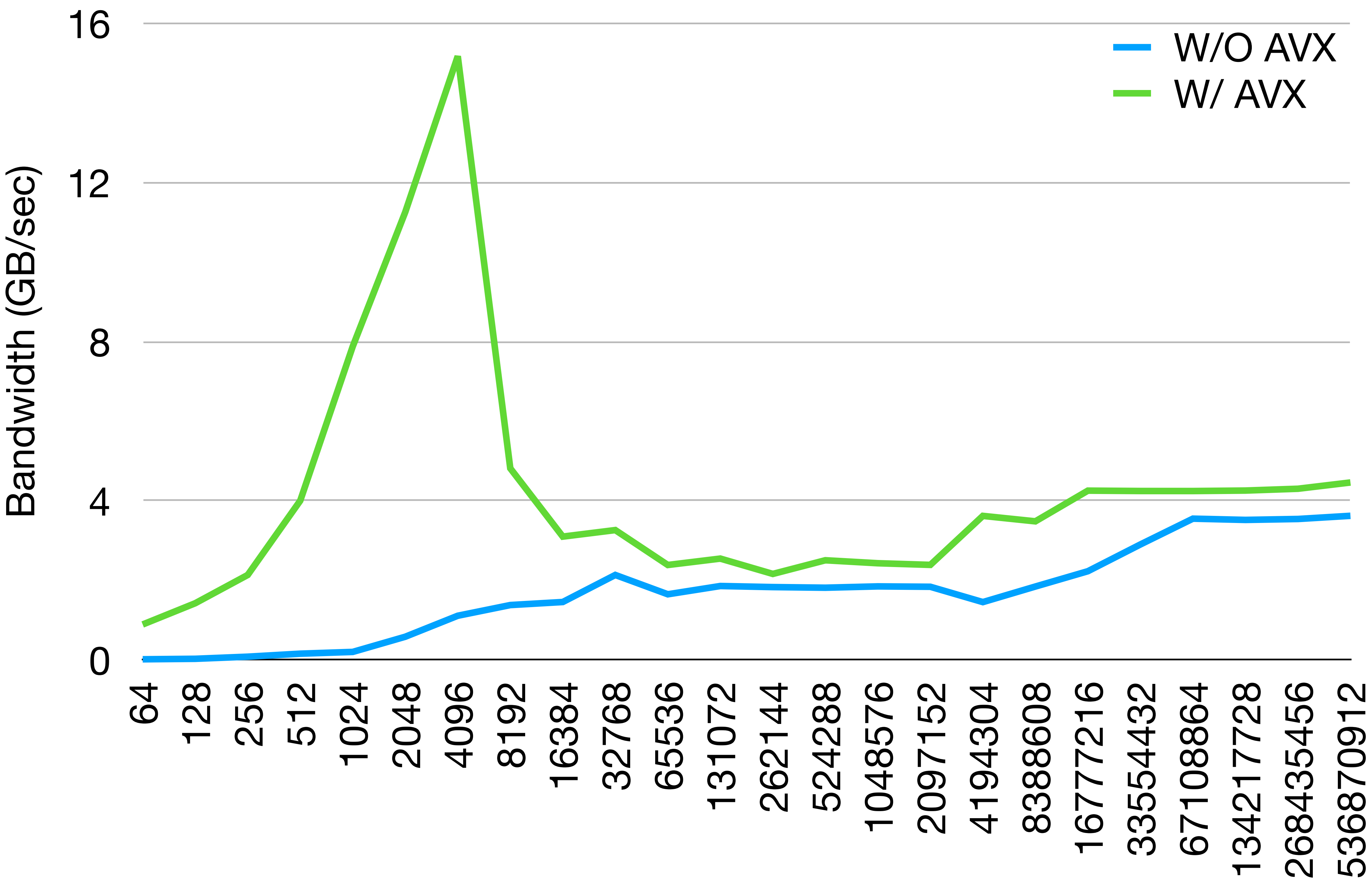
memmove & memcpy: 5% cycles in Google's datacenter

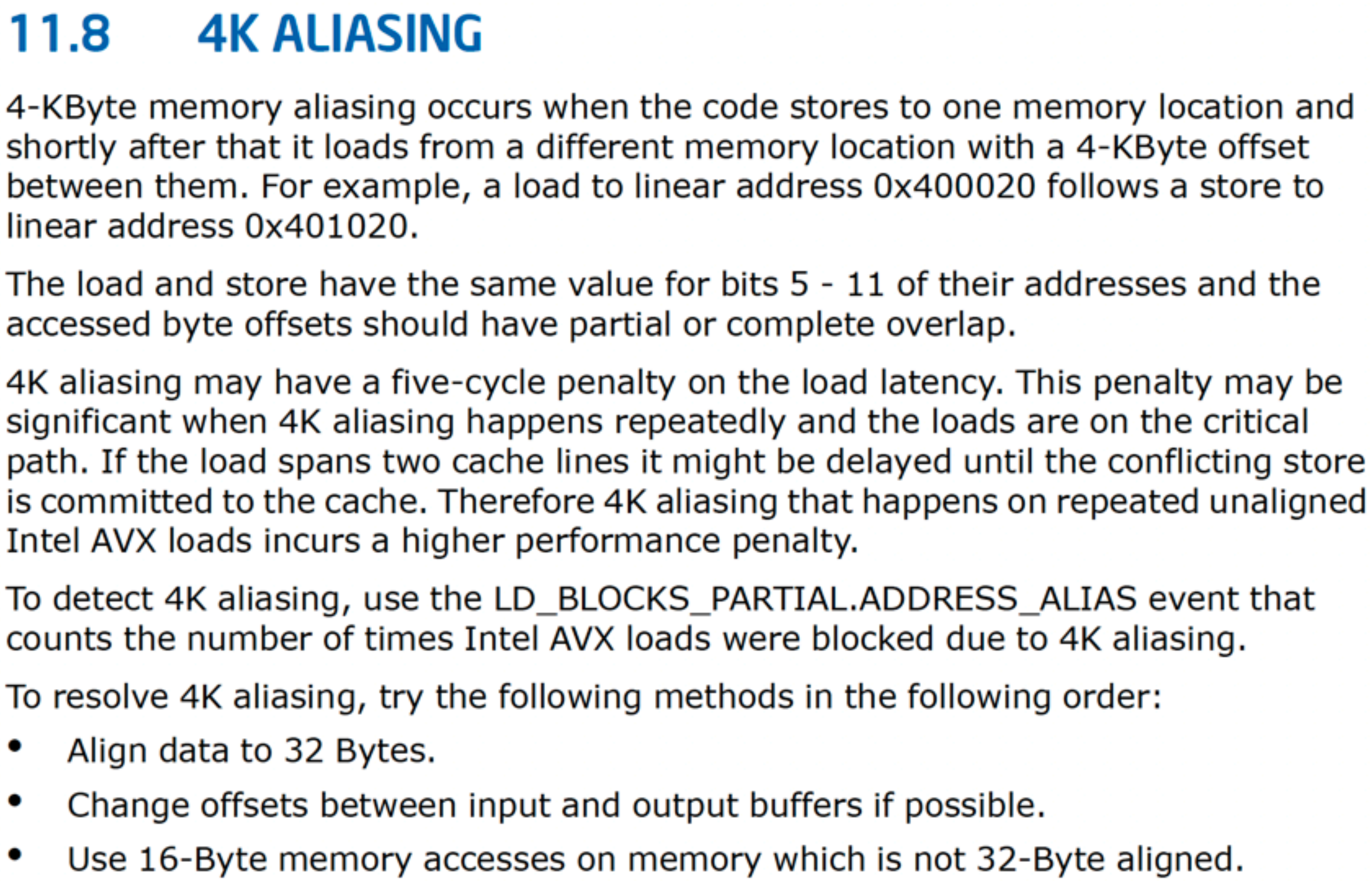
Svilen Kanev, Juan Pablo Darago, Kim Hazelwood, Parthasarathy Ranganathan, Tipp Moseley, Gu-Yeon Wei, and David Brooks. ISCA '15

Memory copy bandwidth



Memory copy bandwidth W/ AVX





11.8 4K ALIASING

4-KByte memory aliasing occurs when the code stores to one memory location and shortly after that it loads from a different memory location with a 4-KByte offset between them. For example, a load to linear address 0x400020 follows a store to linear address 0x401020.

The load and store have the same value for bits 5 - 11 of their addresses and the accessed byte offsets should have partial or complete overlap.

4K aliasing may have a five-cycle penalty on the load latency. This penalty may be significant when 4K aliasing happens repeatedly and the loads are on the critical path. If the load spans two cache lines it might be delayed until the conflicting store is committed to the cache. Therefore 4K aliasing that happens on repeated unaligned Intel AVX loads incurs a higher performance penalty.

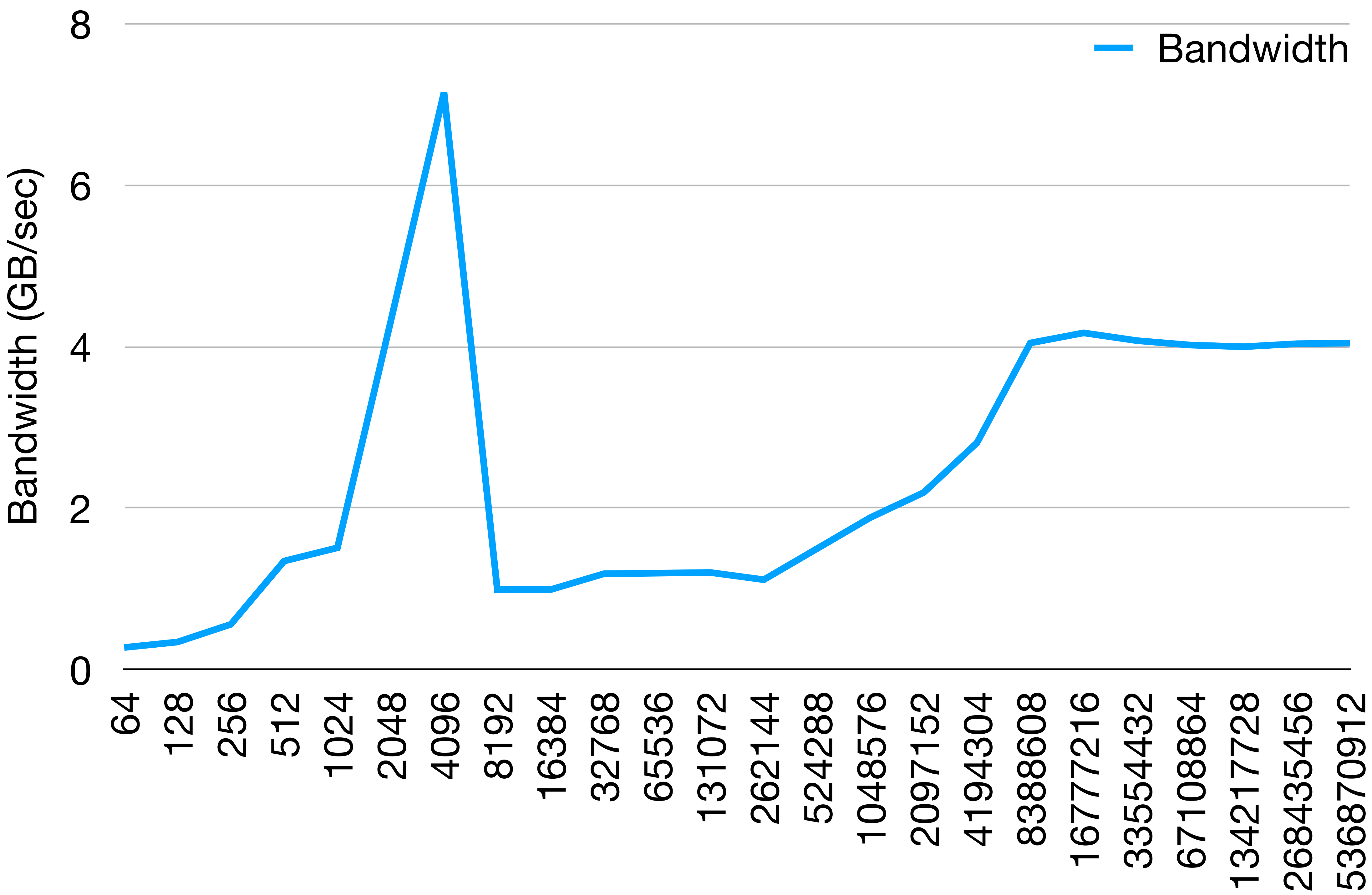
To detect 4K aliasing, use the LD_BLOCKS_PARTIAL.ADDRESS_ALIAS event that counts the number of times Intel AVX loads were blocked due to 4K aliasing.

To resolve 4K aliasing, try the following methods in the following order:

- Align data to 32 Bytes.
- Change offsets between input and output buffers if possible.
- Use 16-Byte memory accesses on memory which is not 32-Byte aligned.

https://www.intel.com/content/dam/doc/manual/64-ia-32-architectures-optimization-manual.pdf

Memory copy bandwidth W/ AVX on Core i5 12500

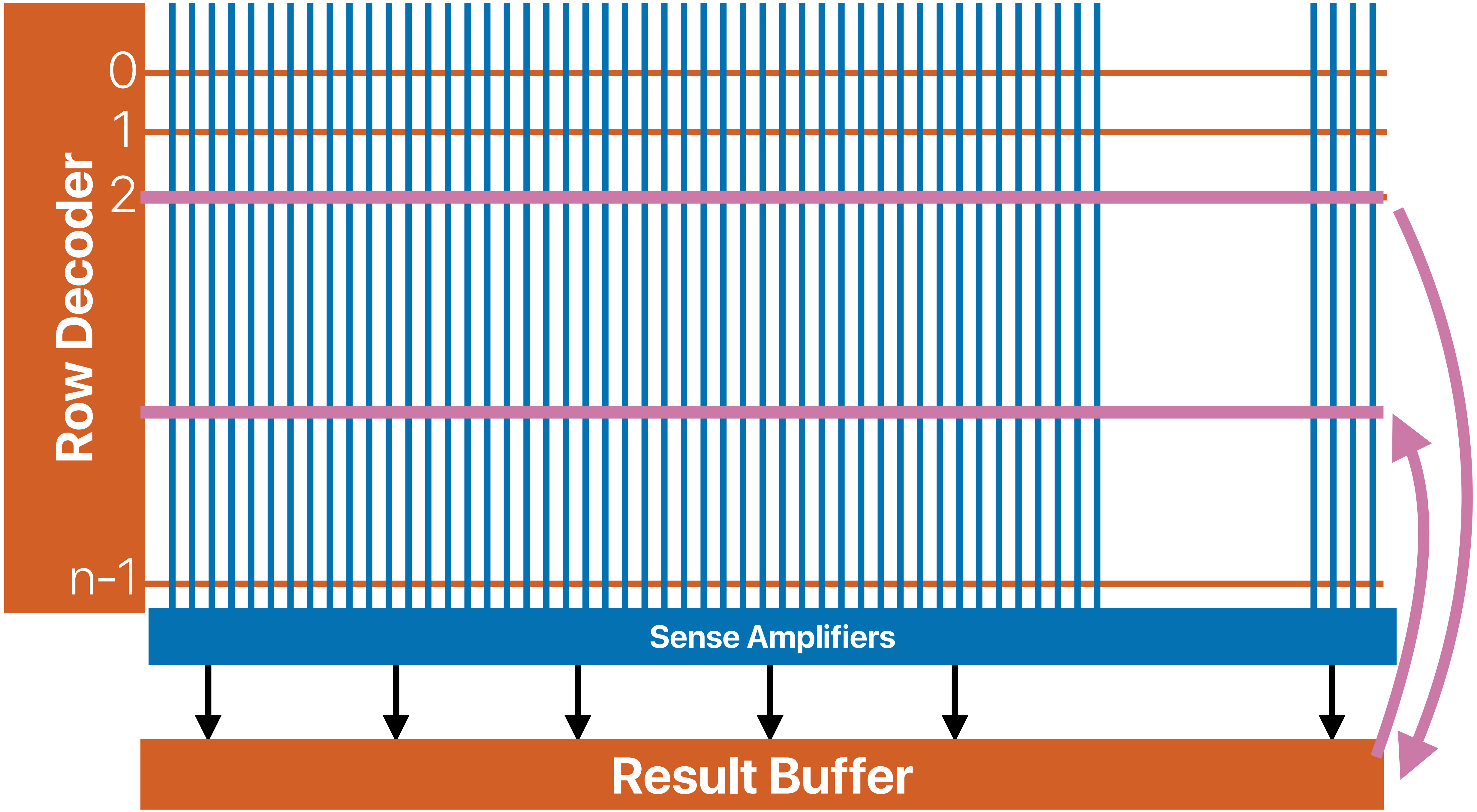


Implications

- The cost of load a word is a lot
  - More than just a load — you need to calculate the effective address
  - That's why we want AVX to load 256-bit (32B or 4 64-bit words) to load in one instruction
- The cost of a page fault is significant
  - That's why we see the first write/read is a lot longer
  - Huge page can be helpful
- Performance optimization in software is hard!!!

“In”-DRAM processing

Or we just clone/move?

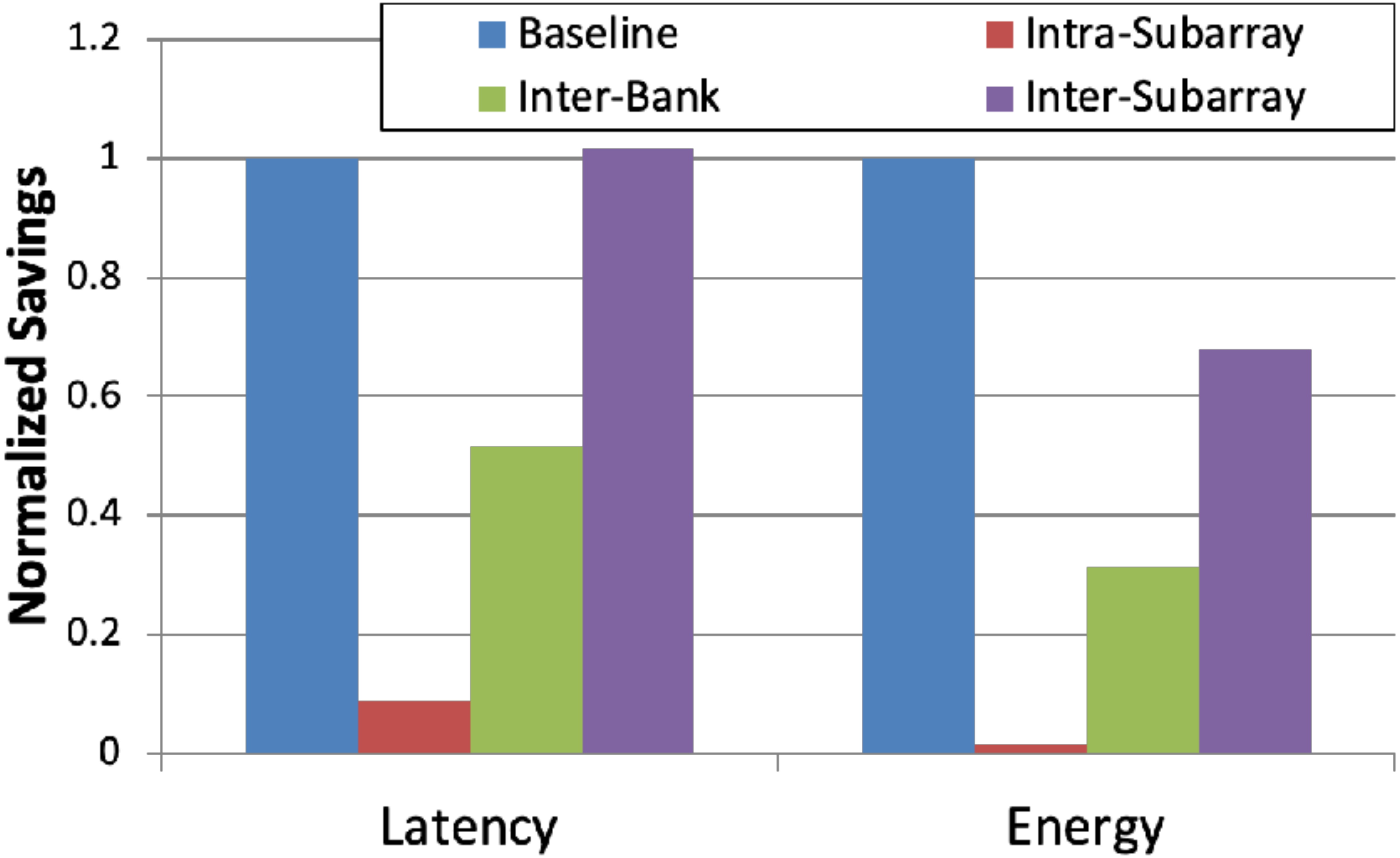


memmove & memcpy: 5% cycles in Google's datacenter

Svilen Kanev, Juan Pablo Darago, Kim Hazelwood, Parthasarathy Ranganathan, Tipp Moseley, Gu-Yeon Wei, and David Brooks.Profiling a warehouse-scale computer ISCA '15

Vivek Seshadri, Yoongu Kim, Chris Fallin, Donghyuk Lee, Rachata Ausavarungnirun, Gennady Pekhimenko, Yixin Luo, Onur Mutlu, Phillip B. Gibbons, Michael A. Kozuch, and Todd C. Mowry. RowClone: fast and energy-efficient in-DRAM bulk data copy and initialization. In MICRO-46.

The effect of RowClone



Limitations of in-DRAM processing

- Programming
- Data mapping
- Hardware design

3:00

How can we lower the data volume?

How can we lower data volume

- Compression

Demo

- Does reduce the overhead of data movement
- Computation overhead is a lot!
- https://github.com/lz4/lz4
- https://github.com/NVIDIA/nvcomp

Benchmarks

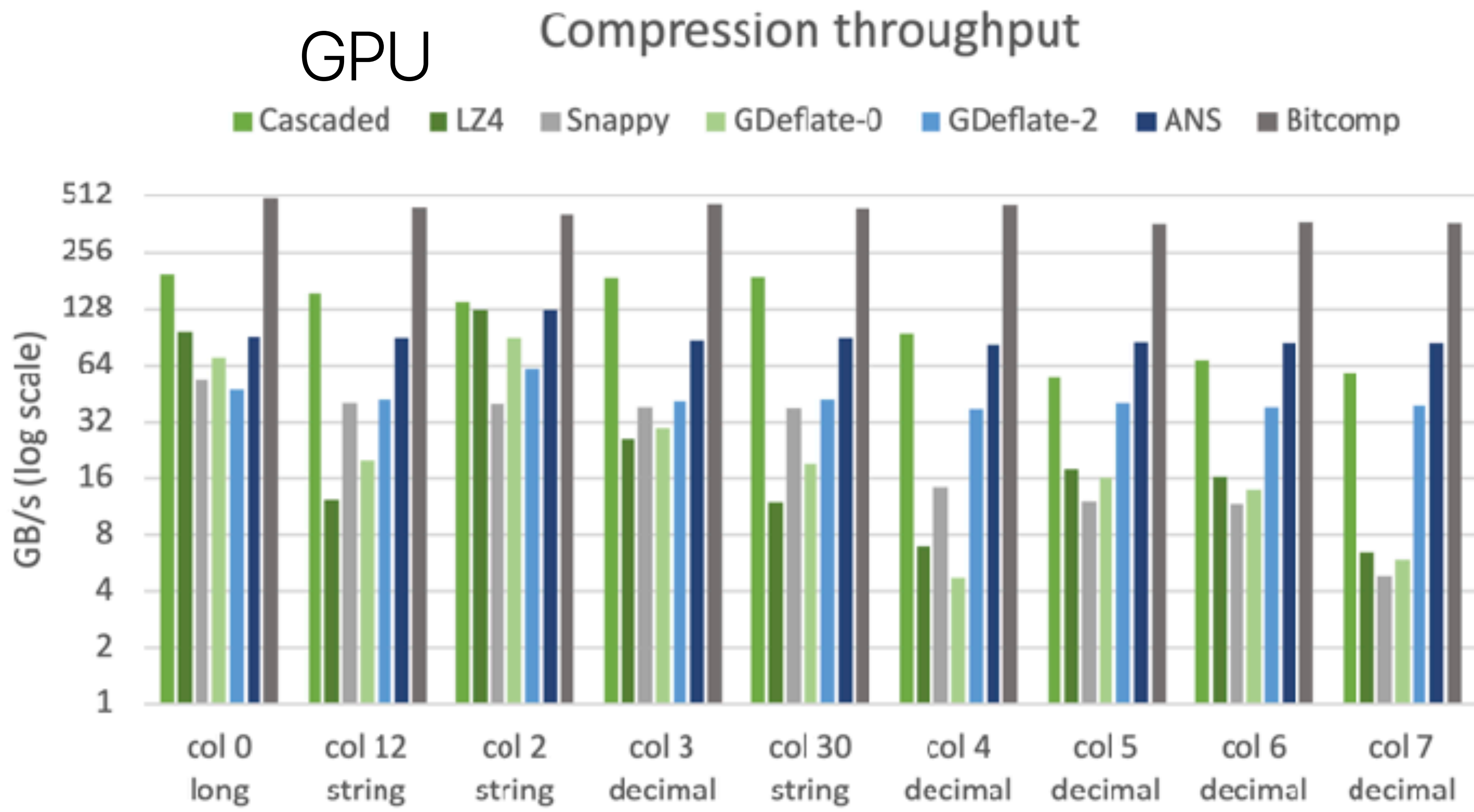
CPU

The benchmark uses lzbench, from @inikep compiled with GCC v8.2.0 on Linux 64-bits (Ubuntu 4.18.0-17). The reference system uses a Core i7-9700K CPU @ 4.9GHz (w/ turbo boost). Benchmark evaluates the compression of reference Silesia Corpus in single-thread mode.

Compressor	Ratio	Compression	Decompression
memcpy	1.000	13700 MB/s	13700 MB/s
LZ4 default (v1.9.0)	**2.101**	**780 MB/s**	**4970 MB/s**
LZO 2.09	2.108	670 MB/s	860 MB/s
QuickLZ 1.5.0	2.238	575 MB/s	780 MB/s
Snappy 1.1.4	2.091	565 MB/s	1950 MB/s
Zstandard 1.4.0 -1	2.883	515 MB/s	1380 MB/s
LZF v3.6	2.073	415 MB/s	910 MB/s
zlib deflate 1.2.11 -1	2.730	100 MB/s	415 MB/s
LZ4 HC -9 (v1.9.0)	**2.721**	41 MB/s	**4900 MB/s**
zlib deflate 1.2.11 -6	3.099	36 MB/s	445 MB/s

LZ4 is also compatible and optimized for x32 mode, for which it provides additional speed performance.

GPU



How can we lower data volume

- Compression
    - Too much computation overhead if no accelerator is presented
- Near/In-memory processing
    - Embed “logic”/“intelligence” near memory locations
    - Will talk more about this later!

Electrical
Computer Science
Engineering

277

つづく