Computer Architecture (263-2210-00L), Fall 2018

HW 5: Cache Partitioning, Memory Coherence, Memory Consistency, Multi-Core, GPU Programming

SOLUTIONS

Instructor: Prof. Onur Mutlu
TAs: Mohammed Alser, Can Firtina, Hasan Hassan, Jeremie Kim, Juan Gómez Luna, Geraldo Francisco de Oliveira, Minesh Patel, Giray Yaglikci

Assigned: Thursday, Dec 6, 2018
Due: **Friday, Dec 21, 2018**

- **Handin - Critical Paper Reviews (1).** You need to submit your reviews to `https://safari.ethz.ch/review/architecture18/`. Please, check your inbox, you should have received an email with the password you should use to login. If you didn't receive any email, contact comparch@lists.ethz.ch. In the first page after login, you should click in "Architecture - Fall 2018 Home", and then go to "any submitted paper" to see the list of papers.
- **Handin - Questions (2-8).** Please upload your solution to the Moodle (`https://moodle-app2.let.ethz.ch/`) as a single PDF file. **Please use a typesetting software (e.g., LaTeX) or a word processor (e.g., MS Word, LibreOfficeWriter) to generate your PDF file. Feel free to draw your diagrams either using an appropriate software or by hand, and include the diagrams into your solutions PDF.**

1 Critical Paper Reviews [200 points]

Please read the following handout on how to write critical reviews. We will give out extra credit that is worth 0.5% of your total grade for each good review.

- Lecture slides on guidelines for reviewing papers. Please, follow this format. `https://safari.ethz.ch/architecture/fall2018/lib/exe/fetch.php?media=onur-comparch-f18-how-to-do-the-paper-reviews.pdf`
- Some sample reviews can be found here: `https://safari.ethz.ch/architecture/fall2018/doku.php?id=readings`

(a) Write a one-page critical review for the first paper of the following list and at least **one** of the other 4 papers:

- M. Aater Suleman, Onur Mutlu, Moinuddin K. Qureshi, and Yale N. Patt, "Accelerating critical section execution with asymmetric multi-core architectures," ASPLOS 2009. `https://people.inf.ethz.ch/omutlu/pub/acs_asplos09.pdf`
- Jose A. Joao, M. Aater Suleman, Onur Mutlu, and Yale N. Patt, "Bottleneck Identification and Scheduling in Multithreaded Applications," ASPLOS 2012. `https://people.inf.ethz.ch/omutlu/pub/bottleneck-identification-and-scheduling_asplos12.pdf`
- G. Pekhimenko, V. Seshadri, O. Mutlu, P.B. Gibbons, M.A. Kozuch, T.C. Mowry, "Base-Delta-Immediate Compression: Practical Data Compression for On-Chip Caches," PACT 2012. `https://people.inf.ethz.ch/omutlu/pub/bdi-compression_pact12.pdf`
- V. Seshadri, O. Mutlu, M. A. Kozuch, and T. C. Mowry, "The evicted-address filter: A unified mechanism to address both cache pollution and thrashing," PACT 2012. `https://people.inf.ethz.ch/omutlu/pub/eaf-cache_pact12.pdf`
- N. Vijaykumar, G. Pekhimenko, A. Jog, A. Bhowmick, R. Ausavarungnirun, C. Das, M. Kandemir, T. C. Mowry, and O. Mutlu, "A Case for Core-Assisted Bottleneck Acceleration in GPUs: Enabling Flexible Data Compression with Assist Warps," ISCA 2015. `https://people.inf.ethz.ch/omutlu/pub/caba-gpu-assist-warps_isca15.pdf`

2 Utility-Based Cache Partitioning [100 points]

(a) Does utility-based cache partitioning guarantee a minimum amount of cache space to each thread/core sharing the cache? Why or why not? Explain.

Solution:
Yes, it is mentioned that at least one way is given to each core by the partitioning algorithm. If not, the thread running on that core could experience very large slowdowns due to the proposed partitioning algorithm.

(b) If yes, describe (and analyze) the minimum level of guarantee provided by utility based cache partitioning to each thread. If no, describe how the basic utility-based cache partitioning mechanism can be modified to provide a minimum amount of cache space to each thread.

Solution:
At least one way of the shared cache is guaranteed to each thread. The partitioning algorithm simply ignores all partitions that give 0 ways to a core and can never consider them 'optimal'.

(c) Describe how you would perform utility based cache partitioning if each core has an identical prefetcher that prefetches into the shared cache. What needs to be modified in the utility based cache partitioning mechanism described by Qureshi and Patt (MICRO 2006) to take into account prefetches? Explain the new hardware design.

Solution:
The main problem with prefetches is that it can affect the cache utility of each application, especially if a prefetcher is aggressive.
Multiple solutions exist. The easiest one is to ignore prefetches altogether (i.e., prefetches do not affect the number of ways given to each core, but may cause pollution), which likely leads to poor performance. Another solution is to treat prefetches as demand loads, in which case the prefetcher will affect the number of ways per core.
A significantly better way would be to estimate the accuracy of the prefetcher dynamically and treat accurate versus inaccurate prefetch requests differently. Prefetches requested by a prefetcher with very accuracy can be treated the same as demand loads, whereas prefetches issued by an inaccurate prefetcher can be ignored.

3 Cache Coherence [100 points]

We have a system with 4 byte-addressable processors. Each processor has a private 256-byte, direct-mapped, write-back L1 cache with a block size of 64 bytes. Coherence is maintained using the Illinois Protocol (MESI), which sends an invalidation to other processors on writes, and the other processors invalidate the block in their caches if *the block is present* (NOTE: On a write hit in one cache, a cache block in Shared state becomes Modified in that cache).

Accessible memory addresses range from 0x50000000 − 0x5FFFFFFF. Assume that the offset within a cache block is 0 for all memory requests. We use a snoopy protocol with a shared bus.

Cosmic rays strike the MESI state storage in your coherence modules, causing the state of a *single* cache line to instantaneously change to another state. This change causes an inconsistent state in the system. We show below the initial tag store state of the four caches, *after* the inconsistent state is induced.

Inital State

Cache 0		
	Tag	*MESI state*
Set 0	0x5FFFFF	M
Set 1	0x5FFFFF	E
Set 2	0x5FFFFF	S
Set 3	0x5FFFFF	I

Cache 1		
	Tag	*MESI state*
Set 0	0x522222	I
Set 1	0x510000	S
Set 2	0x5FFFFF	S
Set 3	0x533333	S

Cache 2		
	Tag	*MESI state*
Set 0	0x5F111F	M
Set 1	0x511100	E
Set 2	0x5FFFFF	S
Set 3	0x533333	S

Cache 3		
	Tag	*MESI state*
Set 0	0x5FF000	E
Set 1	0x511100	S
Set 2	0x5FFFF0	I
Set 3	0x533333	I

(a) What is the inconsistency in the above initial state? Explain with reasoning.

Cache 2, Set 1 should be in S state. Or Cache 3, Set 1 should be in I state.

Explanation. If the MESI protocol performs correctly, it is *not* possible for the same cache line to be in S and E states in different caches.

(b) Consider that, after the initial state, there are several paths that the program can follow that access different memory instructions. In b.1-b.4, we will examine whether the followed path can potentially lead to incorrect execution, i.e., an incorrect result.

b.1) Could the following path potentially lead to incorrect execution? Explain.

order	**Processor 0**	**Processor 1**	**Processor 2**	**Processor 3**
1^{st}			ld 0x51110040	
2^{nd}	st 0x5FFFFF40			
3^{rd}				st 0x51110040
4^{th}		ld 0x5FFFFF80		
5^{th}		ld 0x51110040		
6^{th}		ld 0x5FFFFF40		

No.

Explanation. The 3^{rd} instruction (st 0x51110040 in Processor 3) will invalidate the same line in Processor 2, and the whole system will be back to a consistent state (only one valid copy of 0x51110040 in the caches). Thus, the originally-inconsistent state does not affect the architectural state.

b.2) Could the following path potentially lead to incorrect execution? Explain.

order	**Processor 0**	**Processor 1**	**Processor 2**	**Processor 3**
1^{st}				ld 0x51110040
2^{nd}	ld 0x5FFFFF00			
3^{rd}			ld 0x51234540	
4^{th}	st 0x5FFFFF40			
5^{th}				ld 0x51234540
6^{th}	ld 0x5FFFFF00			

Yes.

Explanation. The 1^{st} instruction could read invalid data. This would be the case if the cosmic-ray-induced change was from I to S in Cache 3 for cache line 0x51110040.

After some time executing a particular path (which could be a path *different* from the paths in parts b.1-b.4) and with no further state changes caused by cosmic rays, we find that the final state of the caches is as follows.

Final State

Cache 0		
	Tag	*MESI state*
Set 0	0x5FFFFF	M
Set 1	0x5FFFFF	E
Set 2	0x5FFFFF	S
Set 3	0x5FFFFF	E

Cache 1		
	Tag	*MESI state*
Set 0	0x5FF000	I
Set 1	0x510000	S
Set 2	0x5FFFFF	S
Set 3	0x533333	I

Cache 2		
	Tag	*MESI state*
Set 0	0x5F111F	M
Set 1	0x511100	E
Set 2	0x5FFFFF	S
Set 3	0x533333	I

Cache 3		
	Tag	*MESI state*
Set 0	0x5FF000	M
Set 1	0x511100	S
Set 2	0x5FFFF0	I
Set 3	0x533333	I

(c) What is the *minimum* set of memory instructions that leads the system from the initial state to the final state? Indicate the set of instructions in order, and clearly specify the access type (ld/st), the address of each memory request, and the processor from which the request is generated.

The minimum set of instructions is:
(1) st 0x533333C0 // Processor 0
(2) ld 0x5FFFFFC0 // Processor 0
(3) ld 0x5FF00000 // Processor 1
(4) st 0x5FF00000 // Processor 3

Alternatively, as instructions (1)(2) and instructions (3)(4) touch different cache lines, we just need to keep the order between (1)(2), and between (3)(4). These are valid reorderings: (3)(4)(1)(2), (1)(3)(2)(4), (3)(1)(4)(2), (1)(3)(4)(2) or (3)(1)(2)(4).

Explanation.
(1) The instruction sets the line 0x533333C0 to M state in Cache 0, and invalidates the line 0x533333C0 in Cache 1 and Cache 2.
(2) The instruction evicts 0x533333C0 from Cache 0, and sets the line 0x5FFFFFC0 to E state in Cache 0.
(3) The instruction sets the line 0x5FF00000 to S state in Cache 1, as well as in Cache 3.
(4) The instruction sets the line 0x5FF00000 to M state in Cache 3, and it invalidates the line 0x5FF00000 in Cache 1.

4 Memory Consistency [100 points]

A programmer writes the following two C code segments. She wants to run them concurrently on a multicore processor, called SC, using two different threads, each of which will run on a different core. The processor implements *sequential consistency*, as we discussed in the lecture.

	Thread T0		Thread T1
Instr. T0.0	`a = X[0];`	Instr. T1.0	`Y[0] = 1;`
Instr. T0.1	`b = a + Y[0];`	Instr. T1.1	`*flag = 1;`
Instr. T0.2	`while(*flag == 0);`	Instr. T1.2	`X[1] *= 2;`
Instr. T0.3	`Y[0] += 1;`	Instr. T1.3	`a = 0;`

`X`, `Y`, and `flag` have been allocated in main memory, while `a` and `b` are contained in processor registers. A read or write to any of these variables generates a single memory request. The initial values of all memory locations and variables are 0. Assume each line of the C code segment of a thread is a *single* instruction.

(a) What is the final value of `Y[0]` in the SC processor, after both threads finish execution? Explain your answer.

2.

Explanation. `Y[0]` is set equal to 1 by instruction T1.0. Then, it will be incremented by instruction T0.3. The sequential consistency model ensures that the operations of each individual thread are executed in the order specified by its program. Across threads, the ordering is enforced by the use of `flag`. Thread 0 will remain in instruction T0.2 until `flag` is set by T1.1, i.e., after `Y[0]` is initialized. So, instruction T0.3 must be executed after instruction T1.0, causing `Y[0]` to be first set to 1 and then incremented.

(b) What is the final value of `b` in the SC processor, after both threads finish execution? Explain your answer.

0 or 1.

Explanation. There are *at least* two possible sequentially-consistent orderings that lead to *at most* two different values of `b` at the end:
Ordering 1: T1.0 $\rightarrow$ T0.1 - Final value = 1.
Ordering 2: T0.1 $\rightarrow$ T1.0 - Final value = 0.

With the aim of achieving higher performance, the programmer tests her code on a new multicore processor, called RC, that implements *weak consistency*. As discussed in the lecture, the weak consistency model has no need to guarantee a strict order of memory operations. For this question, consider a very weak model where there is *no* guarantee on the ordering of instructions as seen by different cores.

(c) What is the final value of `Y[0]` in the RC processor, after both threads finish execution? Explain your answer.

1 or 2.

Explanation. Since there is no guarantee of a strict order of memory operations, as seen by different cores, instruction T1.1 could complete before or after instruction T1.0, from the perspective of the core that executes T0. If instruction T1.1 completes before instruction T1.0, from the perspective of the core that executes T0, instruction T0.3 could complete before or after instruction T1.0. Thus, there are three possible weakly-consistent orderings that lead to different values of `Y[0]` at the end:
Ordering 1 (from the perspective of T0): T1.0 $\rightarrow$ T1.1 $\rightarrow$ T0.3 - Final value = 2.
Ordering 2 (from the perspective of T0): T1.1 $\rightarrow$ T1.0 $\rightarrow$ T0.3 - Final value = 2.
Ordering 3 (from the perspective of T0): T1.1 $\rightarrow$ T0.3 $\rightarrow$ T1.0 - Final value = 1.

After several months spent debugging her code, the programmer learns that the new processor includes a `memory_fence()` instruction in its ISA. The semantics of `memory_fence()` is as follows for a given thread that executes it:

1. Wait (stall the processor) until *all* preceding memory operations from the thread complete in the memory system and become visible to other cores.
2. Ensure *no* memory operation from any later instruction in the thread gets executed before the `memory_fence()` is retired.

(d) What *minimal* changes should the programmer make to the program above to ensure that the final value of `Y[0]` on RC is the same as that in part (a) on SC? Explain your answer.

Use memory fences before T1.1 and after T0.2.

Explanation. The memory fence before instruction T1.1 stalls thread 1 until instruction T1.0 has completed, i.e., ensures that `Y[0]` is initialized to 1 before the flag is set. Thread 0 waits in the loop T0.2 until the flag is set. The memory fence after instruction T0.2 ensures that instruction T0.3 will not happen until the memory fence is retired. Thus, instruction T0.3 will also complete *after* the flag is set. The modified code will be as follows:

	Thread T0		**Thread T1**
Instr. T0.0	`a = X[0];`	Instr. T1.0	`Y[0] = 1;`
Instr. T0.1	`b = a + Y[0];`		**memory_fence();**
Instr. T0.2	`while(*flag == 0);`	Instr. T1.1	`*flag = 1;`
	memory_fence();	Instr. T1.2	`X[1] *= 2;`
Instr. T0.3	`Y[0] += 1;`	Instr. T1.3	`a = 0;`

5 Building Multicore Processors [200 points]

You are hired by Amdahl's Nano Devices (AND) to design their newest multicore processor.
Ggl, one of AND's largest customers, has found that the following program can predict people's happiness.

```
for (i = 12; i < 2985984; i++) {
  past = A[i-12]
  current = A[i]
  past *= 0.37
  current *= 0.63
  A[i] = past + current
}
```

`A` is a large array of 4-byte floating point numbers, gathered by Ggl over the years by harvesting people's private messages. Your job is to create a processor that runs this program as fast as possible.

Assume the following:

- You have magically fast DRAM that allows infinitely many cores to access data in parallel. We will relax this strong assumption in parts (d), (e), (f).
- Each floating point instruction (addition and multiplication) takes 10 cycles.
- Each memory read and write takes 10 cycles.
- No caches are used.
- Integer operations and branches are fast enough that they can be ignored.

(a) Assuming infinitely many cores, what is the maximum steady state speedup you can achieve for this program? Please show all your computations.

The cycle counts are extra information that are not necessary.
Instead, the loop body is sequential, while 12 consecutive iterations are parallel. Notice that the 13th iteration is dependent on the 1st iteration.
Therefore $\frac{1}{12}$ of the program is serial. Using Amdahl's law, this solves to a maximum speedup of 12.

(b) What is the minimum number of cores you need to achieve this speedup?

12 cores are needed.

(c) Briefly describe how you would assign work to each core to achieve this speedup.

Assign iteration `i` to processor `i%12`.

It turns out magic DRAM does not exist except in Macondo[1]. As a result, you have to use cheap, slow, low-bandwidth DRAM. To compensate for this, you decide to use a private L1 cache for each processor. The new specifications for the DRAM and the L1 cache are:

- DRAM is shared by all processors. DRAM may only process one request (read or write) at a time.
- DRAM takes 100 cycles to process any request.
- DRAM prioritizes accesses of smaller addresses and write requests. (Assume no virtual memory)
- The cache is direct-mapped. Each cache block is 16 bytes.
- It takes 10 cycles to access the cache. Therefore, a cache hit is processed in 10 cycles and a cache miss is processed in 110 cycles.

All other latencies remain the same as specified earlier. Answer parts (d), (e), (f) assuming this new system.

(d) Can you still achieve the same steady state speedup as before? Circle one: YES NO

Please explain.

No. Notice that the serial bottleneck is now caused by the DRAM. In steady state, only one memory access is performed for every four iterations. So four iterations take 100 + 200 = 300 cycles. 12 iterations requires 900 cycles.
In order to achieve 12x speedup, each core needs to complete each iteration in 75 cycles. However, fetching from memory alone requires 100 cycles. Therefore, it is not possible.

By just saying that it is now slower than before is not enough, since the single core baseline is also slower.

[1] An imaginary town featured in *One Hundred Years of Solitude* by the late Colombian author Gabriel García Márquez (1927-2014).

(e) What is the minimum number of cores your processor needs to provide the maximum speedup?

3 cores are needed to take the maximum advantage of each cache hit, and a maximum speed up of almost 3 can be achieved.
Core 1 calculates iterations i+0, i+1, i+2, i+3, core 2 calculates iteration i+4, i+5, i+6, i+7, and core 3 calculates iteration i+8, i+9, i+10, i+11.

(f) Briefly describe how you would assign work to each core to achieve this speedup.

3 cores are needed to take the maximum advantage of each cache hit, and a maximum speed up of almost 3 can be achieved.
Core 1 calculates iterations i+0, i+1, i+2, i+3, core 2 calculates iteration i+4, i+5, i+6, i+7, and core 3 calculates iteration i+8, i+9, i+10, i+11.

6 Parallel Speedup [200 points]

You are a programmer at a large corporation, and you have been asked to parallelize an old program so that it runs faster on modern multicore processors.

(a) You parallelize the program and discover that its speedup over the single-threaded version of the same program is significantly less than the number of processors. You find that many cache invalidations are occuring in each core's data cache. What program behavior could be causing these invalidations (in 20 words or less)?

Cache ping-ponging due to (inefficient or poorly-designed) data sharing.

(b) You modify the program to fix this first performance issue. However, now you find that the program is slowed down by a global state update that must happen in only a single thread after every parallel computation. In particular, your program performs 90% of its work (measured as processor-seconds) in the parallel portion and 10% of its work in this serial portion. The parallel portion is perfectly parallelizable. What is the maximum speedup of the program if the multicore processor had an infinite number of cores?

Use Amdahl's Law: for n processors, Speedup(n) $= \frac{1}{0.1+\frac{0.9}{n}}$
As $n \to \infty$, Speedup(n) $\to 10$

(c) How many processors would be required to attain a speedup of 4?

6 processors.
Let Speedup(n) = 4 (from above) and solve:
$4 = 1/(0.1 + \frac{0.9}{n})$
$0.25 = 0.1 + \frac{0.9}{n}$
$0.15 = \frac{0.9}{n}$
$n = 6$

(d) In order to execute your program with parallel and serial portions more efficiently, your corporation decides to design a custom heterogeneous processor.

- This processor will have one large core (which executes code more quickly but also takes greater die area on-chip) and multiple small cores (which execute code more slowly but also consume less area), all sharing one processor die.
- When your program is in its parallel portion, all of its threads execute **only** on small cores.
- When your program is in its serial portion, the one active thread executes on the large core.
- Performance (execution speed) of a core is proportional to the square root of its area.
- Assume that there are 16 units of die area available. A small core must take 1 unit of die area. The large core may take any number of units of die area n^2, where n is a positive integer.
- Assume that any area not used by the large core will be filled with small cores.

(i) How large would you make the large core for the fastest possible execution of your program?

4 units.
For a given large core size of n^2, then the large core yields a speedup of n on the serial section, and there are $16 - n^2$ small cores to parallelize the parallel section. Speedup is thus $1/(\frac{0.1}{n} + \frac{0.9}{16-n^2})$. To maximize speedup, minimize the denominator. One can find that for $n = 1$, the denominator is 0.16. For $n = 2$, the denominator is 0.125. For $n = 3$, the denominator is 0.1619 (this can be approximated without a calculator: 0.0333 plus $0.90/7 > 0.12$ is greater than 0.15, thus worse than $n = 2$.) Hence, $n = 2$ is optimal, for a large core of $n^2 = 4$ units.

(ii) What would the same program's speedup be if all 16 units of die area were used to build a homogeneous system with 16 small cores, the serial portion ran on one of the small cores, and the parallel portion ran on all 16 small cores?

$Speedup = 1/(0.1 + \frac{0.9}{16}) = 6.4$

(iii) Does it make sense to use a heterogeneous system for this program which has 10% of its work in serial sections?
Why or why not?

Yes.
The serial portion of the program is large enough that speedup of the serial portion with the large core speedup outweighs loss in parallel throughput due to the large core.

(e) Now you optimize the serial portion of your program and it becomes only 4% of total work (the parallel portion is the remaining 96%).

(i) What is the best choice for the size of the large core in this case?

4 units.
Same as above, we can calculate n^2. Now the speedup is $1/(\frac{0.04}{n} + \frac{0.96}{16-n^2})$. Again, $n = 2$ maximizes the speedup.

(ii) What is the program's speedup for this choice of large core size?

$1/(\frac{0.04}{2} + \frac{0.96}{12}) = 10$

(iii) What would the same program's speedup be for this 4%/96% serial/parallel split if all 16 units of die area were used to build a homogeneous system with 16 small cores, the serial portion ran on one of the small cores, and the parallel portion ran on all 16 small cores?

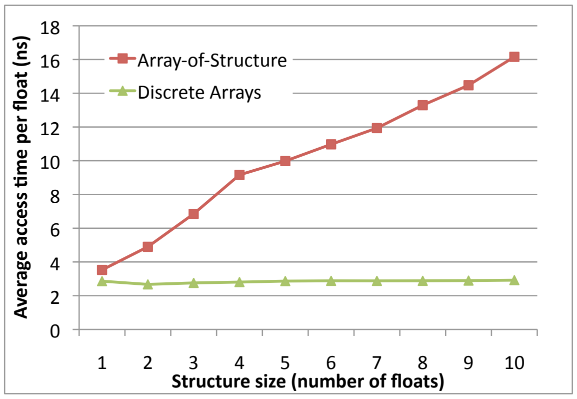
$1/(0.04 + \frac{0.96}{16}) = 1/0.1 = 10$

(iv) Does it make sense to use a heterogeneous system for this program which has 4% of its work in serial sections?
Why or why not?

No.
The heterogeneous system is more complex to design, but the performance is the same for this program.

7 AoS vs. SoA on GPU [50 points]

The next figure shows the execution time for processing an array of data structures on a GPU. Abscissas represent the number of members in a data structure. Consecutive GPU threads read consecutive structures, and compute the sum reduction of their members. The result is stored in the first member of the structure.



The green line is the time for a kernel that accesses an array that is stored as discrete sub-arrays, that is, all i-th members of all array elements are stored in the i-th sub-array, in consecutive memory locations. The red line is the kernel time with an array that contains members of the same structure in consecutive memory locations.

- Why does the red line increase linearly? Why not the green line?

GPU global memory accesses are carried out on a per-warp basis. If all threads in the same warp access the same cache line or memory segment, the efficiency is maximal. This is the case of the green line.
In the A-o-S case, consecutive threads have a stride between them. This increases the number of memory transactions that are necessary for a single warp.

- How can the effect on the red line be alleviated?

The effect for this kernel could be alleviated by the use of caches that store structure members that will be accessed during the reduction operation.

- How would both kernels perform on a single-core CPU with one level of cache? And on a dual-core CPU with individual caches? And on a dual-core CPU with a shared cache?

On a single-core CPU, the A-o-S layout benefits from the cache: structure members are cached when the first member is accessed. The DA layout might result in a similar performance, as long as the relation between structure size and cache size allows for enough cache hits per data structure.
On a dual-core CPU with individual caches, the DA layout might provoke cache conflicts in the output writing. With a shared cache, it is more likely that A-o-S and DA obtain a similar performance.

8 Histogram Calculation on GPU [150 points]

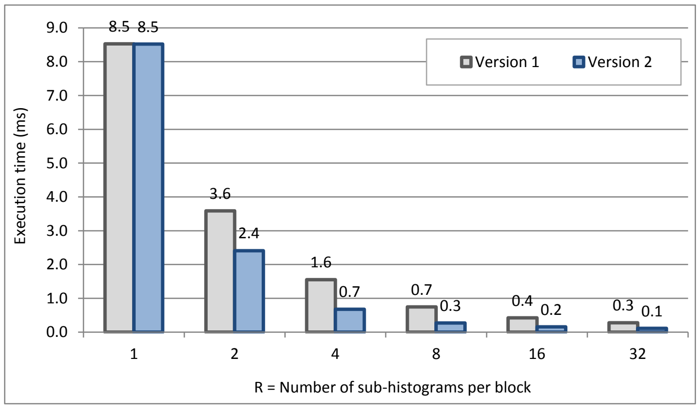
Histograms are a powerful tool in many fields, such as image processing. Their implementation on GPU is challenging because of the need for atomic operations. One way to accelerate their computation is using privatization in the fast shared memory. The following code calculates the histogram of an image "img" using privatization:

```
1 extern __shared__ unsigned int Hs[];// Dynamic shared memory allocation
2 __global__ void histogram_kernel(
3         unsigned int* histo, unsigned int* img, int size, int BINS){
4    // Block and thread index
5    const int bx = blockIdx.x;
6    const int tx = threadIdx.x;
7    // Constants for read access
8    const int begin = bx * blockDim.x + tx;
9    const int end = size;
10   const int step = blockDim.x * gridDim.x;
11   // Sub-histogram initialization
12   for(int pos = tx; pos < BINS; pos += blockDim.x) Hs[pos]=0;
13   __syncthreads();  // Intra-block synchronization
14   // Main loop
15   for(int i = begin; i < end; i += step){
16     // Global memory read
17     unsigned int d = img[i];
18     // Atomic vote in shared memory
19     atomicAdd(&Hs[d], 1);
20   }
21    __syncthreads();  // Intra-block synchronization
22    // Merge in global memory
23   for(int pos = tx; pos < BINS; pos += blockDim.x){
24     unsigned int sum = 0;
25     sum = Hs[pos];
26     // Atomic addition in global memory
27     atomicAdd(histo + pos, sum);
28   }
29 }
```

(a) As natural images are smooth (that is, they present spatial correlation), it is very likely that neighboring pixels fall into the same bin. To avoid atomic conflicts, R sub-histograms per block can be used (and later merged). Lets analyze two different ways of accessing the sub-histograms (to replace line 19):

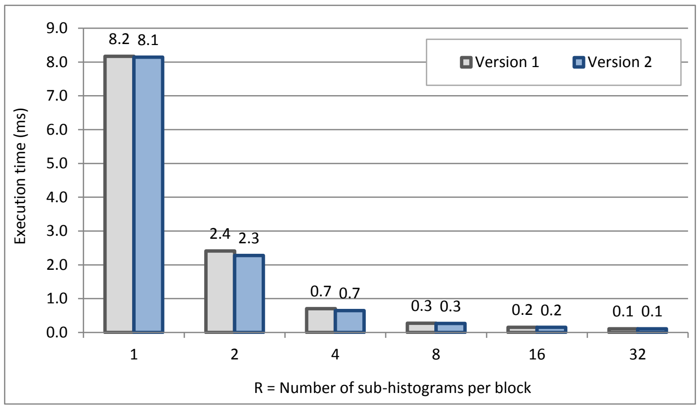
```
atomicAdd(&Hs[BINS * (tx % R) + d], 1); // Version 1
atomicAdd(&Hs[tx % R + d * R], 1); // Version 2
```

This graph shows the execution time for a 32-bins image histogram:

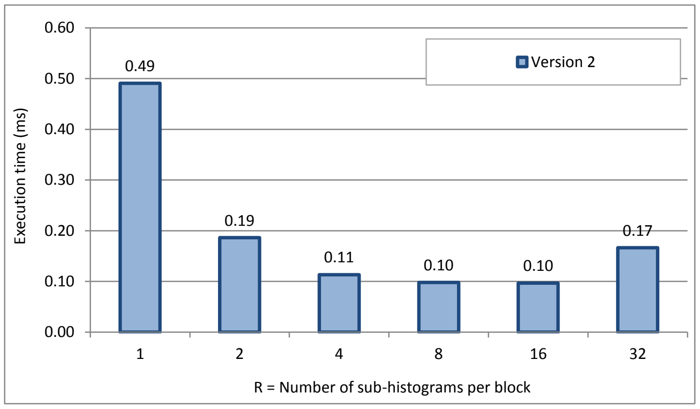


Why does version 2 obtain better results? What would happen for an odd-number-sized histogram?

Solution:
Version 1 makes consecutive threads vote in consecutive sub-histograms. If two adjacent threads have to update the same bin, a bank conflict will occur, because the shared memory has 32 banks. Differently, version 2 makes consecutive threads update bins allocated in consecutive addresses. If the size is an odd number, for instance 33, version 1 will not incur in so many bank conflicts, and the performance will be comparable. See the following graph:



(b) As can be seen in the above graph, increasing the number R of sub-histogram tends to reduce the number of atomic conflicts, and consequently the execution time. Could you then explain the following graph? (Note: Histograms of 256 bins are calculated. Tests have been carried out on a Kepler GPU with a maximum of 64 warps per multiprocessor, and 48 KB of shared memory. Blocks of 256 threads are used).



Solution:
Since there are 64 warps per multiprocessor, we can have up to 8 blocks per multiprocessor. 256 bins occupy 1,024 bytes. With up to 4 sub-histograms per block, we require 4 kB per block. Thus, we can have 8 block per multiprocessor, and the occupancy value is 100%.
With 8 sub-histograms, the occupancy falls down. The conflict reduction compensates for this occupancy reduction. However, with 32 sub-histograms, only 1 block will be active per multiprocessor. This low occupancy has a negative effect on the performance.

(c) For very large histograms, privatization in shared memory is not possible, unless multiple passes are carried out. Assume that, given the limited shared memory availability, N passes are needed. Atomic operations in shared memory take 2 ns to complete. For each pass, 10% of the input data loads hit the L2 cache. Compare this multi-pass approach to an approach where the histogram resides in global memory. Assume a GPU with global memory atomic operations in L2. Each atomic operation takes 10 ns to complete in L2, and 200 ns to complete in DRAM. 95% of the atomic operations hit the L2 cache. Find the value of N that makes worthwhile each of the approaches. (Note: the global memory bandwidth is 100 GB/s, and the L2 is 10 times faster).

Solution:
Let's first calculate the average latency for each atomic vote in global memory. We add the load latency and the atomic latency. The load latency for a 4-byte word can be estimated as 1 / 25 ns. The atomic latency equals to 5%×200+95%×10 = 19.5 ns. We obtain 19.54 ns. For each atomic vote in shared memory, we have to take into account that N passes are needed. The load latency for the first of them is 1 / 25 ns. For the subsequent N-1 passes, 10%×(1 / 250)+90%×(1 / 25) = 0.0364 ns. Thus, the total latency (load + atomic) can be estimated as 2ns+0.04ns+(N-1)×0.0364 ns. Comparing both approaches, we can estimate that the multi-pass approach can be better for N < 481.