Computer Architecture (263-2210-00L), Fall 2017
HW 4: Vector Processing, GPU, Memory Scheduling, Cache Partitioning

Instructor: Prof. Onur Mutlu
TAs: Hasan Hassan, Arash Tavakkol, Mohammad Sadr, Lois Orosa, Juan Gomez Luna

Assigned: Thursday, Nov 9, 2017
Due: **Thursday, Nov 23, 2017**

- **Handin - Critical Paper Reviews (1).** You need to submit your reviews to https://safari.ethz.ch/review/architecture/. Please check your inbox. You should have received an email with the password you can use to login to the paper review system. If you have not received any email, please contact comparch@lists.ethz.ch. In the first page after login, you should click in "Architecture - Fall 2017 Home", and then go to "any submitted paper" to see the list of papers.
- **Handin - Questions (2-7).** 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/fall2017/lib/exe/fetch.php?media=onur-comparch-f17-how-to-do-the-paper-reviews.pdf
- Some sample reviews can be found here: https://safari.ethz.ch/architecture/fall2017/doku.php?id=readings

(a) Write a one-page critical review for at least **two** of the following papers:

- E. Ebrahimi, C. J. Lee, O. Mutlu, and Y. N. Patt, "Fairness via Source Throttling: A Configurable and High-Performance Fairness Substrate for Multi-Core Memory Systems," ASPLOS 2010. https://people.inf.ethz.ch/omutlu/pub/fst_asplos10.pdf
- Y. Kim, D. Han, O. Mutlu, and M. Harchol-Balter, "ATLAS: A Scalable and High-Performance Scheduling Algorithm for Multiple Memory Controllers," HPCA 2010. https://people.inf.ethz.ch/omutlu/pub/atlas_hpca10.pdf
- S. P. Muralidhara, L. Subramanian, O. Mutlu, M. Kandemir, and T. Moscibroda, "Reducing Memory Interference in Multicore Systems via Application-Aware Memory Channel Partitioning," MICRO 2011. https://people.inf.ethz.ch/omutlu/pub/memory-channel-partitioning-micro11.pdf
- L. Subramanian, D. Lee, V. Seshadri, H. Rastogi, and O. Mutlu, "BLISS: Balancing Performance, Fairness and Complexity in Memory Access Scheduling," TPDS 2016. https://people.inf.ethz.ch/omutlu/pub/bliss-memory-scheduler_ieee-tpds16.pdf

2 Vector Processing [200 points]

You are studying a program that runs on a vector computer with the following latencies for various instructions:

- VLD and VST: 50 cycles for each vector element; fully interleaved and pipelined.
- VADD: 4 cycles for each vector element (fully pipelined).
- VMUL: 16 cycles for each vector element (fully pipelined).
- VDIV: 32 cycles for each vector element (fully pipelined).
- VRSHF (right shift): 1 cycle for each vector element (fully pipelined).

Assume that:

- The machine has an in-order pipeline.
- The machine supports chaining between vector functional units.
- In order to support 1-cycle memory access after the first element in a vector, the machine interleaves vector elements across memory banks. All vectors are stored in memory with the first element mapped to bank 0, the second element mapped to bank 1, and so on.
- Each memory bank has an 8 KB row buffer. Vector elements are 64 bits in size.
- Each memory bank has two ports (so that two loads/stores can be active simultaneously), and there are two load/store functional units available.

(a) What is the minimum power-of-two number of banks required in order for memory accesses to never stall? (Assume a vector stride of 1.)

Solution:

64 banks (because memory latency is 50 cycles and the next power of two is 64)

(b) The machine (with as many banks as you found in part a) executes the following program (assume that the vector stride is set to 1):

VLD V1 ← A
VLD V2 ← B
VADD V3 ← V1, V2
VMUL V4 ← V3, V1
VRSHF V5 ← V4, 2

It takes 111 cycles to execute this program. What is the vector length?

Solution:

```
VLD      |----50------|---(VLEN-1)----|
VLD      |1|----50------|
VADD                    |-4-|
VMUL                        |-16-|
VRSHF                            |1|-----(VLEN-1)-----|
```

1 + 50 + 4 + 16 + 1 + (VLEN - 1) = 71 + VLEN = 111 → VLEN = 40 elements

If the machine did not support chaining (but could still pipeline independent operations), how many cycles would be required to execute the same program?

Solution:

```
VLD      |-----50-----|---(VLEN-1)---|
VLD      |1|-----50-----|---(VLEN-1)---|
VADD                                  |-4-|--(VLEN-1)---|
```

```
VMUL                                                          |-16-|--(VLEN-1)---|
VRSHF                                                                          |1|--(VLEN-1)--|
```

50 + 1 + 4 + 16 + 1 + 4 × (VLEN - 1) = 68 + 4 × VLEN = 228 cycles

(c) The architect of this machine decides that she needs to cut costs in the machines memory system. She reduces the number of banks by a factor of 2 from the number of banks you found in part (a) above. Because loads and stores might stall due to bank contention, an arbiter is added to each bank so that pending loads from the oldest instruction are serviced first. How many cycles does the program take to execute on the machine with this reduced-cost memory system (but with chaining)?

Solution:

```
VLD [0]   |----50----|    bank 0  (takes port 0)
    ...
    [31]  |--31--|----50----| bank 31
    [32]             |---50---| bank 0 (takes port 0)
    ...
    [39]                      |--7--|   bank 7
VLD [0]   |1|----50----|    bank 0  (takes port 1)
    ...
    [31]  |1|--31--|----50----| bank 31
    [32]               |---50---| bank 0 (takes port 1)
    ...
    [39]                        |--7--|   bank 7
VADD                                  |--4--| (tracking last elements)
VMUL                                        |--16--|
VRSHF                                              |1|
```

(B[39]: 1 + 50 + 50 + 7) + 4 + 16 + 1 = 129 cycles

Now, the architect reduces cost further by reducing the number of memory banks (to a lower power of 2). The program executes in 279 cycles. How many banks are in the system?

Solution:

```
VLD   [0]      |---50---|
      ...
      [8]               |---50---|
      ...
      [16]                       |--50--|
      ...
      [24]                              |--50--|
      ...
      [32]                                     |--50--|
      ...
      [39]                                            |--7--|
VLD   [39]                                                  |1|
VADD                                                          |--4--|
VMUL                                                                |--16--|
VRSHF                                                                      |1|
```

5×50 + 7 + 1 + 4 + 16 + 1 = 279 cycles → 8 banks

(d) Another architect is now designing the second generation of this vector computer. He wants to build a multicore machine in which 4 vector processors share the same memory system. He scales up the number of banks by 4 in order to match the memory system bandwidth to the new demand. However, when he simulates this new machine design with a separate vector program running on every core, he finds that the average execution time is longer than if each individual program ran on the original single-core system with 1/4 the banks. Why could this be? Provide concrete reason(s).

Solution:

Row-buffer conflicts (all cores interleave their vectors across all banks).

What change could this architect make to the system in order to alleviate this problem (in less than 20 words), while only changing the shared memory hierarchy?

Solution:

Partition the memory mappings, or using better memory scheduling.

3 Vector Processing [100 points]

Consider the following piece of code:

```
for (i = 0; i < 100; i++)
  A[i] = ((B[i] * C[i]) + D[i])/2;
```

(a) Translate this code into assembly language using the following instructions in the ISA (note the number of cycles each instruction takes is shown next to each instruction):

Opcode	Operands	Number of Cycles	Description
LEA	Ri, X	1	Ri ← address of X
LD	Ri, Rj, Rk	11	Ri ← MEM[Rj + Rk]
ST	Ri, Rj, Rk	11	MEM[Rj + Rk] ← Ri
MOVI	Ri, Imm	1	Ri ← Imm
MUL	Ri, Rj, Rk	6	Ri ← Rj × Rk
ADD	Ri, Rj, Rk	4	Ri ← Rj + Rk
ADD	Ri, Rj, Imm	4	Ri ← Rj + Imm
RSHFA	Ri, Rj, amount	1	Ri ← RSHFA (Rj, amount)
BRcc	X	1	Branch to X based on condition codes

Assume one memory location is required to store each element of the array. Also assume that there are 8 registers (R0 to R7).

Condition codes are set after the execution of an arithmetic instruction. You can assume typically available condition codes such as zero, positive, and negative.

Solution:

```
MOVI    R1, 99      // 1 cycle
LEA     R0, A       // 1 cycle
LEA     R2, B       // 1 cycle
LEA     R3, C       // 1 cycle
LEA     R4, D       // 1 cycle
LOOP:
LD      R5, R2, R1  // 11 cycles
LD      R6, R3, R1  // 11 cycles
MUL     R7, R5, R6  // 6 cycles
LD      R5, R4, R1  // 11 cycles
ADD     R8, R7, R5  // 4 cycles
RSHFA   R9, R8, R1  // 1 cycle
ST      R9, R0, R1  // 11 cycles
ADD     R1, R1, -1  // 4 cycles
BRGEZ   R1 LOOP     // 1 cycle
```

How many cycles does it take to execute the program?

Solution:

$5 + 100 \times 60 = 6005$ cycles

(b) Now write Cray-like vector assembly code to perform this operation in the shortest time possible. Assume that there are 8 vector registers and the length of each vector register is 64. Use the following instructions in the vector ISA:

Opcode	Operands	Number of Cycles	Description
LD	Vst, #n	1	Vst ← n (Vst = Vector Stride Register)
LD	Vln, #n	1	Vln ← n (Vln = Vector Length Register)
VLD	Vi, X	11, pipelined	
VST	Vi, X	11, pipelined	
Vmul	Vi, Vj, Vk	6, pipelined	
Vadd	Vi, Vj, Vk	4, pipelined	
Vrshfa	Vi, Vj, amount	1	

Solution:

```
LD      Vln, 50
LD      Vst, 1
VLD     V1, B
VLD     V2, C
VMUL    V4, V1, V2
VLD     V3, D
VADD    V6, V4, V3
VRSHFA  V7, V6, 1
VST     V7, A

VLD     V1, B + 50
VLD     V2, C + 50
VMUL    V4, V1, V2
VLD     V3, D + 50
VADD    V6, V4, V3
VRSHFA  V7, V6, 1
VST     V7, A + 50
```

How many cycles does it take to execute the program on the following processors? Assume that memory is 16-way interleaved.

(i) Vector processor without chaining, 1 port to memory (1 load or store per cycle).

Solution:

The third load (VLD) can be pipelined with the add (VADD). However as there is just only one port to memory and no chaining, other operations cannot be pipelined.
Processing the first 50 elements takes 346 cycles as below

```
| 1 | 1 | 11 | 49 | 11 | 49 | 6 | 49 |
                                     | 11 | 49 | 4 | 49 | 1 | 49 | 11 | 49 |
```

Processing the next 50 elements takes 344 cycles as shown below (no need to initialize Vln and Vst as they stay at the same value).

```
| 11 | 49 | 11 | 49 | 6 | 49 |
                             | 11 | 49 | 4 | 49 | 1 | 49 | 11 | 49 |
```

Therefore, the total number of cycles to execute the program = 690 cycles

(ii) Vector processor with chaining, 1 port to memory

Solution: In this case, the first two loads cannot be pipelined as there is only one port to memory and the third load has to wait until the second load has completed. However, the machine supports chaining, so all other operations can be pipelined.
Processing the first 50 elements takes 242 cycles as below

```
| 1 | 1 | 11 |     49    | 11 |    49    |
                                   | 6 |    49    |
                                                | 11 |     49      |
                                                       | 4 |      49     |
                                                            | 1 |       49      |
                                                                        | 11 |     49    |
```

Processing the next 50 elements takes 240 cycles (same time line as above, but without the first 2 instructions to initialize Vln and Vst).

Therefore, the total number of cycles to execute the program = 482 cycles

(iii) Vector processor with chaining, 2 read ports and 1 write port to memory

Solution:

Assuming an in-order pipeline.
The first two loads can also be pipelined as there are two ports to memory. The third load has to wait until the first two loads complete. However, the two loads for the second 50 elements can proceed in parallel with the store.

```
| 1 | 1 | 11 |    49    |
          | 1 | 11 |    49    |
                     | 6 |    49    |
                           | 11 |    49    |
                                  | 4 |    49    |
                                       | 1 |    49    |
                                            | 11 |    49    |
                                             | 11 |    49    |
                                              | 11 |    49    |
                                                    | 6 |    49    |
                                                              | 11 |    49    |
                                                                    | 4 |    49    |
                                                                         | 1 |    49    |
                                                                              | 11 |    49    |
```

Therefore, the total number of cycles to execute the program = 215 cycles

4 GPUs [150 points]

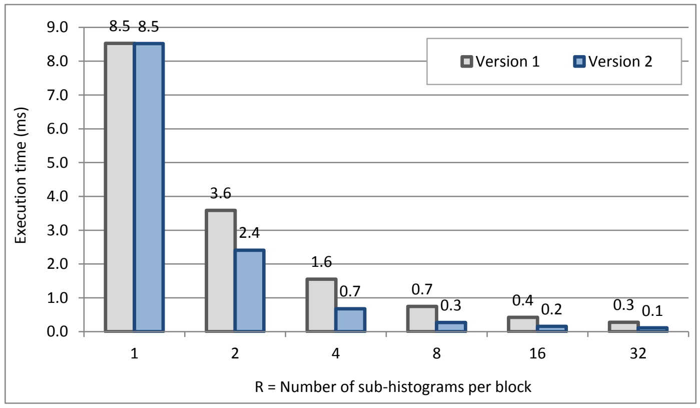
Histograms are a powerful tool in many fields, such as image processing. Their implementation on GPU is challenging because of the need for atomic operations. One way to accelerate their computation is using privatization in the fast shared memory. The following code calculates the histogram of an image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

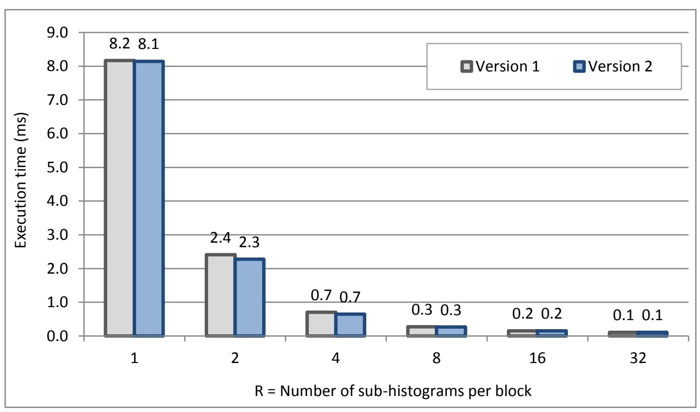
This graph shows the execution time for a 32-bins image histogram:



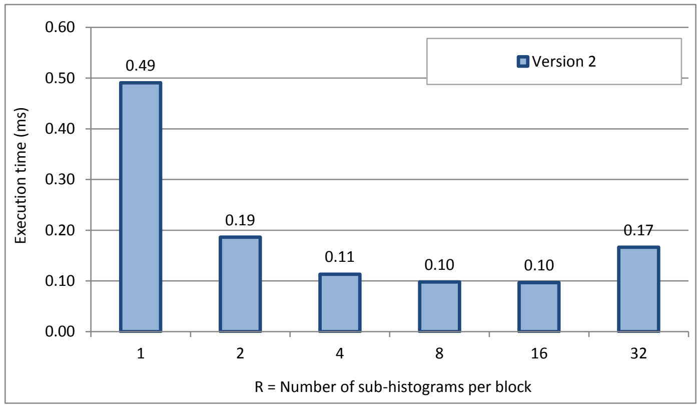
Why does version 2 obtain better results? What would happen for an odd-number-sized histogram?

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.

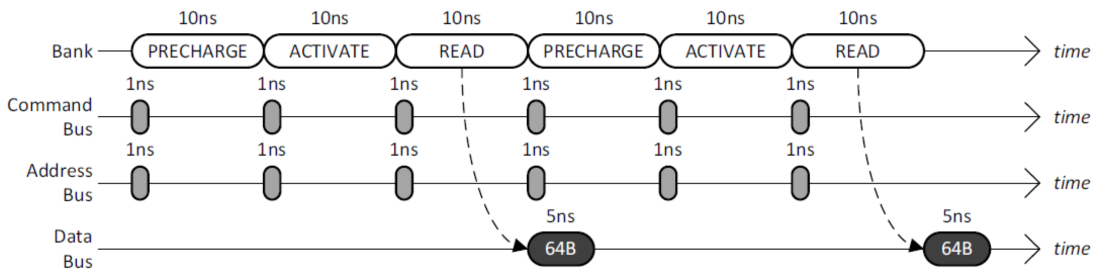
(c) For very large histograms, privatization in shared memory is not possible, unless multiple passes are carried out. Assume that, given the limited shared memory availability, N passes are needed. Atomic operations in shared memory take 2 ns to complete. For each pass, 10% of the input data loads hit 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 $5\% \times 200 + 95\% \times 10 = 19.5$ 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, $10\% \times (1 / 250) + 90\% \times (1 / 25) = 0.0364$ ns. Thus, the total latency (load + atomic) can be estimated as $2\text{ns} + 0.04\text{ns} + (N\text{-}1) \times 0.0364$ ns. Comparing both approaches, we can estimate that the multi-pass approach can be better for $N < 481$.

5 Memory Scheduling [200 points]

Row-Buffer Conflicts. The following timing diagram shows the operation of a single DRAM channel and a single DRAM bank for two back-to-back reads that conflict in the row-buffer. Immediately after the bank has been busy for 10ns with a READ, data starts to be transferred over the data bus for 5ns.



(a) Given a long sequence of back-to-back reads that always conflict in the row-buffer, what is the data throughput of the main memory system? Please state your answer in **gigabytes/second**.

Solution:

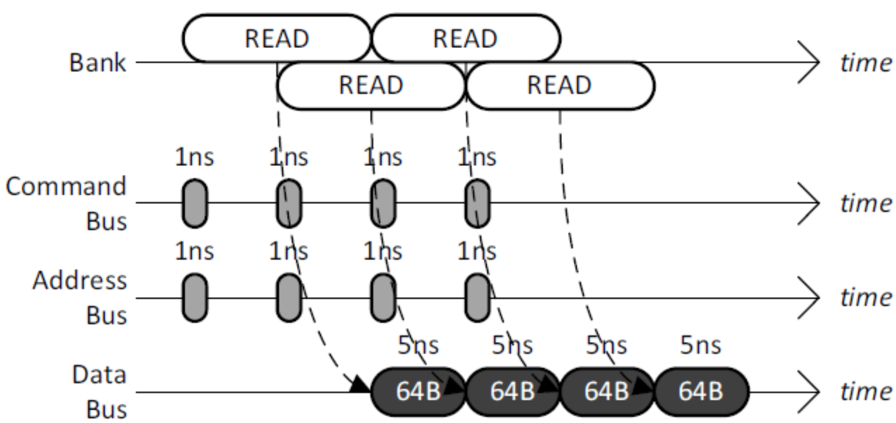
64B/30ns = 32B/15ns = 32GB/15s = 2.13GB/s

(b) To increase the data throughput, the main memory designer is considering adding more DRAM banks to the single DRAM channel. Given a long sequence of back-to-back reads to all banks that always conflict in the row-buffers, what is the minimum number of banks that is required to achieve the maximum data throughput of the main memory system?

Solution:

30ns/5ns = 6

Row-Buffer Hits. The following timing diagram shows the operation of the single DRAM channel and the single DRAM bank for four back-to-back reads that hit in the row-buffer. It is important to note that rowbuffer hits to the same DRAM bank are pipelined: while each READ keeps the DRAM bank busy for 10ns, up to at most **half** of this latency (5ns) can be overlapped with another read that hits in the row-buffer. (Note that this is different from Lab 6 where we unrealistically assumed that row-buffer hits are non-pipelined.)



(c) Given a long sequence of back-to-back reads that always hits in the row-buffer, what is the data throughput of the main memory system? Please state your answer in **gigabytes/second**.

Solution:

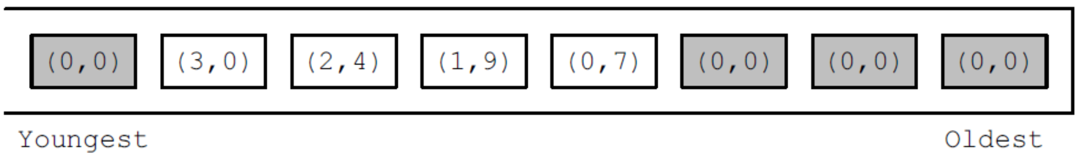
64B/5ns = 64GB/5s = 12.8GB/s

(d) When the maximum data throughput is achieved for a main memory system that has a single DRAM channel and a single DRAM bank, what is the bottleneck that prevents the data throughput from becoming even larger? **Circle** all that apply.

BANK   COMMAND BUS   ADDRESS BUS   <u>DATA BUS</u>

Memory Scheduling Policies. The diagram below shows the memory controllers request queue at time 0. The shaded rectangles are read requests generated by thread T0, whereas the unshaded rectangles are read requests generated by thread T1. Within each rectangle, there is a pair of numbers that denotes the requests (BankAddress, RowAddress). Assume that the memory system has a single DRAM channel and four DRAM banks. Further assume the following.

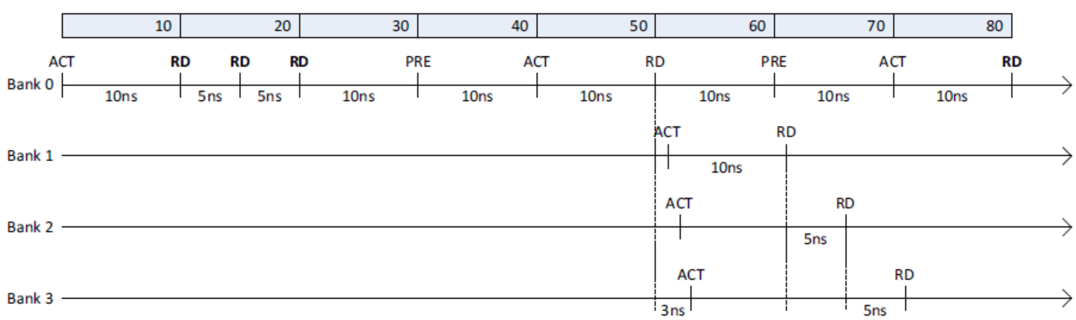
- All the row-buffers are closed at time 0.
- Both threads start to stall at time 0 because of memory.
- A thread continues to stall until it receives the data for all of its requests.
- Neither thread generates more requests.



For extra credits (50 points), please make sure that you model contention in the banks as well as in all of the buses (address/command/data).

(e) For the FCFS scheduling policy, calculate the memory stall time of T0 and T1.

Solution:

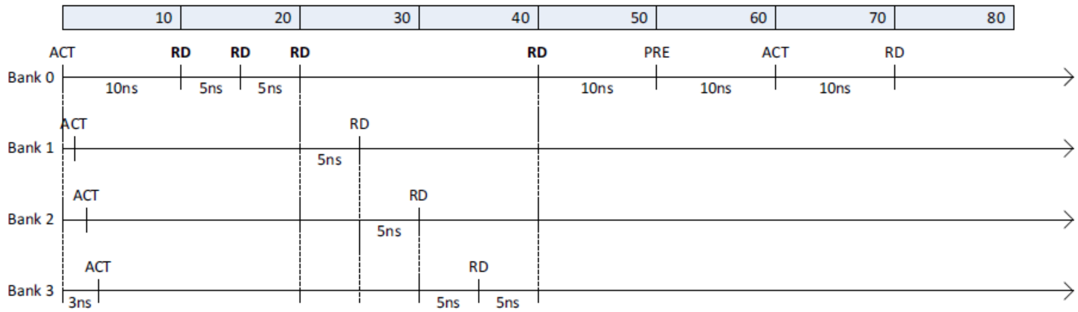


T0: (10 + 5 + 5 + 10 + 10 + 10) + 10 + 10 + 10 + 10 + 5 = 95ns

T1: (10 + 5 + 5 + 10 + 10 + 10) + 1 + 10 + 5 + 5 + 10 + 5 = 86ns

(f) For the FR-FCFS scheduling policy, calculate the memory stall time of T0 and T1

Solution:

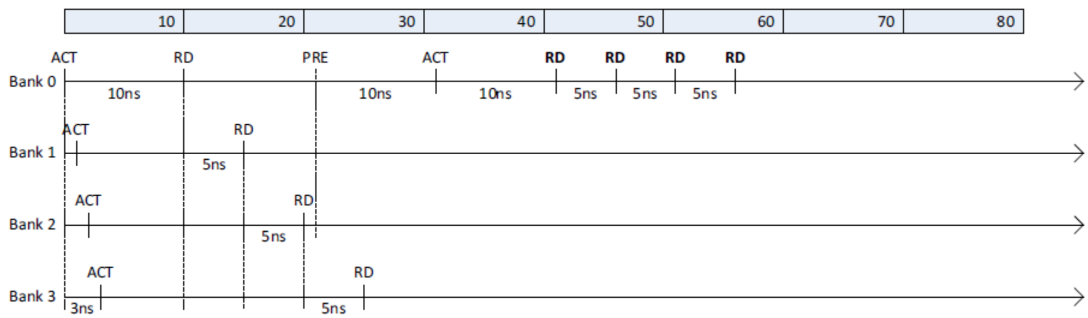


T0: (10 + 5 + 5 + 5 + 5 + 5 + 5) + 10 + 5 = 55ns

T1: (10 + 5 + 5 + 5 + 5 + 5 + 5) + 10 + 10 + 10 + 10 + 5 = 85ns

(g) For the PAR-BS scheduling policy, calculate the memory stall time of T0 and T1. Assume that all eight requests are included in the same batch.

Solution:



T0: (10 + 5 + 5) + 1 + 10 + 10 + 5 + 5 + 5 + 10 + 5 = 71ns

T1: (10 + 5 + 5) + 5 + 10 + 5 = 40ns

Another Solution:

(e) For the FCFS scheduling policy, calculate the memory stall time of T0 and T1.

Solution:

Bank 0 is the critical path for both threads.
T0 = Closed + Pipelined-Hit + Pipelined-Hit + Con ict + Con ict + Data = (ACT+RD)+(RD/2)+(RD/2)+(PRE+ACT-
= 20ns + 5ns + 5ns + 30ns + 30ns + 5ns = 95ns

T1 = Closed + Pipelined-Hit + Pipelined-Hit + Con ict + Data = (ACT+RD)+(RD/2)+(RD/2)+(PRE+ACT+RD)+DA
= 20ns + 5ns + 5ns + 30ns + 5ns = 65ns

(f) For the FR-FCFS scheduling policy, calculate the memory stall time of T0 and T1

Solution:

Bank 0 is the critical path for both threads. First, we serve all four shaded requests since they are row-buffer hits. Lastly, we serve the unshaded request.

T0 = Closed + Pipelined-Hit + Pipelined-Hit + Pipelined-Hit + Data = (ACT+RD)+(RD/2)+(RD/2)+(RD/2)+DATA
= 20ns + 5ns + 5ns + 5ns + 5ns = 40ns

T1 = Closed + Pipelined-Hit + Pipelined-Hit + Pipelined-Hit + Con ict + Data = (ACT+RD)+(RD/2)+(RD/2)+(RD/2
= 20ns + 5ns + 5ns + 5ns + 30ns + 5ns = 70ns

(g) For the PAR-BS scheduling policy, calculate the memory stall time of T0 and T1. Assume that all eight requests are included in the same batch.

Solution:

First, we serve all four unshaded requests in parallel across the four banks. Then, we serve all four shaded requests in serial.

T0 = Closed + Con ict + Pipelined-Hit + Pipelined-Hit + Pipelined-Hit + Data = (ACT+RD)+(PRE+ACT+RD)+(RD/
= 20ns + 30ns + 5ns + 5ns + 5ns + 5ns = 70ns

T1 = Closed + Data = (ACT+RD)+DATA = 20ns + 5ns = 25ns

6 Memory Scheduling [50 points]

In class, we covered "parallelism-aware batch scheduling," which is a memory scheduling algorithm that aims to reduce interference between threads in a multi-core system.

(a) What benefit does request batching provide in this algorithm?

Solution:

Request batching allows PAR-BS to avoid starvation as requests of older batches are always prioritized over requests of younger batches.

(b) How does the algorithm preserve intra-thread bank parallelism?

Solution:

Threads are ranked based on the number of requests they have at all the banks. All banks service requests based on this ranking. Hence, requests from the same thread will likely be serviced in parallel in different banks, which preserves the threads bank-level parallelism.

(c) If thread ranking was formed in a "random manner" (i.e., threads were assigned a random rank), would each thread's parallelism be preserved? Why or why not? Explain.

Solution:

It depends on the applications. Assume there are two banks and there is one application which has a lot of requests to one bank and no requests to the other. A random ranking can prioritize this application over others and thereby preventing the other applications from exploiting their bank-level parallelism.

7 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.