IB Computer Design - Supervision 5

Nandor Licker <nl364@cl.cam.ac.uk>

Due noon before the day of supervision

Some of these questions might go outside the bounds of the course, but it might be a good idea to look into these further topics. For example, some of the problems experienced by scatter/gather instructions might apply to GPU memory access as well.

Q1 Solve the past papers from the last year.

Q2 Consider the following diagram of an actual DDR3 RAM IC[1]:

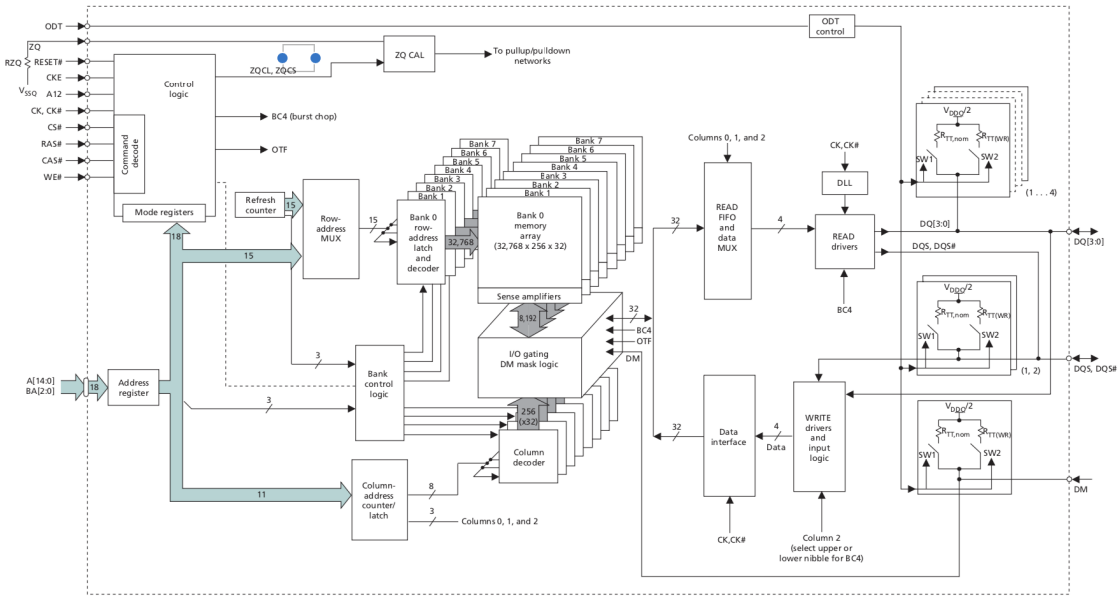

Figure 1: Internals of a DDR3 RAM chip

These ICs are usually placed on DIMM modules [2], either on one or both sides.

1. Why does the IC have a FIFO buffer inside, of 64 bits, when the data bus is only 8 bits in size? What does this say about memory access times and actual DRAM cell read times?
2. Why is the size of the buffer 64 bits? Think about typical cache line sizes and look at how these chips are organised in the DIMM.
3. What is the addressable unit of the memory IC? How about the module? Why?
4. Describe what needs to be loaded from which bank/array/row when accessing the byte at the physical address 123456.

[1] https://www.micron.com/~/media/documents/products/data-sheet/dram/ddr3/2gb_ddr3_sdram.pdf

[2] https://www.micron.com/-/media/client/global/documents/products/data-sheet/modules/unbuffered_dimm/jsf18c256_512_1gx72az.pdf

Q3 Inspect the following GeForce 2080 TI graphics card which has a total of 11Gb of RAM. What can you say about the organisation of the DRAM ICs on the board? Find the memory bus width first.

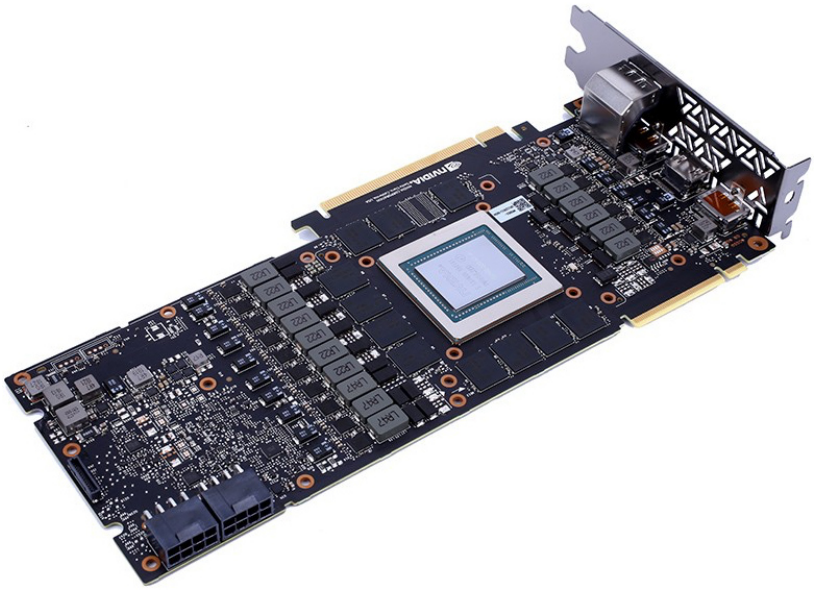

Figure 2: GeForce 2080 Ti card without heat sink

Q4 Consider the following matrix multiplication implementation:

```
void tiled_multiply(const float *a, const float *b, float *c)
{
  for (int bj = 0; bj < N; bj += T) {
    for (int bk = 0; bk < N; bk += T) {
      for (int i = 0; i < N; ++i) {
        for (int j = bj; j < bj + T; ++j) {
          float r = 0.0;
          for (int k = bk; k < bk + T; ++k) {
            r += a[i * N + k] * b[k * N + j];
          }
          c[i * N + j] += r;
        }
      }
    }
  }
}
```

Assuming a block size of 8, provide an implementation using the AVX-2 instruction set extension in C. You should use intrinsic functions, enabled using the `-mavx2` flag passed to the compiler and exposed in the `<immintrin.h>` header. You may find the following intrinsics useful:

- `_mm256_castps256_ps128`
- `_mm256_dp_ps`
- `_mm256_i32gather_ps`
- `_mm256_load_ps`
- `_mm_add_ss`
- `_mm_load_ss`
- `_mm_store_ss`

Q5 Answer the following questions about vectorised instructions:

1. What are the challenges of implementing gather/scatter instructions? Think about cache lines. Implement the 8-lane gather instruction with other SSE instructions, without using gather. Why is it advantageous to have a single instruction doing that much work?
2. Why doesn't the 8-lane dot product actually compute a dot product?
3. Show how you would implement a masked instruction from the AVX-512 instruction set using only AVX-2 instructions. Comment on the advantages of masks. Comment on how masks can reduce power usage in an actual hardware implementation.
4. Transpose a N×N matrix using scatter instructions. What is the bottleneck of a naive implementation? How does tiling help? Compare the number of bytes written to main memory.
5. How effective is hyperthreading when both thread run vector-heavy code? Could you transpose two matrices on two threads in the same time as one matrix on one thread?

Q6 Consider the following RISC-V instruction sequence:

```
lw t0, 10(r0)
addi t0, t0, 1
sw t0, 10(r0)
```

Answer the following questions:

1. Draw a timing diagram indicating how this instruction would execute on a typical 5-stage pipeline. Consider the `load`, `store` and `add` instructions. Describe what happens during the execution of each instruction in each of the pipeline stages, enumerating the actions in a list.
2. Show how you would reduce the pipeline to 4 stages. What stages would you keep and what new stage would you introduce? Show how the instructions would execute on this pipeline.
3. What is the main disadvantage of the previous implementation? Increase the maximum clock rate by providing a 4-stage implementation which does not stall. Describe the function of each pipeline stage.

   Hint: Consider `lw t0, 10(r0)`. Compute `r0 + 10` in one stage and do the load in another. In which stage do you need to perform the addition for the `addi` instruction?
4. Describe in detail at least 3 different ways of forwarding `t0` to the last add instruction.

   ```
   addi t0, t0, 1
   addi t1, t0, 1
   addi t2, t1, t0
   ```

Q7 The `-ftrapv` option available in `gcc` and `clang` emits instructions to trap on integer overflow. Consider the following function:

```
int f(int a, int b) {
  return a + b;
}
```

1. Why are integer overflows problematic? You can answer with a list of exploded rockets.
2. What instructions would you emit to implement the given function, detecting overflow? Show the instruction listings generated by `clang` and comment on the performance of each implementation on X86, ARMv7 and AArch64. If you are brave, build a version of `clang` with the MIPS backend:

```
cmake .. \
  -G Ninja \
  -DCMAKE_BUILD_TYPE=MinSizeRel \
  -DCMAKE_INSTALL_PREFIX=<path-to-install-dir> \
  -DLLVM_TARGETS_TO_BUILD="X86;AArch64;Mips;ARM" \
  -DLLVM_ENABLE_DUMP=ON
```

You can then compile your source using:

```
g++ overflow.c -ooverflow.S -S -ftrapv -c -O3 -fomit-frame-pointer -target mips
```

3. How does hardware support for exceptions on overflow facilitate the implementation of a safe language? Are precise or imprecise exceptions required?
4. How could you optimise null pointer checks using exceptions? You can look at Java.
5. Assuming you have a bytecode interpreter which has a `GET n` instruction to fetch a word from virtual address n, show how you could build a safe interpreter without performing a bounds check. The memory used by the bytecode VM is linear and of size `s`.