Detailed summaries of the classes of Arquiteturas de Alto Desempenho 2024/2025

Tomás Oliveira e Silva

Theoretical lectures:

Date	Summary
23/09	Rules. Overview of the main features of a modern processor.
30/09	Software pipelining. Virtual memory.
07/10	Caches and branch prediction
14/10	SIMD
21/10	SIMD and description of the first assignment.
28/10	CUDA.
04/11	CUDA and the first assignment.
11/11	Performance metrics. Amdahl's law.
18/11	VHDL and description of the second assignment.
25/11	Testbenches, ripple carry adder, barrel shifter.
02/12	VHDL implementation examples.
09/12	Possible ways to implements parts of the second assignment.
16/12	Possible topics for exam questions.

Practice classes:

Date	Summary
23/09 and 24/09	Comparison of C, Java, Python, Fortran, web assembly, and compiled Python (codon).
30/09 and 01/10	Code parallelization using pthreads.
07/10 and 08/10	Code parallelization using OpenMP.
14/10 and 15/10	Code parallelization using sockets (server/client).
21/10 and 22/10	Code parallelization using MPI (Message Passing Interface).
28/10 and 29/10	Code parallelization using CUDA and OpenCL (part 1).
04/11 and 05/11	Code parallelization using CUDA and OpenCL (part 2).
11/11 and 12/11	First assignment.
18/11 and 19/11	First assignment.
25/11 and 26/11	Simulation of combinational logic described in VHDL.
02/12 and 03/12	Simulation of sequential logic described in VHDL.
09/12 and 10/12	Second assignment.
16/12 and 17/12	Second assignment.

Hyperlinks to external documents and web pages are in blue.

[**Preliminary notes**] These summaries do not replace attending (and paying attention) to the theoretical lectures. To better organize the contents of these extended summaries, sometimes we took the liberty of moving things around: something reported as being lectured on a given day may actually have been lectured in the previous or in the next week. They also contain extra information not presented in in lectures.

September 23, 2024

[**Grades**] One written exam in January (grade T), and reports of two practice assignments (grades A_1 and A_2). The written exam has 12 equally valued questions, the best 10 of which will contribute to T. The practice grade is $P = \max\left(\frac{A_1+A_2}{2}, \frac{3A_1+2A_2}{5}\right)$. The final grade is $F = \frac{T+P}{2}$. It is necessary to have $T \geqslant 7$, and $P \geqslant 7$ to pass.

[**MIPS pipeline overview**] The MIPS pipeline has 5 stages:

IF Instruction Fetch: reads instructions, one by one, from a dedicated memory. Increments the program counter.

ID Instruction Decode: reads registers, sign- or zero-extends an immediate value.

EX Execute: performs an arithmetic or logical operation. The arithmetic operation may be the computation of a memory address. Also, a separate unit computes a branch target address.

MEM Memory access: performs a read (load) from a second dedicated memory area, or a write (store) to the dedicated memory area.

WB Write Back: commit a new register value to the register file.

The usage of separate address spaces for instructions and data (Harvard architecture) is not practical.

[**Protection levels**] Modern processors execute code is one of several protection levels (called rings). Ring 0 is for kernel code (operating system and drivers): all instructions are allowed but some may be virtualized. In Intel/AMD, ring 3 is for user programs (for normal users and even for super users): some instructions are privileged and if the program tries to execute them the program is terminated (the illegal instruction signal is raised). Ring -1 is for hipervisors (handle virtualized instructions). Ring -2 is for system management mode (SMM). Ring -3 is for the Management Engine (ME). These last two ring levels are usually only engaged in the BIOS.

[**Address translation (virtual addresses)**] All modern processors have a virtual memory mechanism. All addresses of a program are virtual addresses. Even the kernel uses virtual addresses (that is what makes virtualization possible). On each memory access the virtual address is translated into a physical address.

[**Caches**] On a modern processor the memory area used in the IF stage is the level 1 instruction cache (L1I), and the one used in the MEM stage is the level 1 data cache (L1D). The two caches fetch/store data in a unified level 2 cache (L2U), that stores both instructions and data (thus, instructions and data share a common address space — von Neumann architecture).

The processor may also have more than one execution pipeline (each pipeline has its own level 1 and level 2 caches). If so, a larger capacity level 3 cache (L3U) is often used. Besides caching data from several pipelines, is also guarantees the coherence of the information in the lower level caches.

Modern Intel/AMD processors can store the decoded instructions in a special cache (Op Cache). This is so because due to their variable length the instructions may take some time to decode.

There exists a special cache, called a Translation Lookaside Buffer (TLB) to cache virtual to physical address translations.

Usually, the caches are acessed using physical addresses.

[**Prefetch**] Modern processors have memory prefetch units that monitor the address access patterns used by the program and prefetch data into the data caches before it is needed. They detect strided memory accesses (arithmetic progressions).

[**Superscalar architecture**] The IF stage of the pipeline of modern processors can fetch more than one instruction in one go. This takes advantage of the wide data path between the instruction cache and the IF unit. The EX and MEM stages also have many execution units. Some may perform arithmetic or logical operations, others may perform floating point operations, others may compute memory addresses, and others may perform loads from and stores to memory. The WB stage is sometimes called the instruction retirement unit, and commits the changes made by the instructions in program order.

All this allows the pipeline to execute more than one instruction at the same time, provided, of course, that the instructions work on independent data. For example, in the following code

```
d = a + b; // 1
e = b + c; // 2
f = a + c; // 3
g = d + e; // 4, depends on 1 and 2
```

the first three statements can be performed in parallel, but the fourth has to wait for the completion of the first two (it is usually not necessary to wait for the instructions to retire, the use of data forwarding paths makes the results available from the execution units as soon as possible). Compilers are aware of this when they generate executable code, For each code block, they construct a so-called dependency graph, and output assembly code that takes advantage of any possible data-level parallelism in the source code.

[**Speculative execution**] Conditional branches are big a problem for the pipeline. The next instruction to be fetched after a conditional branch depends on a test whose outcome will be known only several clock cycles later. Modern processors employ a technique called speculative execution to try to ameliorate this problem: it will continue to fetch and (speculatively) execute instructions. The idea is to use a so-called branch predictor to attempt to predict the outcome of the conditional branch. If the predictor is successful, the IF unit fetched the right instructions and things will proceed at full pace. If it fails, something that will only be known some clock cycles later, the instructions that entered the pipeline after the mispredicted conditional branch have to be discarded. The penalty is several (up to about 10) wasted clock cycles.

The branch predictor has to be tightly coupled with the level 1 instruction cache. Branch predictors base their decision on the past behavior of the conditional branch.

[**Out-of-order (OoO) execution**] Modern processors can execute independent instructions out of order. This can happen when an instruction takes many cycles to complete. If some instructions that follows it do not depend on its side effects they are executed out of order. They are, however, retired in order. For example, in the following code

```
c = a / b;     // 1, slow (many clock cycles)
d = a + 2 * b; // 2, fast (1 clock cycle)
e = 2 * a + b; // 3, fast
g = d + e;     // 4, depends on 2 and 3, fast
```

the last three statements with very likely finish execution (but are not retired) before the first statement finishes execution. Each pipeline of a modern processor can have many (> 50) instructions in-flight.

Although there exist a fixed number of architectural registers in the instruction set architecture (for example, 16 general purpose registers for 64-bit Intel/AMD processors), due to the many instructions that can be in-flight, the pipeline internally has physical space for many more registers. When each instruction is issued it is assigned to it, using a technique called register renaming, some of the available physical registers. When the instruction is retired, the contents of the destination physical register is transferred to the architectural register file, that holds the official state of the program.

Register renaming is useful to break false dependency chains. For example, in the following code

```
m = a[i];      // 1
m |= 3;        // 2, depends on 1
a[i + 1] = m; // 3, depends on 2
m = a[i + 2]; // 4, does not depend on 2
```

statement 4 can theoretically start execution at the same time as statement 1. Because `m` will reside in a given architectural register, without register renaming statement 4 would have to wait for the completion of statement 2 (previous last change of `m`). With register renaming, is can be issued at the same time as statement 1.

[**SIMD**] Besides the regular registers, modern processors also have a set of wide registers. Each wide register stores a small vector of integer of floating point values. For example, a 128-bit register can store a vector of sixteen 8 integers, or a vector of eight 16-bit integers, or a vector of four 32-bit integers, or a vector of two 64-bit integers. The arithmetic or logical operations on these wide register work in parallel. Hence it is possible, for example using 128-bit registers, to add in parallel, in one clock cycle, say, four 32-bit integers.

[**SMT**] Modern processors also allow two or more execution flows on the same pipeline (this is called Simultaneous MultiThreading, SMT; Intel calls it hyperthreading, but that is just a marketing term). The threads that execute on the same pipeline share almost all of its resources. The exception is the register file: each thead has its own register file. SMT can make good use of execution units of the pipeline when one execution flow stalls due to a lengthy operation (say, a memory read, or a partial pipeline flush due to a mispredicted conditional branch).

[**Many cores**] Finally, all modern processors have multiple execution pipelines (one execution pipeline is one physical core). Due to simultaneous multithreading, one physical core will give rise to two or more virtual cores (vCPUs in cloud parlance). Modern server processors can have more than 100 physical cores!

[**All of the above**] All the features described above makes writing a super fast program for a modern processor a challenging task. The compiler takes care of some of the details, but to make a program super fast, the programmer should be aware of what may slow a program down. In particular, the size of the data caches is something that sometimes has to be taken in consideration. Also, if the problem to be solved allows it, SIMD instructions and multiple execution flows (threads) should be used. This last point will be explored in depth in the practice classes.

The figure on the next page, extracted from a recent AMD presentation at Hop Chips 2024, depicts the general architecture of a modern server processor pipeline. The floating point part of the pipeline is displayed in violet, and the integer part in light blue. Although there exist 16 architectural integer registers, there exist 240 physical integer registers! The floating point situation is even more extreme: 32 SIMD architectural registers and 384 physical registers. In each clock cycle, the pipeline can handle up to 6 integer instructions, 4 address calculations (the AGU is an Address Generation Unit), and 4 floating point instructions. In each clock cycle, it can also handle 4 read data transfers and 2 write data transfers from/to the L1 data cache.

September 30, 2024

[**Software pipelining**] Superscalar processor make possible several code optimizations. Consider for example, the following code:

```
for(i = 0;i < 1000;i++)
{
  x0 = a[i];    // first part of the loop
  x1 = x0 + 3; // first part of the loop
  x2 = 7 * x1; // second part of the loop
  b[i] = x2;    // second part of the loop
}
```

The statements inside the loop must be executed sequentially because they form a dependency chaini (to compute `x1` the value of `x0` is necessary and so on. A technique known as software pipelining can be used to break the loop in two parts, and to execute them potentially in parallel (superscalar!):

```
x0 = a[0];      // first part for i=0
x1 = x0 + 3;    // first part for i=0
for(i = 0;i < 999;i++)
{
```

“Zen 5” Microarchitecture Overview

2 Threads/Core

NextGen Branch Predictor

Caches

- I-Cache: 32KB, 8-way; 2x 32B fetch/cycle
- Op-Cache: 6K inst; 2x 6-wide fetch/cycle
- D-Cache: 48KB, 12-way; 4 mem ops/cycle
- L2-Cache: 1MB, 16-way

Dual I-Fetch/decode pipes, 4 inst/pipe

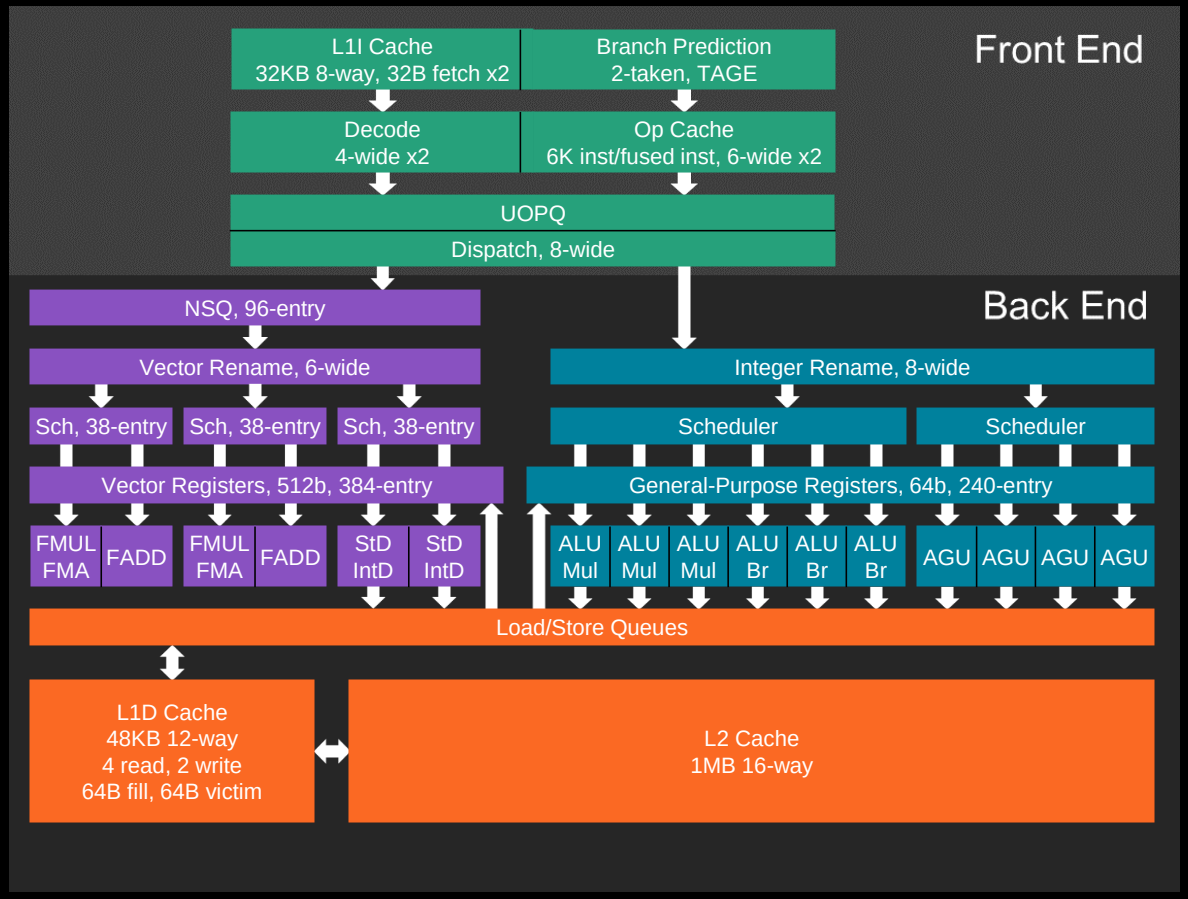
8 ops/cycle dispatched to Integer or FP

Execution capabilities

- 6 integer ALU
- 4 AGU, 4 addresses to LS per cycle
- 4 FP ops/cycle; 2cycle FADD

TLBs

- L1: 64entry ITLB, 96entry DTLB
- L2: 2K ITLB; 4K DTLB everything but 1G



4 NEXT GENERATION “ZEN 5” CORE | AUGUST 2024 AMD

Slide 4 of AMD presentation at Hop Chips 2024 (link).

```
    x2 = 7 * x1; // second part for i   (a)
    b[i] = x2;   // second part for i   (b)
    x0 = a[i];   // first part for i+1 (a)
    x1 = x0 + 3; // first part for i+1 (b)
  }
  x2 = 7 * x1;   // second part for i=999
  b[999] = x2;   // second part for i=999
```

Now the (a) statements can be put together and can also be executed together, and the same happens with the (b) statements. Note, however, that for this to work it is necessary to ensure that the memory regions `a[]` and `b[]` do not overlap. This is ensured if they are declared as arrays, but if they are pointers, to tell the compiler that the corresponding memory regions cannot overlap the pointer declarations have to be decorated with the `restrict` keyword.

[**Virtual memory**] The translation of virtual memory addresses to physical addresses is done in the following way (the translation is `|v_page_number|offset| -> |p_page_number|offset|`):

- The virtual memory address `v_addr` is split into two parts:

  $$\texttt{v\_addr} = \texttt{v\_page\_number} \times \texttt{page\_size} + \texttt{v\_offset}$$

  The page size of a power of two, so this amounts to split the address bits info two groups: the lower order bits specify the offset and the rest specify the page number

- The page number is used as an index to an array holding physical page numbers:

  $$\texttt{p\_addr} = \texttt{table[v\_page\_number]} \times \texttt{page\_size} + \texttt{v\_offset}$$

In practice, the `table[]` array is multidimensional (i.e., the page number bits as further subdivided and used as indices to access a multidimensional array. This saves memory. (page tables.)

Besides storing the physical page number each entry of the (page) table also stores bits that specify (more details):

- if the page is present in physical memory,
- if it can be written to,
- if it dirty,
- if code can be executed from it, and
- other information.

All this makes possible sharing physical pages among processes (examples: shared memory and dynamic libraries). Also, part of the memory used by the program may not be resident (stores in a swap partition).

The virtual to physical address translation is cached (TLB).

October 07, 2024

[**Data caches**] Each data cache holds a fixed amount of information. Each (physical) address used to access a cache is split into three parts (`|tag|set|offset|`, more info):

- an offset (the 6 least significant bits on most current processors),
- a set number (the next few bits), and
- a tag number (the rest of the bits).

For each set number, the cache can hold information about n different tags (n-way set associative cache). We have a cache hit when the tag part of the address matches one of the tags stored in the set specified by the set number part of the address.

For example, for an 32KiB 8-way set associative cache with cache lines holding 64 bytes (the offset part of the address has 6 bits), we have $32768/64/8i = 64$ sets, i.e., 6 bits are also used to specify the set number. A 64-bit address will then be split into a 52-bit tab, a 6-bit set number, and a 6-bit offset (`|52|6|6|`). This means that the following code will generate a lot of cache misses:

```
double a[512][512],b[512];
for(int i = 0;i < 512;i++)
{
  double sum = 0.0;
  for(int j = 0;j < 512;j++)
    sum += a[j][i]; // access &a[0[0]+512*j+i
  b[i] = sum;
}
```

But doubles have 8 bytes, and so the actual addresses used in the inner loop are `(void *)a+4096*j+8*i`. When j increases the address increases by $4096 = 2^{12}$, and that means that all addresses of the inner loop hit the same set number. If the cache is, say, 8-way set associative, when j reached 8, the data for $j = 0$ has to be evicted from the cache, and so it will have to be reloaded for the next value of i. (It is even worst: we are loading 64 bytes — a cache line — but we will only be using 8, what a waste!).

To solve this problem, we can to 8 iterations of the inner loop at the same time:

```
double a[512][512],b[512];
for(int i = 0;i < 512;i += 8
{
  double sum0 = 0.0;
  ...
  double sum7 = 0.0;
  for(int j = 0;j < 512;j++)
  {
    sum0 += a[j][i + 0]; // &a[0[0]+512*j+(i+0)
    ...
    sum7 += a[j][i + 7]; // &a[0[0]+512*j+(i+7)
  }
  b[i + 0] = sum0;
  ...
  b[i + 7] = sum7;
}
```

Assuming that `a[]` starts in an address that is a multiple of 64, in the inner loop we will be accessing all 64-bytes of the **same** cache line. No waste. Things are even better, because now it becomes possible to use SIMD instructions (the following works for a recent Intel/AMD processor with AVX512 instructions): all 8 doubles can be loaded into a processer register in one instruction, the 8 additions can also be done in parallel using only one instruction, and the 8 sums (stored in a single 512-bit SIMD register) can also be stored using only one instruction!

Finally, the data prefetcher unit finds that memory is being accessed with a stride of 4096 and will start prefetching data before it is actually referenced by the program.

[**Instruction caches and branch prediction**] Instruction caches are organized in almost the same was as a data cache. Sometimes, they store already decoded instructions (op cache). They also store branch prediction data (for example, for each conditional branch, recent taken/not taken history). The branch predictor used this information to attempt to predict if a conditional branch will be taken or not. If it succeeds, the next instruction that will be fetched is the right one, so things will proceed at full pace. It is fails, the pipeline has to be partially flushed, and that incurs a significant performance penalty (say, 10 clock cycles; note that due to the superscalar nature of current processors, 10 clock cycles may mean 30 or 40 instructions...).

October 14, 2024

[**SIMD, Single Instruction, Multiple Data**] SIMD registers are wider registers (more than 64-bits), than can be subdivided into smaller parts (see these slides for a nice overview of SIMD stuff). For example, a 256-bit register can be subdivided into 32 8-bit parts, 16 16-bit parts, 8 32-bit parts, and 4 64-bit parts. Arithmetic and logical instructions operate in parallel on each part. We will call to each part of the register a lane. Intel/AMD SSE, AVX, and AVX2 instructions act unconditionally on each lane. AVX512 instructions can use a so-called mask register to control which lanes are active. This is similar to what CUDA does.

We you experienced in the first practical assignment, to take advantage of SIMD instructions will will be usually necessary to change the data layout of the information. For example, to search for 4 DETI coins in parallel, the data layout had to be changed from

$$\underbrace{\begin{matrix} a[0] & a[1] & \cdots & a[12] \\ b[0] & b[1] & \cdots & b[12] \\ c[0] & c[1] & \cdots & c[12] \\ d[0] & d[1] & \cdots & d[12] \end{matrix}}_{\text{consecutive}} \quad \text{to} \quad \underbrace{\begin{matrix} a[0] & b[0] & c[0] & d[0] \\ a[1] & b[1] & c[1] & d[1] \\ \vdots & \vdots & \vdots & \vdots \\ a[12] & b[12] & c[12] & d[12] \end{matrix}}_{\text{interleaved}}$$

Addresses grow from the left to the right, and then from top to bottom.

October 21, 2024

Nothing relevant.

October 28, 2024

[**CUDA**]
See `CUDA_ARB.pdf`.

November 04, 2024

Nothing relevant.

November 11, 2024

[**Performance metrics**] Benchmarks measure how fast a computer is. There exist standard benchmarks for the CPU (SPEC), memory, Input/Output, etc. Supercomputers also have their specific benchmarks (TOP 500 and GREEN 500). See also phoronix.

A benchmark suite is composed of several programs. How to combine their execution times? Using the arithmetic mean is a very idea, because the programs with the largest execution times will dominate the performance metric. For example, on one machine two programs have execution times $t_1 = 99$ and $t_2 = 1$; their arithmetic mean is 50. On a second machine the same two programs have execution times $t'_1 = 98$ and $t'_2 = 2$; their arithmetic mean is also 50. However, the execution times of the first program are almost the same (98 versus 99), while for the second one is twice as fast on one machine. Saying that they have similar performance is not right. One way to address this shortcoming of the arithmetic mean is to replace it by the geometric mean. In the hypothetical example given above, for the first machine we get $\sqrt{99} \approx 10$, and for the second we get $\sqrt{196} = 14$, which is much more realistic.

[**Amdahl's law**] If you have a program that has a part which cannot be run in parallel i(execution time t_1) and another part that can (execution time t_2), Amdahl's law states how much the program can be speed up if you run the parallel part using n processors. At best, you can replace an execution time of $t_1 + t_2$ if you run it sequentially, by $t_1 + \frac{t_2}{n}$ if you run the parallelizable part in the n processors with zero inter-process communication cost.

November 18, 2024

Nothing relevant.

November 25, 2024

[**Adder**] To sum two 2^b integers, a ripple-carry adder connects sequentially 2^b full-adders (each sums three bits: two summands and one carry in bit, and produces a 1-bit sum and a carry out bit). The worst propagation delay is proportional to 2^b, and that is too high. Carry-lookahead circuits can be used to reduce substantially the propagation delay. Another possibility is to split the vector in, say, half, and to duplicate the

summation circuits for the upper half (one with a carry in of 0, and another with a carry in of 1); a multiplexor can then be used to select the appropriate summation (that depends of the carry out signal of the first half). This almost halves the worst propagation delay.

To subtract, one-complement and use a carry in of 1 (i.e., two complement!). This can be done efficiently using exclusive or gates.

[**Barrel shifter**] One simple entity receives as input a (std logic) vector, and a signal that indicates if the output vector is the input vector unmodified, or if it is shifted right by a fixed number of bits (specified as a generic parameter).

To build an efficient barrel shifter, just concatenate several of the entities briefly described in the previous paragraph, one that shifts of not by one bit (use the least significant bit of the shift amount to control it), another that shifts of not by two bits (use the next least significant bit of the shift amount to control it), and so on. To shift a vector with 2^b bits, b stages are needed. The worst propagation delay is proportional to b.

December 02, 2024

Nothing relevant.

December 09, 2024

Nothing relevant.

December 16, 2024

Possible question subjects for the final exam (there may be others, but most of the topics below will appear in the exam):

- Virtual to physical address translation. Study: process isolation, memory area protections (no execute, read only, ...), sharing memory between processes (including dynamic libraries), address translation cache (TLB).
- Data and instruction caches. Study: cache organization (number of ways), associativity problems, op cache (of already decoded instructions), what happens in the data caches when we multiply two 1024×1024 matrices (ways to solve this, which open the way to further optimizations, like using SIMD instructions).
- Software pipelining. Study it.
- Data prefetch. Study: what is it good for.
- Superscalar CPU architecture. Study: what is it (fetch more than one instruction in the same clock cycle, pipeline with several execution units, ...), advantages.
- Out of order execution. Study: what is it, speculative execution, out of order execution (instructions with different latencies, instructions that do not depend on the results of previous instructions, dependency chain).
- Single Instruction Multiple Data (SIMD). Study: what is it, what is it good for.
- Simultaneous MultiThreading (hyperthreading). Study: what is it (same pipeline, duplication of the register file), advantages (sharing of execution units across SMT threads makes possible using them more fully), and disadvantages (the caches are shared).
- Many cores. Study: OpenMP (pragma parallel, critical, reduce)
- Arithmetic versus geometric mean in a benchmark. Exercise about Amdahl's law.
- CUDA (one or more questions). Study: general architecture of a CUDA pipeline (warp, 32-way SIMD instructions), good and bad memory access patterns (good: the threads of the same warp access the same memory location, or memory locations with consecutive addresses; bad: addresses are not consecutive), thread divergence (different threads of the same warp attempt to execute different

code; study how this is implemented). Kernel launching grids (what are they, why are they needed).

- Question about the first assignment.
- Question about the second assignment (probably, how can a barrel shifter be implemented in an efficient way, worst propagation delay as a function of the number of bits of the vector).)