Introduction to High Performance Computing

April 8, 2018

1 Syllabus

This calendar is subject to change. Anything posted in the future is certainly tentative. Also, anything posted in the recent past may not be correct if I haven't updated it yet.

1.1 Week 1

1. **Von Neumann Architecture (VNA)** Tue April 17, 2018

- Using the ULM (Ulm's Lecture Machine) as reference model.
- Components: RAM, Data bus, CPU.
- Representation of unsigned integer numbers as decimal, binary and hexadecimal numerals.
- Instruction set (Machine code) of the ULM.
- Von Neumann Cycle: How machine code gets executed.

2. **VNA: Conditional Jumps and Integer Arithmetic** Fri April 20, 2018

- Assembly language for the ULM.
- Connection to C:
  - pointers and (referenced) values.
  - variable types.
- Signed integer (two's complement).
- Arithmetic logical unit (ALU) and its status flags.
- Conditional jumps in machine/assembly code

1.2 Week 2

3. **VNA: Control Structures** Tue April 24, 2018

- Macros for assembly code
- Example: Stack
- Control structures (if-then-else, for-loops, while-loops)

4. **VNA: Function Calls** Fr April 27, 2018

- Functions and procedures:
  - Passing parameters (call by value, call by reference).
  - Returning from call.
  - Local and global variables.
- Memory layout of a process: Text segment, data segment, stack and heap.
- Layout of a program file.
- Concept of a linker.

1.3 Week 3

5. **Holiday** Tue May 1, 2018

Time to work on the more intense quiz from last Friday.

6. **Introduction to C (C99)** Fr May 4, 2018

- How to describe the syntax of a programming language
- Tools for creating a executable program:
  - Preprocessor
  - Tokenizer
  - Parser
  - Assembler
  - Linker
- Differences and similarities between ULM and actual computers:
  - Virtual memory
  - Processes
  - Program loader

1.4 Week 4

7. **C99: Control Statements and Functions** Tue May 8, 2018

- Types and variables
- C control statements: if-then statements and loops.
- Special variable types: Pointers
- Functions

8. **C99: Arrays, Memory Allocation and Benchmarks** Fr May 11, 2018

- Dynamic memory allocation
- BLAS (Basic Linear Algebra Subprogram) Level 1 operations (as application for arrays).
- Timer for benchmarks
- Gnuplot

1.5 Week 5

9. **C99: Representing matrices** Tue May 15, 2018

- Full storage format for matrices (and vectors).
- Introducing some BLAS Level 2 Operations.
- Notes on testing and benchmarking BLAS operations:
  - General form of a benchmark (that also tests correctness).
  - Test cases with random numbers and NaNs
  - Estimating the error
  - Controlling test parameters through macros

10. **CPU Caches** Fr May 18, 2018

- Concept of CPU caches:
  - Cache line, cache miss, cache hit.
  - Cache associativity.
- Prefetching

1.6 Week 6

11. **CPU Caches: Optimized matrix vector product** Tue May 22, 2018

- Mathematical notation for blocked matrix operations.
- Simple optimization for row or column major storage.
- Advanced optimization using fused vector operations.

12. **Outlook: Generic and object oriented programming** Fr May 25, 2018

- Namespaces in C99 and C++.
- Generic programming (of functions):
  - In C99 with macros.
  - In C++ with template functions.
- Object oriented programming (OOP)
  - In C99 with *struct* and function pointers.
  - In C++ with *struct* (or *class*) and builtin language features (RAII, methods, operators).
- Combining generic programming and OOP.
- Typical pitfalls (focus on examples from generic programming).

1.7 Week 7

13. **General Matrix-Matrix Product (GEMM)** Tue May 29, 2018

- Definition of the GEMM operation.
- Potential for high performance.
- Relevance for other numerical linear algebra operations.
- Some obvious ways to implement the GEMM operation relatively efficient.
- Test and benchmark suite for the GEMM operation.

14. **GEMM: Simple Cache Optimization.** Fr June 1, 2018

- Benchmark comparing our GEMM with the Intel MKL implementation.
- Blocked and buffered computation.

1.8 Week 8

15. **GEMM: Advanced Cache Optimization** Tue June 5, 2018

- Exploiting the complete cache hierarchy:
  - Frame function and macro/micro kernel for GEMM.
  - Packing matrix blocks.
- Testing components of the overall algorithm.

16. **GEMM: Advanced Cache Optimization** Fr June 8, 2018

- Using a profiler: identify code spots where performance gets lost.
- Instruction pipeline optimizations: Enable compilers to unroll loops.
- Enable compilers to inline functions.

1.9 Week 9

17. **GEMM: Optimized Micro Kernel** Tue June 12, 2018

- Single Instruction Multiple Data (SIMD).
- GEMM micro kernel in assembly code exploiting SIMD processor features.

18. **GEMM: Optimized Micro Kernel** Fr June 15, 2018

- Manual loop unrolling in the assembly micro kernels.
- Manual instruction pipeline optimizations.
- Manual cache prefetching.

1.10 Week 10

19. **Building libraries (ulmBLAS)** Tue June 19, 2018

- Static and dynamic libraries.
- Creating and working with static libraries:
  - Split a project in different compile units (and provide header files).
  - Identify and resolve linker problems.
  - Awareness of issues with inline functions.
- Concept of *functional programming* in Makefiles.
- Special features of GNU make:
  - Text functions.
  - Recursions.

20. **ulmBLAS: Providing a Fortran interface** Fr June 22, 2018

- Short introduction of Fortran.
- Calling Fortran functions from C.
- Calling C functions from Fortran.

1.11 Week 11

21. LU Factorization (LU) — Tue June 26, 2018

- Mathematical background: LU factorization with and without pivoting.
- Using the factorization to solve linear matrix equations.
- LU factorization of non-square matrices.
- Exercise: Setup a test and benchmark suite.

22. LU: Unblocked — Fr June 29, 2018

- Deriving unblocked variants (i.e. based on BLAS level 2 operations) of the LU factorization.

1.12 Week 12

23. LU: Blocked — Tue July 3, 2018

- Deriving blocked variants (i.e. based on BLAS level 3 operations and a unblocked variant) of the LU factorization.

24. Triangular Solvers (TRS) — Fr July 6, 2018

1.13 Week 13

25. TRS: Unblocked — Tue July 10, 2018

26. TRS: Blocked — Fr July 13, 2018

1.14 Week 14

27. TRS: Blocked — Tue July 17, 2018

28. Final Benchmarks and Outlook on Multithreading — Fr July 20, 2018

2 Von Neumann Architecture (VNA)

The historical *Von Neumann architecture (VNA)* can be regarded as a model that describes how nowadays computers work in principle. It consists of four main components, namely a memory, some input/output devices, the central processing unit (CPU) and a data bus. We will use the fictional *Ulm Lecture Machine (ULM)* as an example of a VNA for the following reasons:

- In the computer lab we will work with the Intel64 architecture. The assembly language for the ULM has many similarities with the assembly language that will be used later for the Intel64 architecture.
- A UNIX process has many similarities with a program running on the ULM

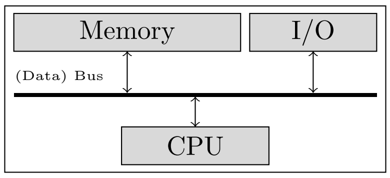

Figure 1: Components of the ULM (Ulm Lecture Machine) which we use as a reference model for the Von Neumann Architecture.

2.1 Integers

Computers are programmed through numbers and work with numbers. These numbers are represented as binary numerals. Hence, for truly understanding how a computer works we need a good understanding of this matter.

2.1.1 Unsigned Integers

Definition 2.1 (Positional Numeral Notation)
For constants $b, n \in \mathbb{N}_0$ with $b > 1$ and values $a_0, \dots, a_{n-1} \in \{0, \dots, b-1\}$ we define

$$(a_{n-1}, \dots, a_0)_b := a_{n-1}b^{n-1} + \dots + a_1 b + a_0 = \sum_{k=0}^{n} a_k b^k. \tag{1}$$

We will denote b as the *base* or *radix*, values $a_0, \dots, a_{n-1}$ as the digits and $(a_{n-1}, \dots, a_0)_b$ as the *(positional) numeral notation* of the number defined by (1).

Often we do not explicitly distinguish between the terms *number* and *numeral*. However, this is formally not correct. Numbers are elements of some algebraic structures like $\mathbb{N}_0$ and numerals are labels for these elements. The set $\mathbb{N}_0$ for example can be characterized through the Peano axioms.

Numerals are just names for the elements of an algebraic structure. So for instance, the different numerals ‘null’, ‘zero’ or ‘0’ can be used to refer to the same number in $\mathbb{N}_0$. From the postulated axioms it follows that each element has a successor that is different from all its predecessors. Commonly we denote with 1 (or ‘one’) the successor or 0, with 2 (or ‘two’) the successor of 1, etc.

A *digit* is a number for which we will use a numeral that consists of a single symbol. If b is not larger than ten we use the decimal numerals $0, 1, \dots, 9$ to represent a digit. Otherwise we extend the set of symbols, like in the following example for hexadecimal numerals.

Example 2.2 (Further Conventions for Numeral Notation)

a) (Decimal Numerals)

$$10 := (1, 0)_{10} = 1 \cdot 10^1 + 0 \cdot 10^0$$

$$123 := (1, 2, 3)_{10} = 1 \cdot 10^2 + 2 \cdot 10^1 + 3 \cdot 10^0$$

b) (Binary Numerals)

   Also will use the following terms:

   i. With *bit* we denote a binary digit and
   ii. with *bit pattern* the list of bits in a binary numeral.
   iii. Furthermore, a *byte* is a bit pattern consisting of eight bits. A *word* refers to two bytes, a *long word* to four bytes and a *quad word* to eight bytes.

c) (Hexadecimal Numerals)

Lemma 2.3
Numbers representable through (1) as numerals with n digits and base b, i. e.

$$M = \left\{(a_{n-1}, \dots, a_0)_b \,:\, a_0, \dots, a_{n-1} \in \{0, \dots, b-1\}\right\}$$

are non-negative integers and more precise

$$M = [0, \dots, b^n - 1) \subset \mathbb{Z}$$

Proof. The inclusion $M \subseteq [0, \dots, b^{n-1})$ follows from

$$0 \leq (0, \dots, 0)_b = (a_{n-1}, \dots, a_0)_b \leq (b-1, \dots, b-1)_b = \sum_{k=0}^{n-1} (b-1)\, b^k = b^n - 1.$$

Next we show that also $M \supseteq [0, \dots, b^n - 1)$ holds. So let $z \in [0, \dots, b^n - 1)$ be fixed but arbitrary:

$\underline{n = 1}$: Because of $0 \leq z < b$ we can choose $a_0 := z$ and get $z = (a_0)_b \in M$.

$\underline{n \rightsquigarrow n+1}$: For $z \in [0, \dots, b^n - 1)$ we can find integers k, r with $k \geq 0$ and $0 \leq r < b^{n-1}$ such that $z = k \cdot b^n + r$. From the induction hypothesis follows a representation $k = (a_{n-1}, \dots, a_0)_b$ and choosing $a_n = k$ gives $z = kb^n + r = a_n b^n + (a_{n-1}, \dots, a_0)_b = (a_n, a_{n-1}, \dots, a_0)_b \in M$.

□

Obviously, one gets k and r through Euclidean division (i. e. integer division with quotient and reminder)

$$a_k := \left\lfloor \frac{z}{b^k} \right\rfloor - b \left\lfloor \frac{z}{b^{k+1}} \right\rfloor \quad (0 \leq k < n)$$

This motivates the following definition.

Definition 2.4 (Range of an n digit unsigned Integer)
With $\mathbb{U}_{n,b} := \{0, \dots, b^n - 1\}$ we denote the range of unsigned integers that can be represented with an n digit numeral for base b. As binary numerals are of particular interest to us, we define

$$\mathbb{U}_n := \{0, \dots, 2^n - 1\}$$

for the range of an n bit unsigned integer.

Example 2.5
On computers numbers are represented through a certain number of bytes. On the ULM (and basically any computer in the real world) the number of bytes is a power of two). As each byte consists of eight bits the following ranges are foremost relevant:

a) Unsigned integer with 1 byte

   $\mathbb{U}_8 = \{0; \dots; 255\}$

b) Unsigned integer with 2 bytes (also denoted as *1 word*)

   $\mathbb{U}_{16} = \{0; \dots; 65{,}536\}$

c) Unsigend integer with 4 bytes (also denoted as *1 long word*)
$\mathbb{U}_{32} = \{0; \ldots; 4{,}294{,}967{,}295\}$

d) Unsigned integer with 8 bytes (also denoted as *1 quad word*)
$\mathbb{U}_{64} = \{0; \ldots; 18{,}446{,}744{,}073{,}709{,}551{,}615\}$

Example 2.6 (Convertion of numerals to different base)

$$\begin{aligned}
2^0 &= 1\\
2^1 &= 2\\
2^2 &= 4\\
2^3 &= 8\\
2^4 &= 16\\
2^5 &= 32\\
2^6 &= 64\\
2^7 &= 128\\
2^8 &= 256\\
2^9 &= 512\\
2^{10} &= 1024\\
2^{11} &= 2048\\
2^{12} &= 4096\\
2^{13} &= 8192\\
2^{14} &= 16{,}384\\
2^{15} &= 32{,}768\\
2^{16} &= 65{,}536
\end{aligned}$$

$$\begin{aligned}
16^0 &= 1\\
16^1 &= 16\\
16^2 &= 256\\
16^3 &= 4096\\
16^4 &= 65{,}536
\end{aligned}$$

Decimal	Hexadecimal	Binary
0	0	0000
1	1	0001
2	2	0010
3	3	0011
4	4	0100
5	5	0101
6	6	0110
7	7	0111
8	8	1000
9	9	1001
10	A	1010
11	B	1011
12	C	1100
13	D	1101
14	E	1110
15	F	1111

1. (Conversion between decimal and binary)

2. (Conversion between decimal and hexadecimal)

3. (Conversion between binary and hexadecimal)

Lemma 2.7 (Shift operations on unsigend Integers) 1. (Left shift) $x \cdot b = (a_{n-1}, \ldots, a_0, 0)_b$.

2. (Right shift) $\left\lfloor \frac{x}{b} \right\rfloor = (a_{n-1}, \dots, a_1)_b$.
3. (Rounding downwards to powers of b) $y = (a_{n-1}, \dots, a_s, 0, \dots, 0)$ is dividable by b^s, $y \leq x$ and furthermore $y = \max\{u \in \mathbb{U}_n \,:\, u \leq x\}$.

a)

$$\begin{array}{cccc} & 1 & 0 & 0 \\ + & 0 & 1 & 1 \\ \hline & 1 & 1 & 1 \end{array}$$

b)

$$\begin{array}{cccc} & 0 & 0 & 1 \\ + & 0 & 1 & 1 \\ 0 & {\scriptstyle 1} & {\scriptstyle 1} & {\scriptstyle 0} \\ \hline & 0 & 0 & 1 \end{array}$$

c)

$$\begin{array}{cccc} & 0 & 0 & 1 \\ + & 0 & 1 & 1 \\ \hline \text{CF} & 0 & 1 & 0 \\ \hline & 1 & 0 & 0 \end{array}$$

d) $\begin{array}{cccc} & 1 & 0 & 0 \\ + & 0 & 1 & 1 \\ \hline & 1 & 1 & 1 \end{array}$

e) $\begin{array}{cccc} & 0 & 0 & 1 \\ + & 0 & 1 & 1 \\ \hline & 1 & 0 & 0 \end{array}$

2.1.2 Signed Integers

2.1.3 Integer Arithmetic