Workshop: School of Data Science and Computation Thinking, Stellenbosch University

Introduction to Parallel Computation with GPUs: Programming with JAX, NUMBA, THURST, and CUDA

Martin BUCHER
(bucher@sun.ac.za)

Course dates (4-5pm SAST):

Tu 18 Feb	Th 20 Feb
Tu 25 Feb	Th 27 Feb
Tu 4 March	Th 6 March
Tu 11 March	Th 13 March

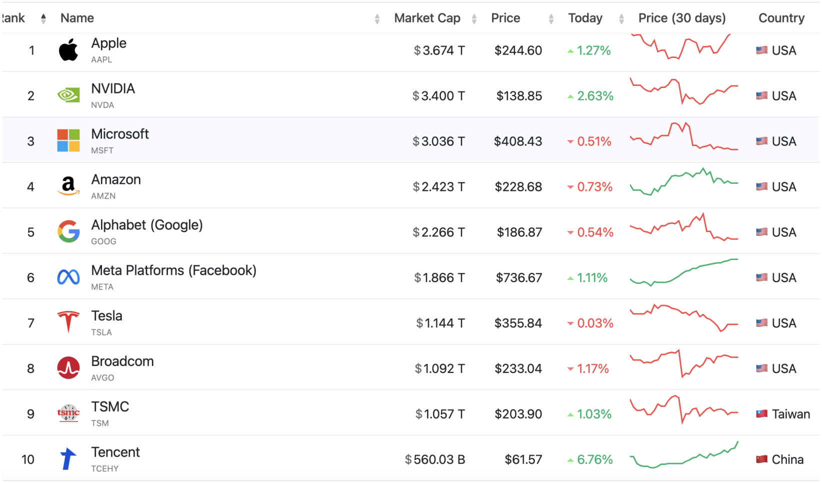
Zoom link:

`https://u-paris.zoom.us/j/82740807191?pwd=8yAGlP3R6VKbsjNCIeRjJU3u0AQtU6.1`

Course webpage:

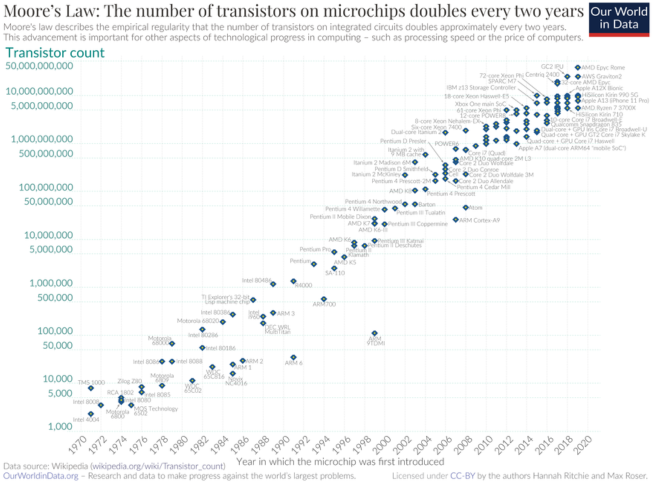
`http://www.sun.ac.za/english/data-science-and-computational-thinking/parallelcomputation`

Largest Tech Companies (18 Februray 2025



ank	Name	Market Cap	Price	Today	Price (30 days)	Country
1	Apple AAPL	$3.674 T	$244.60	▲1.27%		USA
2	NVIDIA NVDA	$3.400 T	$138.85	▲2.63%		USA
3	Microsoft MSFT	$3.036 T	$408.43	▼0.51%		USA
4	Amazon AMZN	$2.423 T	$228.68	▼0.73%		USA
5	Alphabet (Google) GOOG	$2.266 T	$186.87	▼0.54%		USA
6	Meta Platforms (Facebook) META	$1.866 T	$736.67	▲1.11%		USA
7	Tesla TSLA	$1.144 T	$355.84	▼0.03%		USA
8	Broadcom AVGO	$1.092 T	$233.04	▼1.17%		USA
9	TSMC TSM	$1.057 T	$203.90	▲1.03%		Taiwan
10	Tencent TCEHY	$560.03 B	$61.57	▲6.76%		China

Gordon Moore (1965)


Moore's Law: The number of transistors on microchips doubles every two years

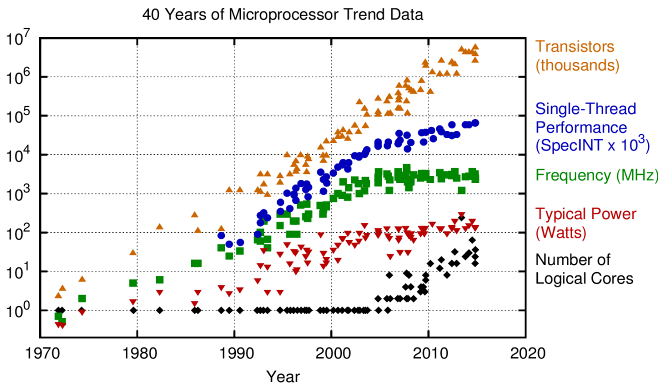
Moore's law describes the empirical regularity that the number of transistors on integrated circuits doubles approximately every two years. This advancement is important for other aspects of technological progress in computing – such as processing speed or the price of computers.

Data source: Wikipedia (wikipedia.org/wiki/Transistor_count)

OurWorldinData.org – Research and data to make progress against the world's largest problems.

Licensed under CC-BY by the authors Hannah Ritchie and Max Roser.

Trends in Single-Thread CPU Performance



Original data up to the year 2010 collected and plotted by M. Horowitz, F. Labonte, O. Shacham, K. Olukotun, L. Hammond, and C. Batten
New plot and data collected for 2010-2015 by K. Rupp

On Parallel Computing: "There is no alternative."

C/C++ Is Much Faster Than Python (or Julia)

We compare a tree program, where no Python libraries (such a numpy) can be called to accelerate critical code sections, and one is forced to program in native Python (or Julia).

Here are the comparisons:

41.1 sec Python
21.1 sec julia
1.459 sec C/C++ -g compilation
0.912 sec C/C++ -O3 compilation

Why?

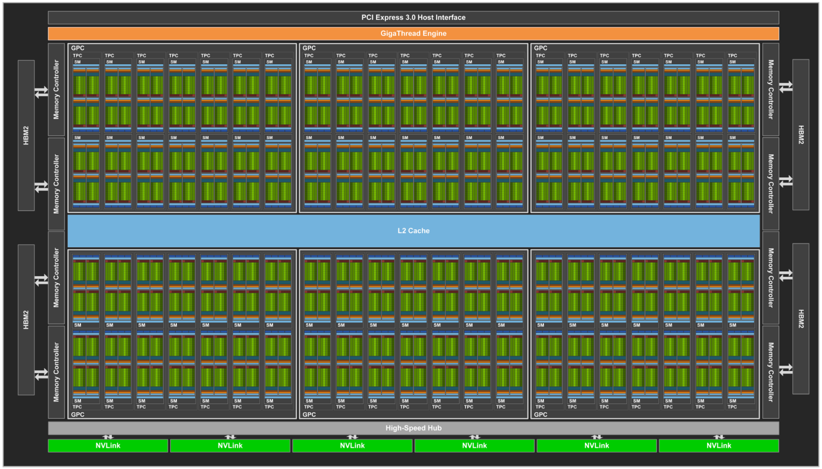
1. C/C++ is much closer to the assembly/machine code.
2. Compiler can be very clever. Interpreters must execute instruction by instruction.

Differences between Python and C/C++/CUDA

- Python is a "dynamically typed" language.
  - Types of variables are known only when the program is run.
  - No need for wordy type definitions, but this comes at a cost.
  - Few mistakes caught at compilation. More testing with real data required.
  - Sometimes it is hard to read programs because type information is missing.
  - Variables (or "objects") carry additional type information around, which slows down execution.
- C/C++/CUDA are "strongly typed" and "statically typed" languages.
  - All types are known at compilation. Allows for many mistakes to be caught before execution is attempted.
  - No type checking when the program is running, which leads to faster code and allows for more optimizations.

Parallel Computing Began as Super-Expensive Niche Hardware

Cray-1 (1975) US$7.9 million in 1977 (equivalent to $38.2 million in 2022) 5.5 tons (Cray-1A) Power 115 kW Once upon a time world's fastest computer Pioneered vector processing Buzzword is SIMD - single instruction multiple data



Volta GV100 Full GPU with 84 SM (Streaming Multiprocessor) Units. 5376 FP32 cores, 5376 INT32 cores, 2688 FP64 cores, 672 Tensor Cores.

GPU Architecture

CPUs are **really smart**:

- Rearrangement of instruction stream, analysis of dependencies
- Branch prediction, speculative execution
- Pipeline execution of instructions, instruction level parallelism
- Microcode
- Multiple memory caches (for low latency)

GPUs are **simple** and **dumb**:

- Multiprocessing stream dispatching the same instructions to many processors,
- No pipe-lining and most of the above absent
- Simple caches and limited shared (local) memory

But there are many (**thousands**) dumb GPU processors on a card, so that full compute power (flops per second) and memory bandwidth far exceeds that of a CPU, so when they are all kept busy and access global GPU memory in a coherent way, they significantly outperform CPUs.

Using GPUs

- Can be easy or hard.
- For many problems (eg, linear algebra), one can use installed GPUs almost invisibly. ‘numpy’ functions, for example can compute using special libraries with GPUs on some setups. cuPy is a NumPy/SciPy-compatible Array Library for GPU-accelerated Computing. The cudaBLAS and MAGMA (LAPACK compliant) libraries implement linear algebra with GPUs painlessly.
- But for other problems parallelizing is less straightforward, and it is often at the outset unclear whether the problem is amenable to a parallelization that can outperform the CPU program.

CUDA: A language to turn GPU programming mainstream

- Graphics cards first developed as special purpose hardware for graphics rendering, etc. Some scientists figured out how to hack GPUs to do general purpose computations on hijacked graphics cards.
- Gamers and the entertainment industry have been the economic drivers for making massively parallel computing possible on low-cost commodity hardware.
- In 2007 Nvidia released the CUDA Programming Language extending C/C++ to enable general purpose hybrid CPU/GPU parallel programming through language extensions and special subroutines.
- Parallel programming is by nature harder because of synchronization issues, and need to avoid race conditions, deadlocks, livelocks, and the like.
- For genomics, most programming involves trees in one way or another. Tree algorithms have to some extent been studied. Tree insertion and traversal can in many cases be parallelized. Conflicts can be avoided using atomic operations in critical code sections, which avoid deadlocks by ensuring that at least one attempt to write will be accessible in each round. In many cases, data structures need to be re-adapted to the GPU architecture.
- Many open research questions remain.

GETTING STARTED

Baby C Hello World Program (from K&R)

```
(base) apcmc2227:NITHeCS mabucher$ cat hello_world.c
// Simple C program to display "Hello World"
#include <stdio.h> // Header file for input output functions

// main function - where the execution of program begins
int main()
{
    // prints hello world
    printf("Hello World\n");
    return 0;
}

  528  history
(base) apcmc2227:NITHeCS mabucher$ gcc -o hello_world hello_world.c
(base) apcmc2227:NITHeCS mabucher$ ./hello_world
Hello World
(base) apcmc2227:NITHeCS mabucher$ ls
hello_world hello_world.c hello_world.o
(base) apcmc2227:NITHeCS mabucher$ gcc -c hello_world.c
(base) apcmc2227:NITHeCS mabucher$ gcc -o hello_world hello_world.o
(base) apcmc2227:NITHeCS mabucher$ ./hello_world
Hello World
```

Some Useful References

https://en.wikipedia.org/wiki/The_C_Programming_Language

https://ia903407.us.archive.org/35/items
/the-ansi-c-programming-language-by-brian-w.-kernighan-dennis-m.-ritchie.org
/The%20ANSI%20C%20Programming%20Language%20by%20Brian%20W.%20Kernighan%2C%20Dennis%20M.%20Ritchie.pdf

https://www.cs.cmu.edu/~ab/15-123S09/lectures/Lecture%2001%20-%20Introduction%20to%20Unix%20and%20C.pdf

https://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html

https://docs.nvidia.com/cuda/cuda-runtime-api/
group__CUDART__DEVICE.html#group__CUDART__DEVICE_1g1bf9d625a931d657e08db2b4391170f0

https://docs.nvidia.com/cuda/index.html

The C++ Programming Language, 4th Edition 4th Edition
by Bjarne Stroustrup (Author)
https://www.amazon.com/C-Programming-Language-4th/dp/0321563840

Github CUDA Samples
https://github.com/NVIDIA/cuda-samples

Parallel Hello World Program (I)

```
bash-4.2$ cat cudaEnhancedHelloWorldTer.cu
#include <stdlib.h>
#include <stdio.h>

__global__ void helloWorldKernel();

int main(){

  cudaError_t err;

  printf("Calling kernel function on device.\n");
  helloWorldKernel<<<1,10>>>();
  printf("Returning from kernel function on device.\n");
  cudaDeviceSynchronize();
  err = cudaGetLastError();
  if ( err != cudaSuccess ){
     printf("CUDA Error: %s\n", cudaGetErrorString(err));
     exit(1); // return error code
  }

  return 0;
}

__global__ void helloWorldKernel(){
  for(int i=0;i<10;i++){
    if (threadIdx.x==i )
      printf("Hello world from device, block= %d, thread=%d \n", blockIdx.x, threadIdx.x);
    __syncthreads();
  }
}
```

Parallel Hello World Program (II)

```
bash-4.2$ nvcc -o cudaEnhancedHelloWorldTer -g -gencode arch=compute_70,code=sm_70
-gencode arch=compute_75,code=sm_75 cudaEnhancedHelloWorldTer.cu
bash-4.2$ ./cudaEnhancedHelloWorldTer
Calling kernel function on device.
Returning from kernel function on device.
Hello world from device, block= 0, thread=0
Hello world from device, block= 0, thread=1
Hello world from device, block= 0, thread=2
Hello world from device, block= 0, thread=3
Hello world from device, block= 0, thread=4
Hello world from device, block= 0, thread=5
Hello world from device, block= 0, thread=6
Hello world from device, block= 0, thread=7
Hello world from device, block= 0, thread=8
Hello world from device, block= 0, thread=9
```

Parallel Hello World Program (III)

- CUDA programs have a suffix .cu rather than .c (for C programs) or .C or .cpp for (C++). CUDA is an extension of a subset of the C/C++ language and must be compiled with `nvcc` rather than `gcc`.
- Every C program starts with a function called `main` which is called when the program is executed. This is a function run by the CPU, also known as the host.
- A function called from the host that runs on the GPU is declared using __global__ keyword, which is part of the CUDA extension of C/C++. These are run by the GPU, as known as the device. And is compiled separately and converted into CUDA assembly code. The assembly code must match the architecture and capabilities of GPU on which the executable is to be run. The nvcc compiler option indicates that code should be generated for the specified architecture. `-gencode arch=compute_70,code=sm_70` More than one architecture, compute capability pair may be specified, and in this case, several versions of device code are generated, the correct one being chosen at runtime.

Parallel Hello World Program (IV)

- A kernel program called from the host and run on the device is of the type void (no value is returned) is called using the following syntax, which is an extension of C/C++.

  ```
  kernelFunction<<<numberOfBlocks,threadsPerBlock>>>
  ```

  Here we have a single block of 10 threads:

  ```
  helloWorldKernel<<<1,10>>>();
  ```

  In this simplest example no data is passed between the host and device; hence there are no arguments.
- Here a one-dimensional **grid** or **threads** organized into **blocks** is launched. (In later lectures we shall explore two- and three-dimensional grids.) The kernel function call causes `numberOfBlocks*threadsPerBlock` semi-independent incarnations of the global function to be launched on the device, distinguished by the built-in variables `blockIdx.x` and `threadIdx.x`, taking the values `blockIdx.x=0..(nBlocks-1)` and `threadIdx.x=0..(nThreadsPerBlock)`.
- Each streaming multiprocessor runs a block at a time. The blocks are distributed between the available SMs, and threads are assigned to each of the cores associated with the SM.

In the kernel function code,

```
__global__ void helloWorldKernel(){
   for(int i=0;i<10;i++){
     if (threadIdx.x==i )
        printf("Hello world from device, block= %d, thread=%d \n",
                blockIdx.x, threadIdx.x);
     __syncthreads();
     }
}
```

care is taken so that processors write to the output stream in turn.
In each passage of the for loop, only one thread is allowed to print. Overlapping (parallel) writes would likely result in gibberish.
What syntactically looks like a function call (and is part of the CUDA extension of C/C++)

```
     __syncthreads();
```

creates a barrier that causes the threads in a block to wait until all the threads of the block have arrived at the barrier before proceeding.
Synchronisation of threads is a major concern for the correctness of parallel programs.
Synchronisation among blocks is more difficult.

Probing GPU Hardware and Error Handling

In C/C++ (unlike in interpreted languages such as Python) many errors are caught at compilation. C/C++ is strongly and statically typed, which means that all variables and functions must be declared at compilation, and this catches a large number of errors, although it may appear tedious and verbose.

Just because C/C++ code is syntactically correct and compiles does not necessarily imply that it will run successfully. Machines have limits that when exceeded will cause the program to crash. A CUDA program run with no available CUDA-enabled GPU may seem to run successfully but produce wrong answers. There are limits on the number of allowed, blocks, threads, etc.

For this reason, it is important to check the error statuses returned by CUDA functions and kernel call. While the code will be slightly longer and more cumbersome to write, it will be easier to debug.

Error handling for CUDA kernels

```
helloWorldKernel<<<1,10>>>();
printf("Returning from kernel function on device.\n");
cudaDeviceSynchronize();
err = cudaGetLastError();
if ( err != cudaSuccess ){
   printf("CUDA Error: %s\n", cudaGetErrorString(err));
   exit(0);
}
```

Compare with UNIX shell error handling

In UNIX shell programming the UNIX shell variable $? accesses the return status of the last command executed.

```
(base) apcmc2227:cudaCourseSlides mabucher$ cat success.c
/* success.c */
int main(){
  return 0;
}
(base) apcmc2227:cudaCourseSlides mabucher$ gcc success.c
(base) apcmc2227:cudaCourseSlides mabucher$ ./a.out
(base) apcmc2227:cudaCourseSlides mabucher$ echo $?
0
```

with

```
(base) apcmc2227:cudaCourseSlides mabucher$ cat failure.c
/* failure.c */
int main(){
  return 1;
}
(base) apcmc2227:cudaCourseSlides mabucher$ gcc failure.c
(base) apcmc2227:cudaCourseSlides mabucher$ ./a.out
(base) apcmc2227:cudaCourseSlides mabucher$ echo $?
1
```

Pointers, malloc, free and all that (I)

```
#include <stdlib.h>
#include <stdio.h>

int main(){
  float x=13.;  // x is a value, a real number represented using 4 bytes
  float *y;  // y is the address (in memory) of a number of the type float
  y=&x; //  now y points to x, whose value is 13. & is the address of operator
  printf("The value of x is %f.\n", x);
  printf("The value of pointed to by y  is %f.\n", *y);
  //  * = dereferencing operator. (value at the memory location pointed to)

  // printing pointers

  printf("The address of x is %p.\n", &x);
  printf("y points to the memory location %p.\n", y);
  return 0;
}
```

```
(base) apcmc2227:~ mabucher$ ./a.out
The value of x is 13.000000.
The value of pointed to by y  is 13.000000.
The address of x is 0x16fce7588.
y points to the memory location 0x16fce7588.
```

Pointers, malloc, free and all that (II)

```
#include <stdlib.h>
#include <stdio.h>

int main(){ //Pointer arithmetic

  float a[4]={0., 1., 2., 3.};
  // a is a pointer to a[0]
  //  *(a+2) is the same as a[2]
  printf("a[2] = %f \n", a[2] );
  printf("*(a+2) = %f \n", *(a+2));

  return 0;

}
```

```
(base) apcmc2227:cudaCourseSlides mabucher$ ./a.out
a[2] = 2.000000
*(a+2) = 2.000000
```

& is the "address of" operator. Maps variable indicating contents of a memory location $\rightarrow$ pointer (aka memory address).
* is the "derefercning" operator. Maps variable pointer to its contents.

Pointers, malloc, free and all that (III)

```
#include <stdlib.h>
#include <stdio.h>
int main(){  // Variable size arrays (length unknown as compile time)
   int len = 100;
   float *a= malloc(len*sizeof(float));
   // allocates number of bytes requested and returns pointer
   if ( a == NULL ){  // Error checking
     fprintf(stderr,"Memory allocation failed. Exiting.\n");
     exit(-1); // exit with error
   }
   for(int j=0;j<len;j++)
      *(a+j)=2.*j;
   for(int j=0;j<len;j++)
      printf("a[%d] = %f \n", j, a[j]);
   free(a);
   return 0;
}
(base) apcmc2227:cudaCourseSlides mabucher$ ./a.out
a[0] = 0.000000
a[1] = 2.000000
****************
a[99] = 198.000000
```

Java/Python philosophy: "Pointers are the root of all evil."
Memory leaks.

Defensive programming

```
#include <stdlib.h>
#include <stdio.h>

int main(){
  cudaError_t err;
  cudaDeviceProp prop;
  int device=0; int count; int major, minor;

  err=cudaGetDeviceCount(&count);
  if (err != cudaSuccess ){
    printf("Error: cudaGetDeviceCount call failed.\n");
    exit(0);
  }
  if (count==0){
    printf("Error: No CUDA enabled devices found.\n");
    exit(0);
  }
  ****
```

cuda.... functions return an integer error code (of the type cudaError) and these should be checked and errors should be handled.

Defensive programming without clutter

The C/C++ preprocessor and inline function (for example placed in a header file)

```
#define errCheck(command)       errCheck2((command),#command,__FILE__,__LINE__)

inline void errCheck2(int command, const char *commandString, const char *file,
    int value=command;
    if ( value != cudaSuccess ){
      printf("%s  in file %s at line %d \n", commandString, file, line);
      printf("Error: program aborting.\n");
      exit(0);
    }
}
```

allows us to wrap a CUDA function so that error checking with semi-informative error messages result using the following syntax

```
errCheck(cudaMalloc((void**) &a_d,DIM*sizeof(float)));
```

Querying Device Properties (I)

The following is a C structure defined in one of the CUDA header files:

```
struct cudaDeviceProp {
              char name[256];
              cudaUUID_t uuid;
              size_t totalGlobalMem;
              size_t sharedMemPerBlock;
              int regsPerBlock;
              int warpSize;
              size_t memPitch;
              int maxThreadsPerBlock;
              int maxThreadsDim[3];
              int maxGridSize[3];
              int clockRate;
******

}
```

Everything you would ever want to know about the GPU running on your computer is described by a member variable of this structure.

Querying Device Properties (II)

Snippet from the parallel hello world code where the device properties structure for device with number `best_device` is obtained.

```
 cudaDeviceProp prop;
 cudaError_t err;
 // *****
 err=cudaGetDeviceProperties (&prop, best_device);
 if (err != cudaSuccess){
   printf("Error: unable to probe device %d.\n", best_device);
   exit(-1);}
 // *****
int major=prop.major, minor=prop.minor; // get cuda version
 // *****
```

Passwordless login

```
(base) apcmc2227:cudaCourseSlides mabucher$ ssh sungpu
Last login: Tue Feb 18 12:25:56 2025 from 80.214.17.124
bucher@stoertebeker:~$ cat .ssh/authorized_keys

ecdsa-sha2-nistp256 AAAAE2VjZHNhLXNoYTItbmlzdHAyNTYAAAAIbmlzdHAyNTYAAABBBLyjaGC

bucher@stoertebeker:~$ exit
logout
Connection to stoertebeker.sun.ac.za closed.
(base) apcmc2227:cudaCourseSlides mabucher$ cat ~/.ssh/config

Host sungpu
        HostName stoertebeker.sun.ac.za
        User bucher
        IdentityFile /Users/mbucher_1/.ssh/ronaldo_ecdsa

(base) apcmc2227:~ mabucher$ ls .ssh
authorized_keys
config
ronaldo_ecdsa
ronaldo_ecdsa.pub
(base) apcmc2227:~ mabucher$ cat .ssh/ronaldo_ecdsa.pub
ecdsa-sha2-nistp256 AAAAE2VjZHNhLXNoYTItbmlzdHAyNTYAAAAIbmlzdHAyNTYAAABBBLyjaGC
```

Generating a Public/Private Key Pair

```
(base) apcmc2227:~ mabucher$ ssh-keygen -t ecdsa
Generating public/private ecdsa key pair.
Enter file in which to save the key (/Users/mbucher_1/.ssh/id_ecdsa): klaus
Enter passphrase (empty for no passphrase):
Enter same passphrase again:
Your identification has been saved in klaus
Your public key has been saved in klaus.pub
The key fingerprint is:
SHA256:zscescaUG68TLTVRW2OQrksTUrzogakwsmvQkw/LrPI mabucher@apcmc2227.local
The key's randomart image is:
+---[ECDSA 256]---+
|          .  o+o.|
|           o...o.|
|.o.=    o Xo+    |
|.o+ .    +.+     |
|+oE       o.     |
+----[SHA256]-----+
(base) apcmc2227:~ mabucher$ ls -tr1
klaus_key
klaus_key.pub
```

`klaus_key` contains the private key; `klaus_key.pub` contains the public key. These files are placed in your .ssh hidden subdirectory of the main directory.

Some good references:

- John L. Hennesy & David Patterson, Computer Architecture: A Quantitative Approach, Sixth Edition. M K Morgan 2019
- Wen-mei Hwo, David Kirk and Izzat El Hajj, Programming Massively Parallel Processors: A Hands-On Approach, Fourth Edition. M K Morgan 2023
- Maurice Herlihy, Victor Luchangco, Nir Shavit and Michael Spear, The Art of Multiprocessor Programming. Second Edition. M K Morgan 2021

PLEASE COPY THIS LINK TO THE COURSE WEBPAGE WHERE COURSE MATERIALS WILL BE MADE AVAILABLE AS WE GO ALONG

`http://www.sun.ac.za/english/data-science-and-computational-thinking/parallelcomputation`