High Performance Parallel Systems

12 – Computing with accelerators

Kenneth Skovhede & Carl-Johannes Johnsen

What is an accelerator?

- Accelerators are a type of hardware, which is specialized for one particular type of task. In other words, it can accelerate a task.
- They are usually external components, which are connected to a host system. In most cases, the host is a generic CPU/RAM model.
- Due to their specialization, they usually have lower performance than general-purpose hardware for tasks, which do not fit their pattern.

Why use accelerators?

- The use of the Von-Neumann architecture introduces a bottleneck
  - Instructions need to be fetched over same memory channel
  - Only one instruction at a time
- Specialized application
  - Applications that rely heavily on a particular feature, say matrix multiplication, can increase performance if there is special hardware for this operation, instead of using a sequence of CPU instructions
- No need for general-purpose functionality
  - If the application does not need the flexibility of the CPU, those unused elements of the CPU are essentially wasted

Examples of accelerators

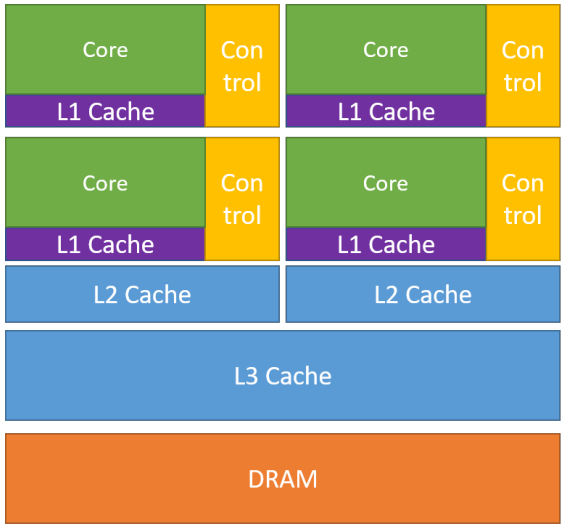
- Graphics card
- Bitcoin miner
- TPU
- TCP offload engine
- RAID controller
- Cryptography (AES)
- Video encode/decode (h.264)

Graphics card

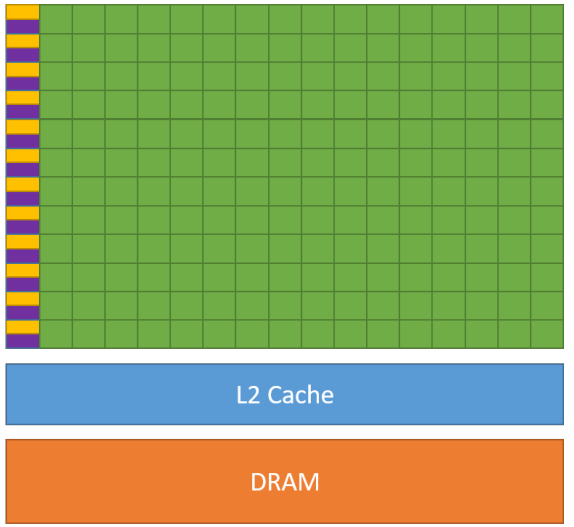
- They were introduced in the 80’s, as graphics became more and more popular for computers.
- They supported a wide range of linear algebra methods, as this is what makes up computer graphics.
- They discovered that these computations could be made in parallel and that they didn’t need to be flexible in the types of computations.
- Due to the memory access pattern of graphics processing usually being sequential, they can achieve a better memory bandwidth, compared to the CPU RAM.

Graphics card – The good

- As they grew more and more popular, graphics cards became faster and cheaper.
- Their perfomance increased by leveraging the parallelism through a lot of computation cores, or shaders, and by increasing memory bandwidth.
- All these hundreds, and even thousands, of cores run in lock-step. This means that every core runs the exact same program at the same time.
- They can be more energy efficient, compared to CPUs, due to more of the circuitry being dedicated for compute, rather than branch prediction, random access, etc.



CPU



GPU

Graphics card – The bad

- They are not good at single/lightly threaded programs.
- They are not good at random access memory patterns.
- They are not good at small problems, as there is an overhead when using external components.
- They are not good at branching.
- They are not good at running multiple different programs.
- They are not as easy as CPUs for getting ”quick” results.

OpenCL / CUDA

- As graphics card evolved, they started to have quite powerful computing capabilities.
- Traditionally when using a graphics card, one had to program them using graphical operations. Generally, this was done through shaders, where you specify how to draw and color graphics.
- This wasn’t a great fit for many general-purpose programs, as this required a substantial amount of work.
- To solve this, two frameworks were introduced: OpenCL by the Khronos group and CUDA by NVIDIA.

OpenCL / CUDA

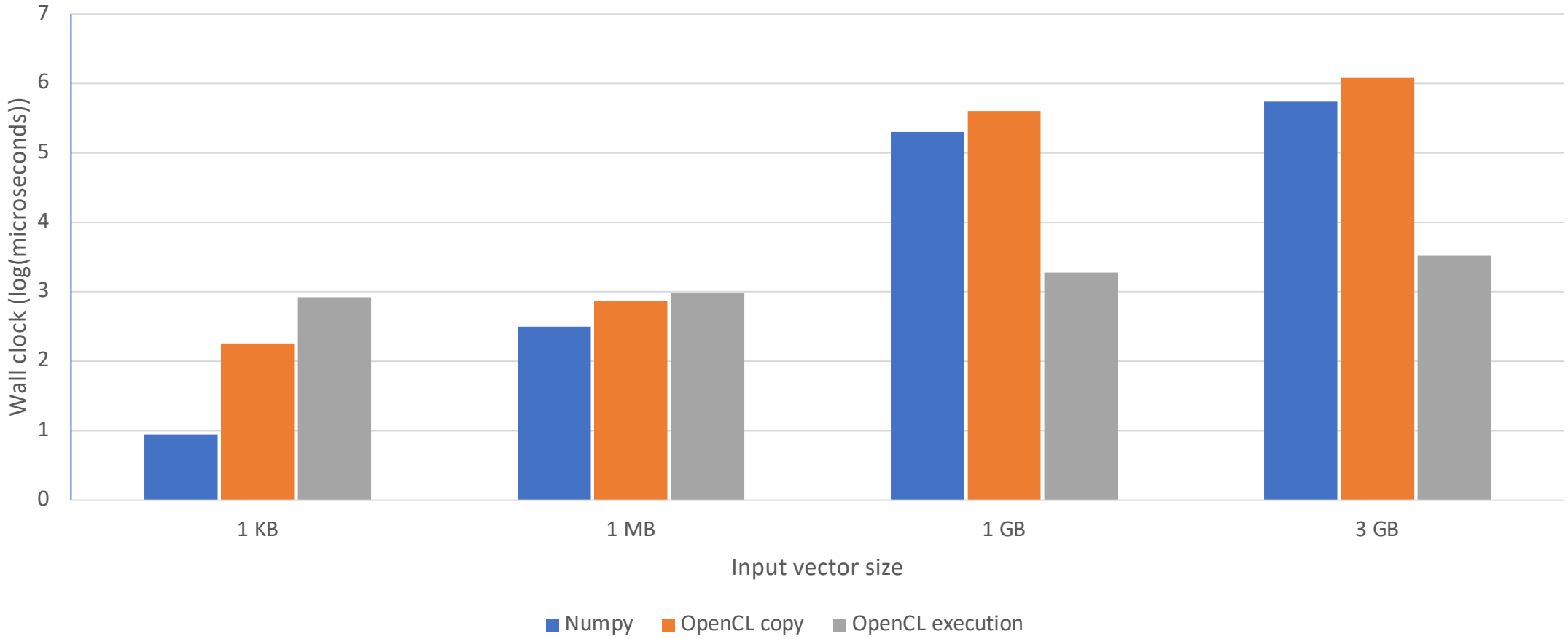
- Both frameworks uses the host/device model, where the programmer must write a host program and a device program.
- The device program is made up of one or more compute kernels.
- Each kernel is responsible for moving memory throughout the device memory hierarchy.
- The host program is responsible for moving memory to/from the host and device memory, and for invoking the device kernels.
- The host program can be written in C or C++ and as such also any language which can utilize C bindings, such as Python.
- Kernels are written in a C derivation in OpenCL and extended C++ in CUDA.

OpenCL example – vector add

```python
import numpy as np
import pyopencl as cl

# Generate input data
n = np.uint32(1024) # 4 KB
a = np.random.rand(n).astype(np.float32)
b = np.random.rand(n).astype(np.float32)
c = np.empty_like(a)

# Compute the expected result using CPU
c_expected = a + b

# Create the OpenCL context
context = cl.create_some_context()
queue = cl.CommandQueue(context)

# Allocate device memory
mf = cl.mem_flags
a_device = cl.Buffer(context, mf.READ_ONLY, a.nbytes)
b_device = cl.Buffer(context, mf.READ_ONLY, b.nbytes)
c_device = cl.Buffer(context, mf.WRITE_ONLY, c_expected.nbytes)

# Load and compile the kernel
with open('vector_add.cl', 'r') as f:
    kernel_source = f.read()
program = cl.Program(context, kernel_source).build()

# Copy input to device
cl.enqueue_copy(queue, a_device, a)
cl.enqueue_copy(queue, b_device, b)

# Execute kernel
program.vector_add(queue, a.shape, None, a_device, b_device, c_device, n)

# Copy result from device
cl.enqueue_copy(queue, c, c_device)

# Check results
print ('Difference:', np.linalg.norm(c - c_expected))
```

```c
__kernel void vector_add(
                           __global float *a,
                           __global float *b,
                           __global float *c,
                           const unsigned int n) {
    // Get the thread id
    int id = get_global_id(0);

    // Don't go out of bounds
    if (id < n)
        c[id] = a[id] + b[id];
}

```

OpenCL example – performance



Application Specific Integrated Circuit (ASIC)

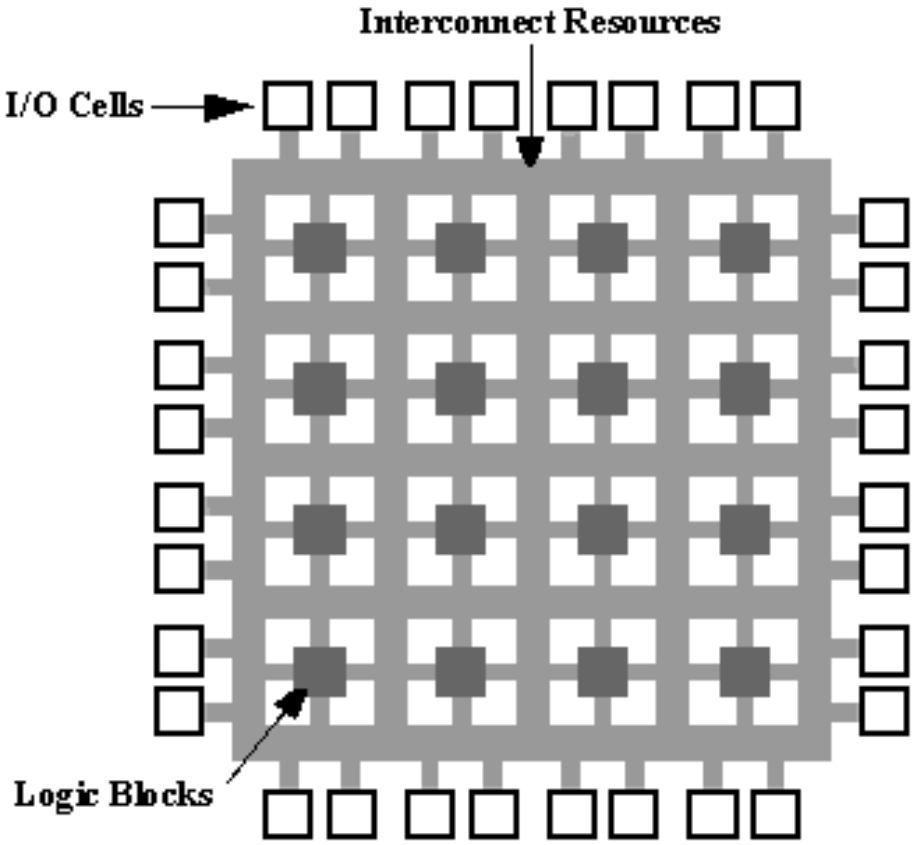
- All the examples given are ASICs, where the circuit is final once printed.
- This will always give the best performance, compared to general-purpose hardware, as one could make said hardware in the worst case.
- However, the starting price is 500 000 $, where one has to order 1000s of samples.
- If there is an error in the design the samples are useless, as they cannot be modified once printed.

Reconfigurable hardware

- Reconfigurable hardware is the middleground between general-purpose hardware and ASICs.
- Rather than supporting an instruction set, the programmer has to describe the circuit performing the computations. This enables better utilization, as only the circuit needed is implemented, but becomes harder to implement complex designs.
- Their design can be reconfigured multiple times, which make them more flexible than ASICs. This also lowers the performance, compared to ASICs.

Field-Programmable Gate Array (FPGA)

- FPGAs are an example of reconfigurable hardware.
- It consists of a big interconnect and a grid of components, such as logic gates, Look-Up Tables (LUT), multiplexors and Flip-Flop (FF) registers.
- Hardware is designed through how these components are connected through the interconnect. The resulting implementation is equivalent to a printed circuit of the same design, although at a performance hit.
- FPGAs are generally more efficient at tasks, compared to generel-purpose hardware, due to the same benefits as GPUs for compute.



Working with hardware models

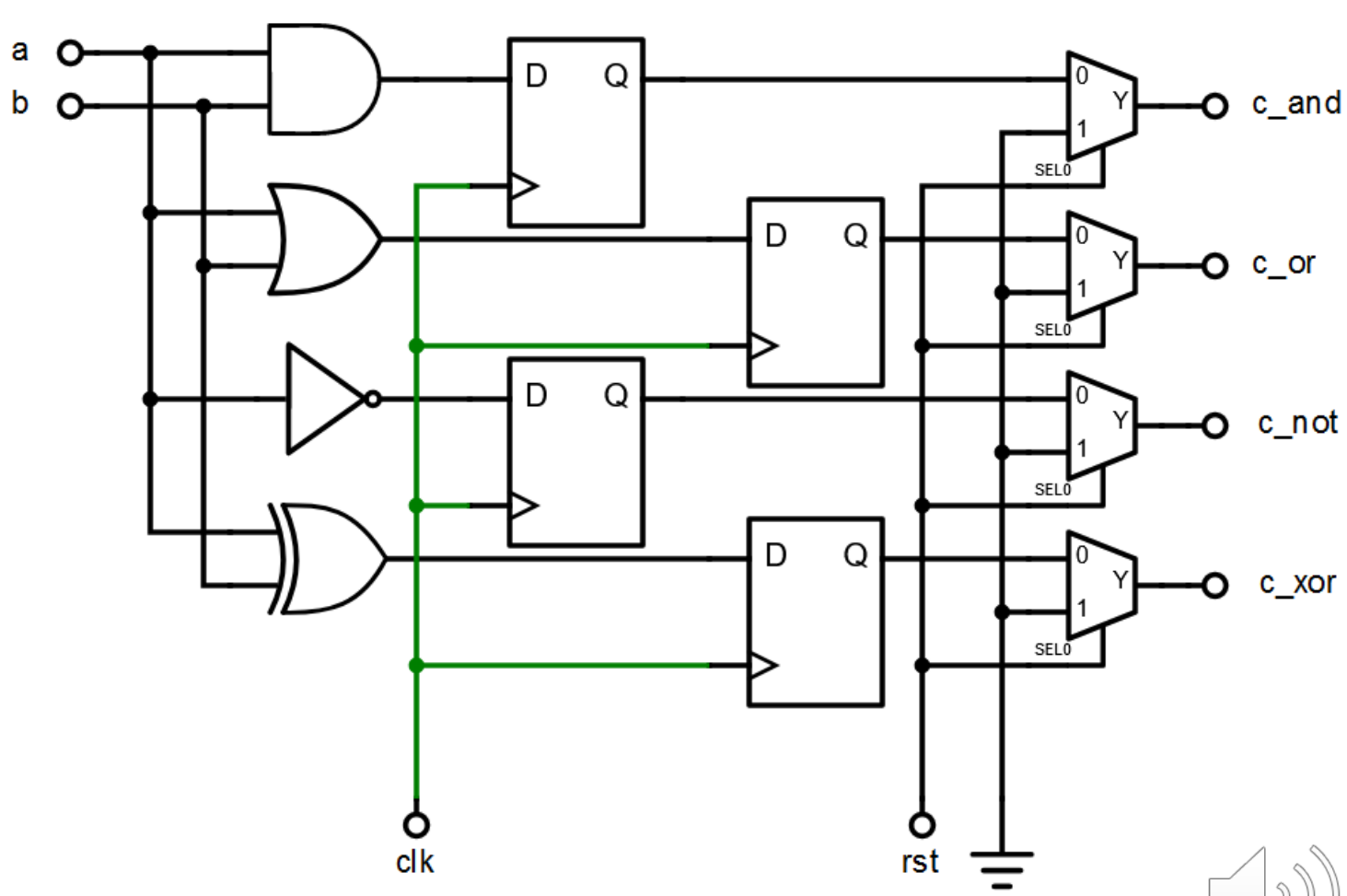
- While parallel programming becomes harder, as multiple events occur at the same time, programming hardware models become even harder as there is a temporal dimension.
- Events occurs concurrently, and signals take time to propegate on the wires.
- Most designs handles this signal propagation, by guarding all the registers with a clock signal.
- This clock signal is then used to synchronize events, so the programmer can better control the order of events.

Hardware Description Language (HDL)

- FPGAs are programmed using HDLs, such as VHDL and Verilog.
- Both languages are parallel languages for describing circuits.
- They are very tedious to work with, as they are very verbose and very low-level.
- Even though the languages have evolved, FPGA vendors are slow to adopt these changes / additions to the languages.

VHDL example – Basic logic gates

```vhdl
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
USE IEEE.NUMERIC_STD.ALL;

entity logic_gates is
    port(
        a: in std_logic;
        b: in std_logic;

        c_and: out std_logic;
        c_or:  out std_logic;
        c_xor: out std_logic;
        c_not: out std_logic;

        rst: in std_logic;
        clk: in std_logic
    );
end logic_gates;

architecture RTL of logic_gates is

    process(clk, rst)
    begin
        if RST = '1' then
            c_and <= '0';
            c_or  <= '0';
            c_xor <= '0';
            c_not <= '0';
        elsif rising_edge(CLK) then
            c_and <= a and b;
            c_or  <= a or b;
            c_xor <= a xor b;
            c_not <= not a;
        end if;
    end process;

end RTL;
```



Test benches

- In order to test an implementation written in HDLs, test benches are needed.
- Generally, they are an additional program, which drives the input wires and verifies whether the output wires behave as expected.
- They have to be written in HDLs as well, although they are no longer limited to a synthesizable subset of the language.

VHDL testbench example – Basic logic gates

```vhdl
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
USE IEEE.NUMERIC_STD.ALL;

entity logic_gates_tb is
end logic_gates_tb;

architecture tb of logic_gates_tb is
    signal a, b : std_logic;
    signal c_and, c_or, c_not, c_xor : std_logic;

    signal clk, rst : std_logic;
    signal stop_clock : boolean := false;
begin
    inst_logic_gates : entity work.logic_gates
    port map(
        a => a,
        b => b,
        c_and => c_and,
        c_or => c_or,
        c_not => c_not,
        c_xor => c_xor,
        clk => clk,
        rst => rst
    );

    clk_proc: process
    begin
        while not stop_clock loop
            clk <= '1';
            wait for 5 ns;
            clk <= '0';
            wait for 5 ns;
        end loop;
        wait;
    end process;

    tester: process
    begin
        rst <= '1';
        wait for 5 ns;
        rst <= '0';

        wait until rising_edge(clk);
        a <= '0';
        b <= '0';

        wait until rising_edge(clk);
        a <= '0';
        b <= '1';

        wait until falling_edge(clk);
        assert(c_and = '0');
        assert(c_or = '0');
        assert(c_not = '1');
        assert(c_xor = '0');

        wait until rising_edge(clk);
        a <= '1';
        b <= '0';

        wait until falling_edge(clk);
        assert(c_and = '0');
        assert(c_or = '1');
        assert(c_not = '1');
        assert(c_xor = '1');

        wait until rising_edge(clk);
        a <= '1';
        b <= '1';

        wait until falling_edge(clk);
        assert(c_and = '0');
        assert(c_or = '1');
        assert(c_not = '0');
        assert(c_xor = '1');

        wait until rising_edge(clk);

        wait until falling_edge(clk);
        assert(c_and = '1');
        assert(c_or = '1');
        assert(c_not = '0');
        assert(c_xor = '0');

        stop_clock <= true;
        wait;
    end process;
end architecture tb;
```

VHDL testbench example – Basic logic gates