






FPGAs with SYCL

Syllabus

-  Introduction to FPGAs
-  Key FPGA metrics
-  Some Optimization patterns
 -  Shift Registers
 -  Pipes

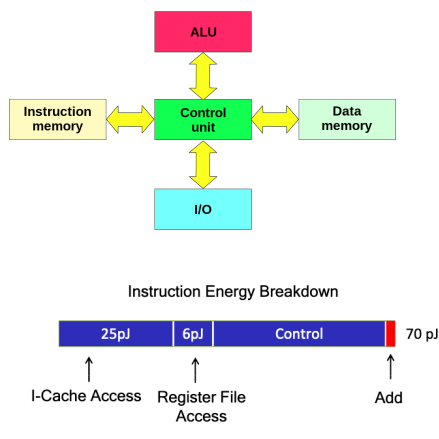
Goals

- Understand the architecture of FPGA
- Learns the basics of SYCL programming on FPGA
- Use optimized patterns for coding on FPGA

Load Store vs. Dataflow Architectures

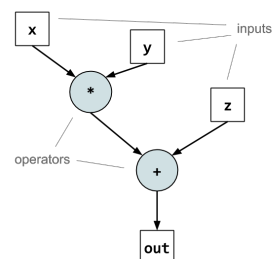
Load Store

- Energy per instruction: 70 pJ



Static Dataflow ("non von Neumann")

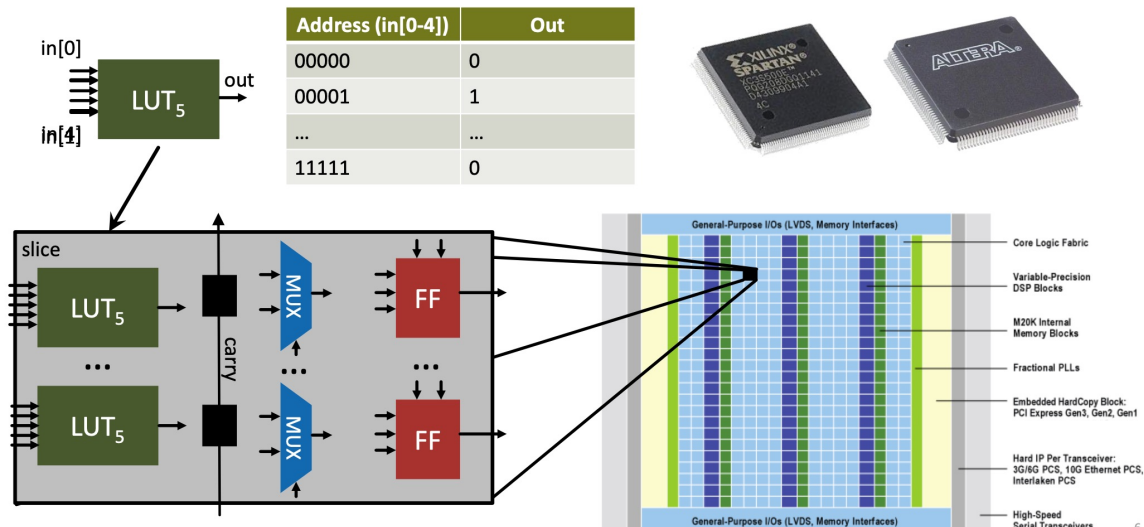
- Energy per operation: 1-5pJ



Turing Award 1977 (Backus): "Surely there must be a less primitive way of making big changes in the store than pushing vast numbers of words back and forth through the von Neumann bottleneck."

Source: Mark Horowitz and Torsten Hoefler

Field Programmable Gate Arrays (FPGAs)



Source: Torsten Hoeffler

High Performance FPGA Hardware Overview

High-Performance FPGA Hardware Overview



- 14 nm Intel Tri-Gate
 - 1 GHz
- 10 TF single precision
- 5.5M Logic Elements
 - 4-input LUT, register, carry, etc.
- Block RAM: 28.6 MiB
- Hardened DRAM controller DDR 4
 - Various options for memory
- Hyper Flex Interconnect with Regs.
- TDP: 125W (estimated)



- 14 nm
 - ~600 MHz
- 8,490 DSPs (3.8 TF single prec.)
- 2.5M Logic Elements
 - 1,082,000 5-input LUTs
 - 2,164,000 FFs
- Block RAM: 39.4 MiB
- TDP: 225 W

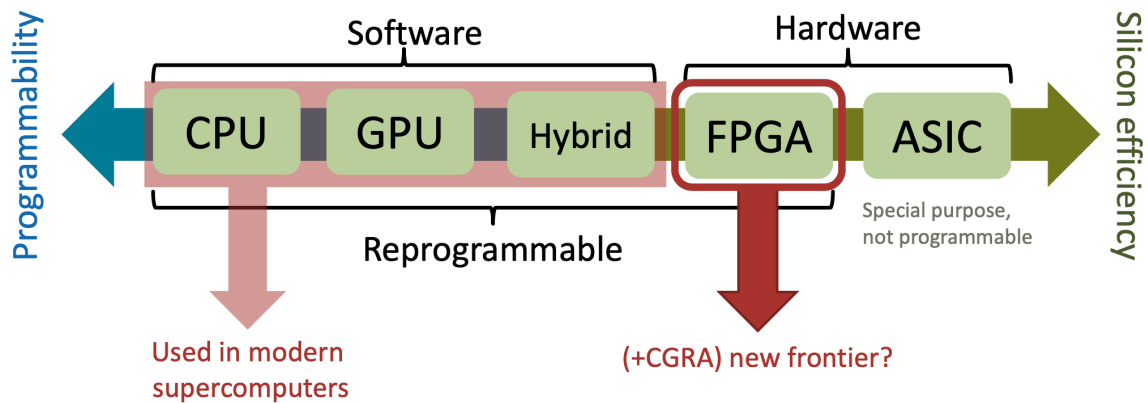
GPU



- 12 nm
 - 1455 MHz
- 5,120 cores (15.7 TF single prec.)
 - CUDA programming
- On-chip memory:
 - Registers: 20.8 MiB
 - L1/SM: 7.7 MiB
 - L2 Cache: 6.1 MiB
- TDP: 300W

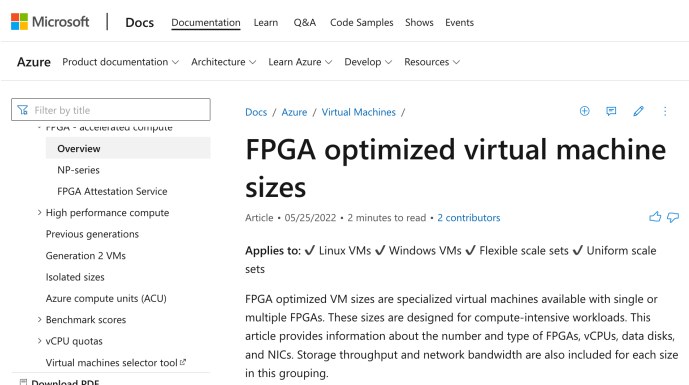
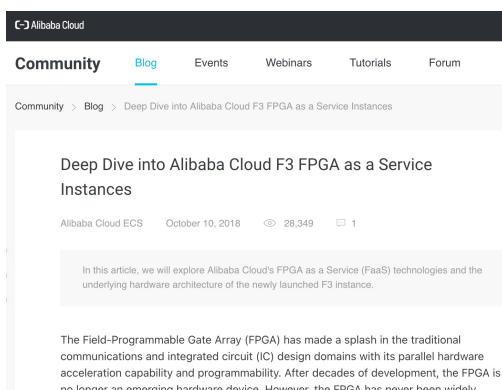
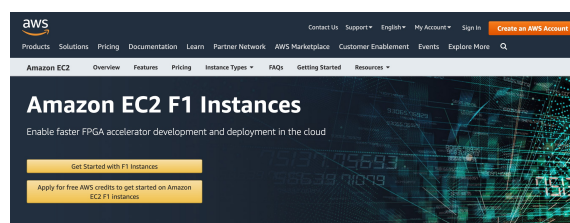
Source: Torsten Hoeffler

Programmability vs. efficiency



Source: Torsten Hoeffler

FPGAs already available at mayor cloud providers



Why GPU is the main accelerator?



FASTER ?



HIGHER PRODUCTIVITY?

GPUs are more productive

• e.g.; CUDA, OpenCL, ...

• More optimized libraries for GPUs

• Latest advancement in High Level Synthesis with FPGA...

• High Level Synthesis translates code from programming languages such C++/C/SYCL to Hardware Description Languages to enable synthesis, placement, and routing

HLS Design Methodology

Design Creation	Functional Verification	RTL Synthesis	Place and Route	Gate Level Verification
-----------------	-------------------------	---------------	-----------------	-------------------------

Traditional RTL Design Methodology

Design Creation	Functional Verification	RTL Synthesis	Place and Route	Gate Level Verification
-----------------	-------------------------	---------------	-----------------	-------------------------

FPGA Design Concepts

FPGA Architecture

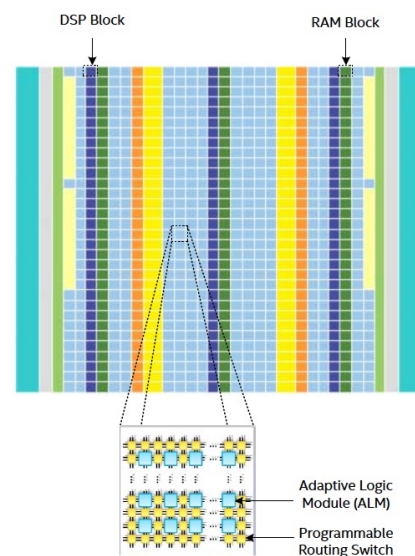
2D fabric with several blocks:

- Adaptive Logic Modules
- Digital Signal Processing (DSP)
- Random Access Memories

Configurable interconnection network

Design area or FPGA area:

- number ALM + DSP + RAM
(design)

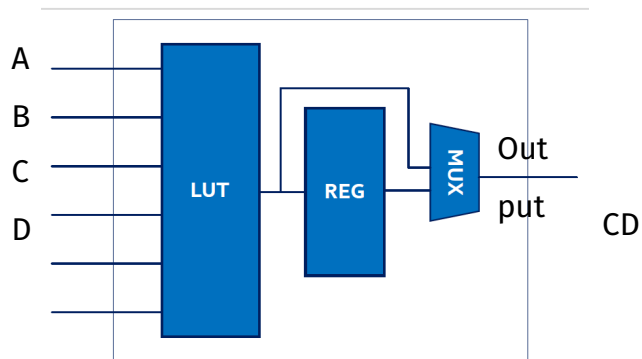


Source: Intel

Adaptive Logic Module

Basic Block of the FPGA

- Lookup table (LUT) + output register
- LUT builds boolean circuits
- Inside the LUT there is an SRAM block

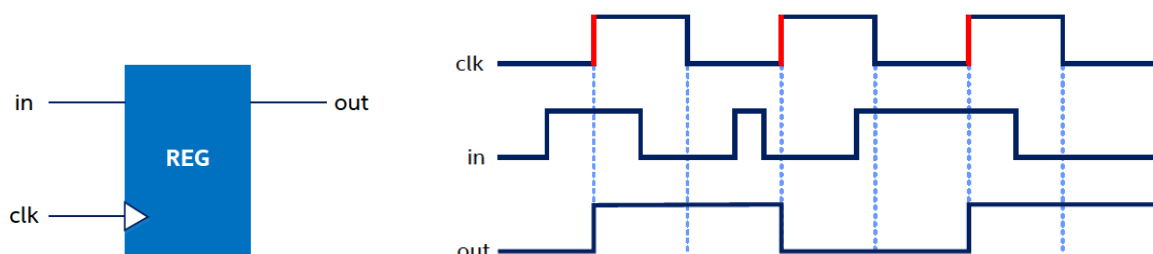


Output truth table

		AB			
		00	01	10	11
00	0	0	0	0	0
01	0	0	0	0	0
10	0	0	0	0	0
11	0	0	0	0	1

The register inside the Adaptive Logic Module

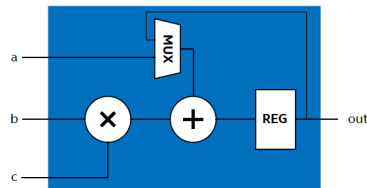
- Basic storage element
- Synchronous operation



Source: Intel

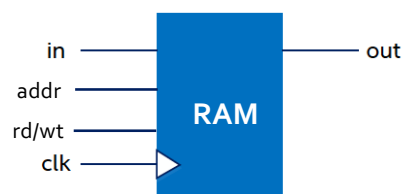
Digital Signal Processors DSP

- ALM are not suitable for fixed- and floating- point arithmetic
- FPGAs includes DSP to speed-up those operations
- DSPs are fast compared to ALM for FP
 - Use in ML, ...



Block Random Access Memory (RAM)

- High Density Memory Block inside the FPGA



FPGA Metrics for Performance

Maximum Frequency

Latency

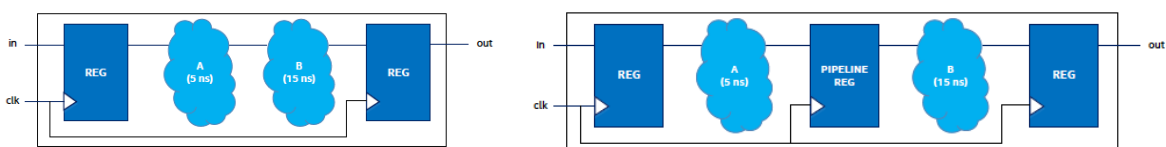
Initiation Interval

Throughput

Maximum Frequency

Maximum rate at which registers are updated

The longest combinational path, *critical path*, sets the maximum frequency



What is the maximum frequency of these circuits?

Pipelining: add registers to increase F_{\max}

Latency (L)

≡ Total delay to complete one or several operations

≡ Pipelining may increase total delay

≡ Programming Caveat:

≡ In single-work-item kernels (task-parallel programming), the number of stages of the pipeline gives an insight for the minimum number of elements that should be processed by the kernel to guarantee good occupation

Pipeline 6 stages



```
for(i=0; i < N; ++i) {  
    ...  
}  
// what if N < 6 ?
```

Throughput (Th)

≡ Number of elements/item processed per unit of time

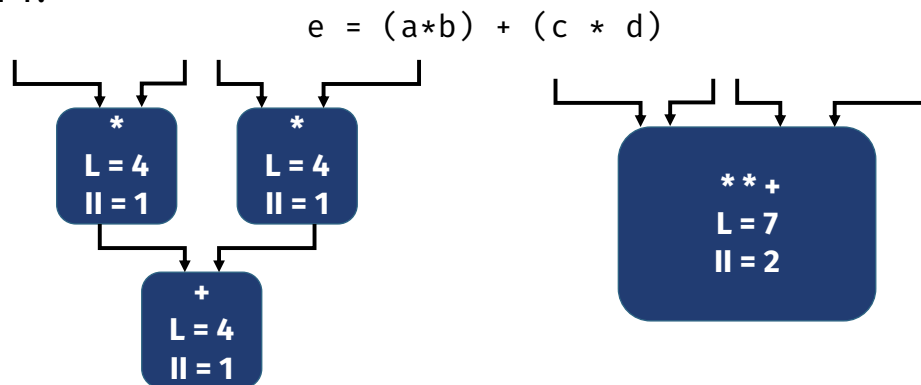
≡ Key metric for many circuits/programs



Source <https://www.cronica.com.ar/sociedad/Escapada-a-Cascadas-Cifuentes-Una-maravilla-oculta-de-la-Provincia-de-Buenos-Aires--20220128-0112.html>

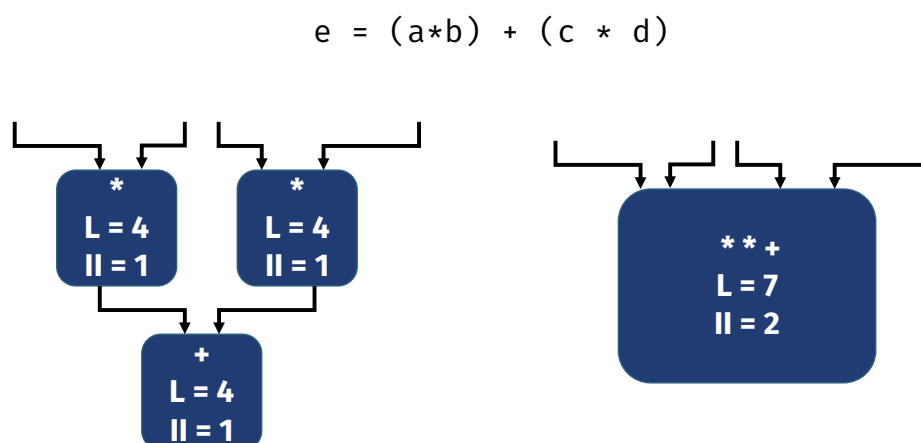
Initiation Interval (II)

- Minimum delay between consecutive elements, many times for loop iterations
- How fast is your application running with II values greater than 1?

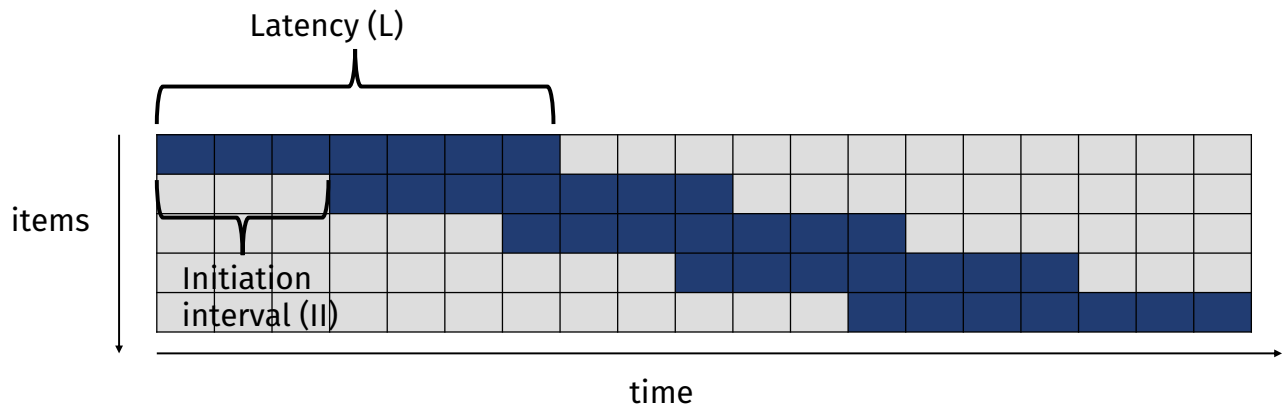


Quiz:

- Compute the latency, initiation interval and throughput for the previous example



Pipeline simple performance model

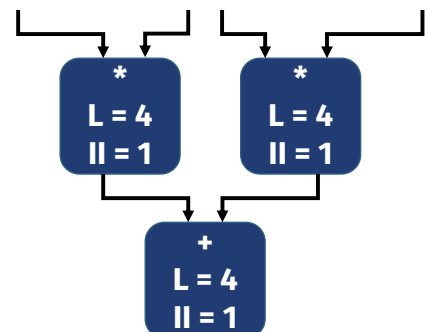


$$\text{Execution Time} = L + (\text{items} - 1) * II$$

Loops for single-item/task kernel

```
for(size_t i = 0; i < n; ++i) {
    e[i] = (a[i]*b[i]) + (c[i] * d[i]);
}
```

Iterations	Delay
1	8
2	9
...	
100	107
N	$8 + (N - 1)$



For very large N values,
latency is irrelevant

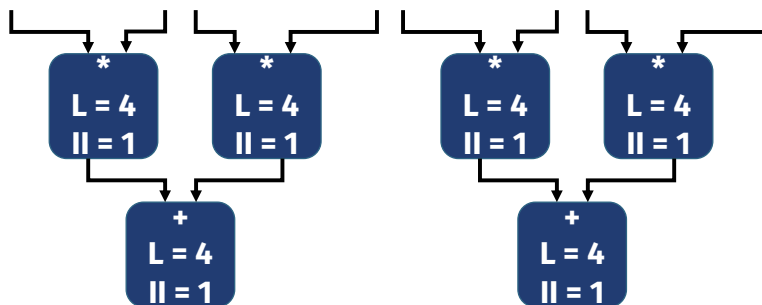
Loop unrolling

Enabled by `#pragma unroll <unroll_factor>`

Source of performance improvement, *spatial computing*

Compiler never unrolls by default

```
#pragma unroll 2
for(size_t i = 0; i < n; ++i) {
    e[i] = (a[i]*b[i]) + (c[i] * d[i]);
}
```



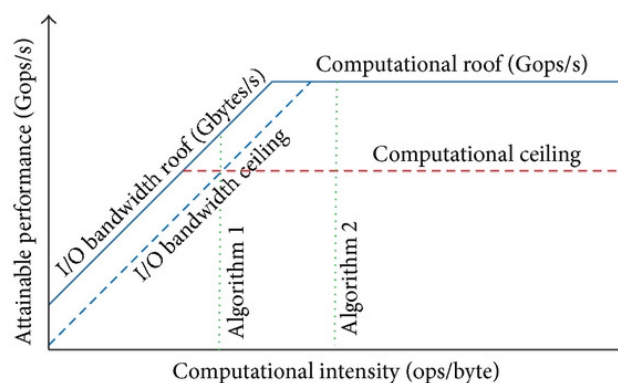
Iters	Delay
1	8
2	8
...	
100	$8 + 45 = 53$
N	$8 + \lceil (N - 1) / \text{unroll} \rceil$

Vectorization

The compiler can automatically perform vectorization

Effect similar to loop unrolling

Unroll and/or vectorization can reach bandwidth limit



Source: <https://doi.org/10.1155/2013/428078>

Shift Register Pattern

⌚ Sometimes the latency of an operation degrades performance

```
mul = 1.0f
for(size_t i = 0; i < n; ++i) {
    mul*=a[i];
}
```

⌚ Solution: Relax Loop-Carried Dependency

⌚ instead of using a single variable to store the multiplication results, operate on M copies of the variable, and use one copy every M iterations

Relaxed Loop-Carried Dependency

⌚ Implementation:

```
float mul = 1.0f
float mul_copies[M]= {1.0f, ..., 1.0f}

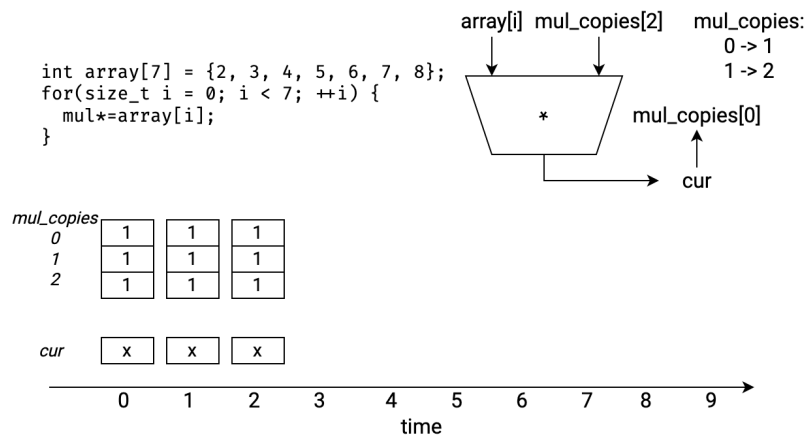
for(size_t i = 0; i < n; ++i) {
    auto cur = mul_copies[M-1] * A[i];

    // shift values in register
    #pragma unroll
    for(size_t j = M-1; j > 0; --j) {
        mul_copies[j]=mul_copies[j-1];
    }
    // store value in lowest position
    mul_copies[0] = cur;
}

// final reduction
#pragma unroll
for (int i = 0; i < M; ++i) {
    mul *= mul_copies[i];
}
```

Timing of Relaxed Loop-Carried Optimization

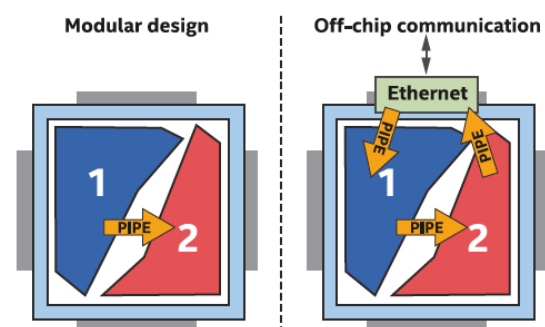
Assume multiplication latency 3 cycles



Total Cycles
 original: $3 * N = 3 * 7 = 21$
 optimized: $3 + (N-1) + \text{reductions} = 3 + 6 + 6 = 15$

Pipes

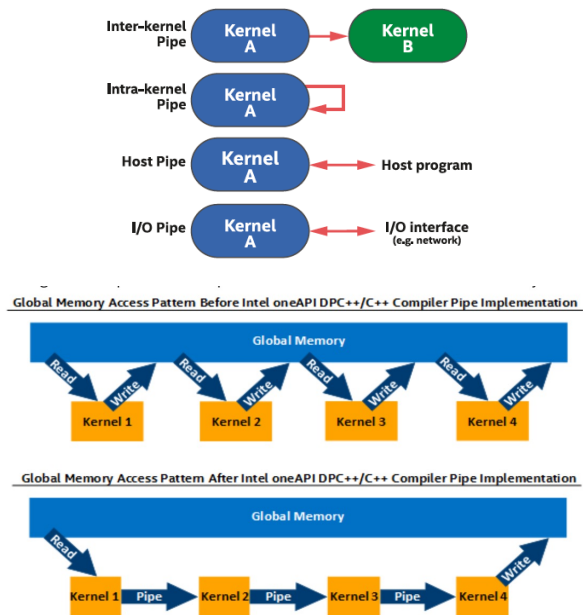
- Share data and synchronize between concurrent kernels
- Implemented with FIFO queues
 - Implicit control synchronization between kernels
 - Buffer enables some latency tolerance (dynamic behavior)
- Each kernel should be launched in a different command queue
- Enable modular designs



Source: Data Parallel C++

Pipe Description

- Inter kernel are the most common pipe type
- Two basic operations
 - Read
 - Write
- Two modes
 - Blocking (single param)
 - Non-blocking (two params, status)
- Capacity
 - Number of entries written without intermediate read ops



Pipe example

```
using my_pipe = ext::intel::pipe<class some_pipe, int, 8 >;
// the class my_pipe has to be unique
```

```
void Producer(queue &q, buffer<int, 1> &input_buffer) {
    std::cout << "Enqueuing producer...\n";

    auto e = q.submit([&](handler &h) {
        accessor input_accessor(input_buffer, h, read_only);
        auto num_elements = input_buffer.get_count();

        h.single_task<ProducerTutorial>([&]() {
            for (size_t i = 0; i < num_elements; ++i) {
                my_pipe::write(input_accessor[i]);
            }
        });
    });
}
```

```
void Consumer(queue &q, buffer<int, 1> &output_buffer) {
    std::cout << "Enqueuing consumer...\n";

    auto e = q.submit([&](handler &h) {
        accessor out_accessor(out_buf, h, write_only, no_init);
        size_t num_elements = output_buffer.get_count();

        h.single_task<ConsumerTutorial>([&]() {
            for (size_t i = 0; i < num_elements; ++i) {
                int input = my_pipe::read();
                int answer = ConsumerWork(input);
                output_accessor[i] = answer;
            }
        });
    });
}
```

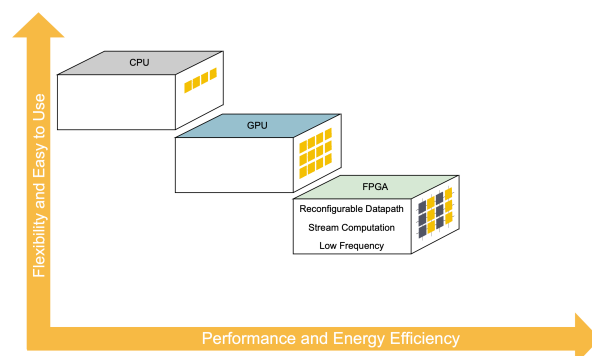


Some peculiarities of SYCL on Intel's FPGAs

- “A SYCL system that has FPGAs installed does not support multi-process execution”
- Always perform all FPGA related activity from a single host thread

Conclusions

- FPGAs can provide better perf/watt than CPU, GPU
- Now, you can use SYCL for all three devices
- For FPGA, use specific optimizations



Source: Angélica Dávila