Introduction to FPGA computing for the HPC ecosystem

Emmanuel Kieffer

High Performance Computing & Big Data Services

hpc.uni.lu

hpc@uni.lu

@ULHPC

LUXEMBOURG LET'S MAKE IT HAPPEN

SUPERCOMPUTING LUXEMBOURG

UNIVERSITÉ DU LUXEMBOURG

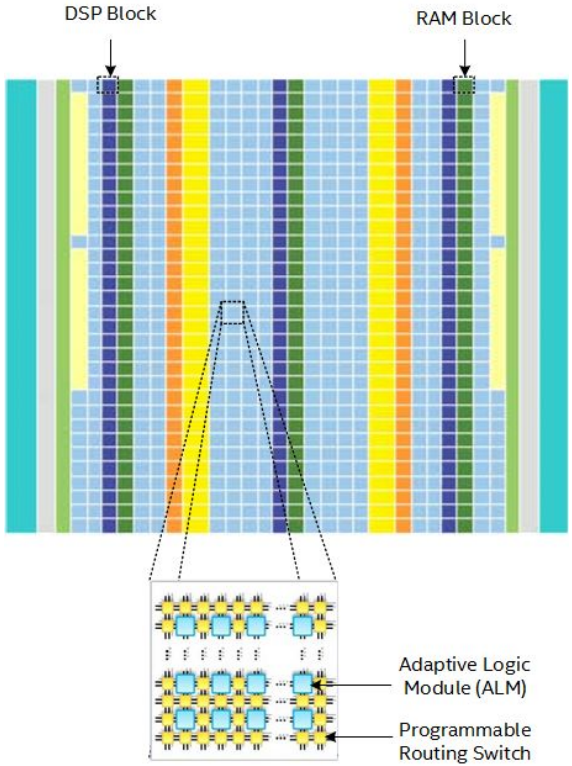
FPGA architecture

UNIVERSITÉ DU LUXEMBOURG

FPGA Architecture Overview

- A field-programmable gate array (FPGA)
- Reconfigurable semiconductor integrated circuit (IC)
- FPGAs are a cheaper off-the-shelf alternative

- Grid of configurable logic composed of:
  - Logic Element (LEs) or Adaptive Logic Modules (ALMs)
  - Programmable switches
  - Digital Signal Processing blocks
  - RAM blocks
  - Etc …



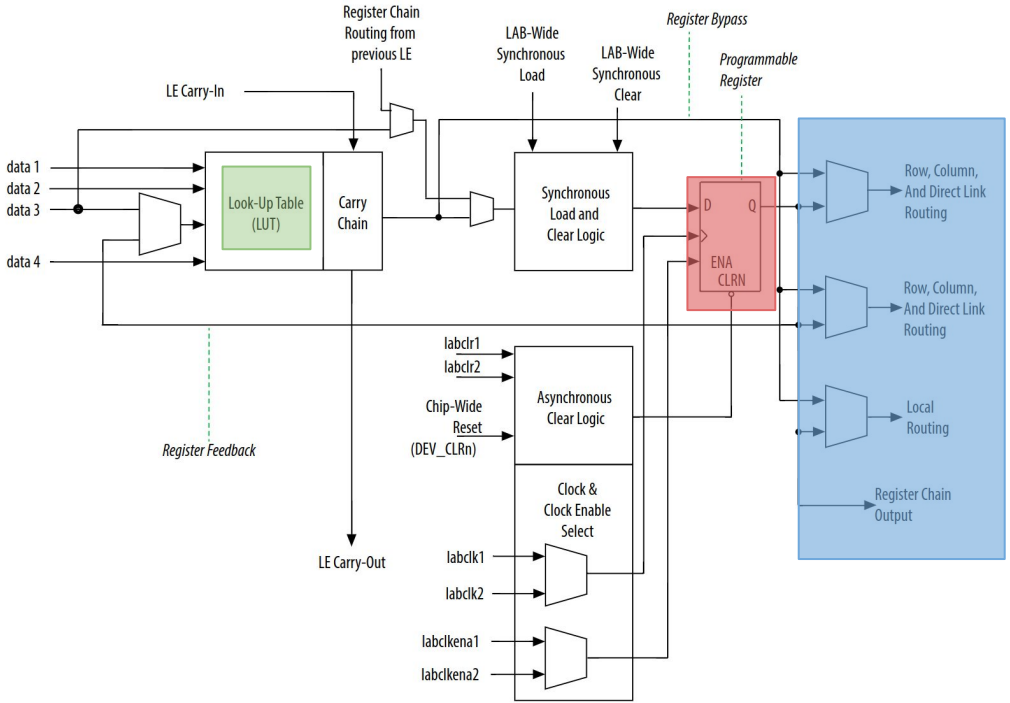
source: Intel

Logic Element

- Main building block
- Can built any arbitrary logic circuit

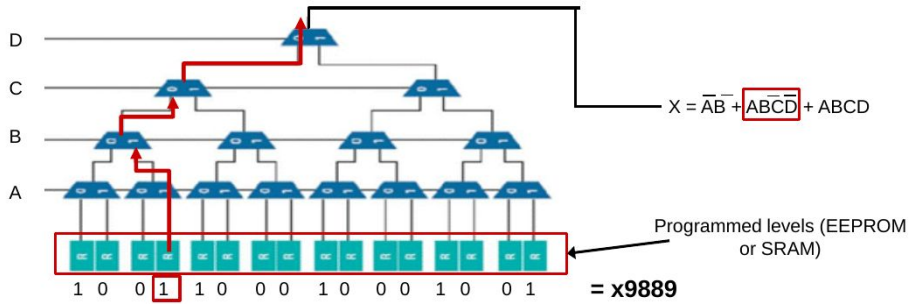
- Lookup table (LUT)
- Register
- Multiplexer part

LE High-Level Block Diagram for Intel® Cyclone® 10 LP Devices (source: Intel)

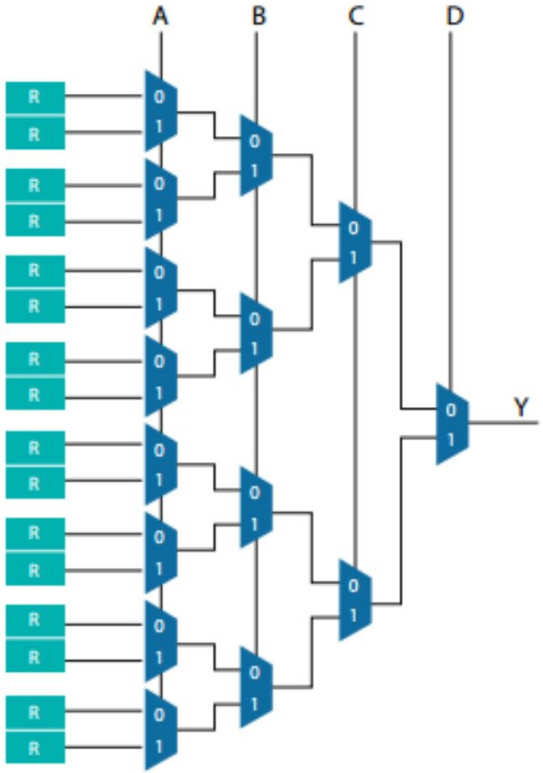


Look-up Table (LUTs)

- Built out of:
  - EEPROM or SRAM holding the configuration, i.e., LUT-mask
  - Set of multiplexers for bits selection drove to the output
- A k-LUT can implement any function of k inputs
- 2^k SRAM bits and 2^k:1 multiplexers
- Ex: 4-LUT with (A,B,C,D) as inputs

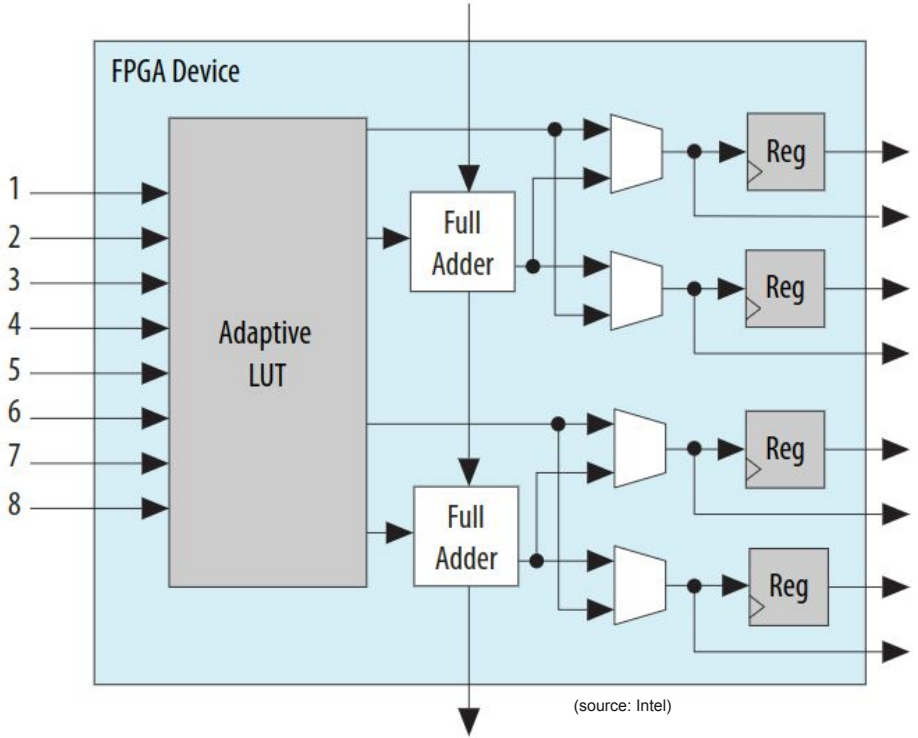


source: Altera doc



Adaptive Logic Modules (ALM)

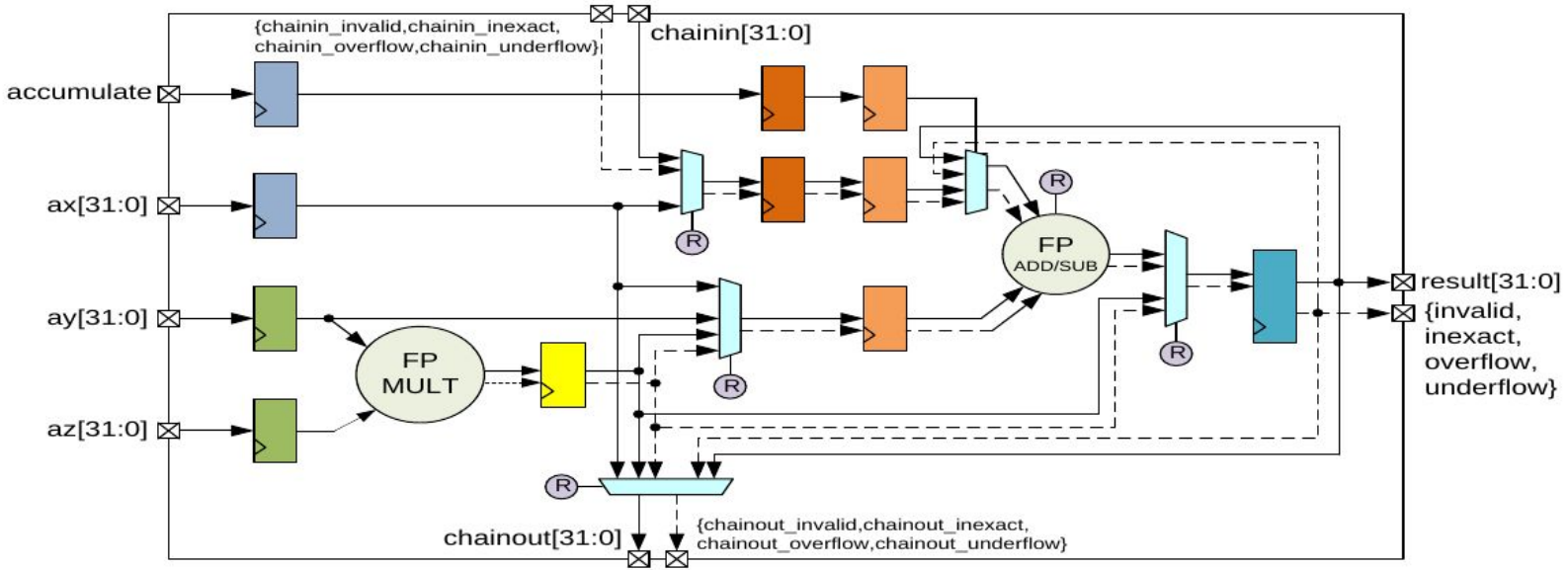
- Extend LEs
- Improved performance
- More complex but also more flexible



(source: Intel)

DSP Block

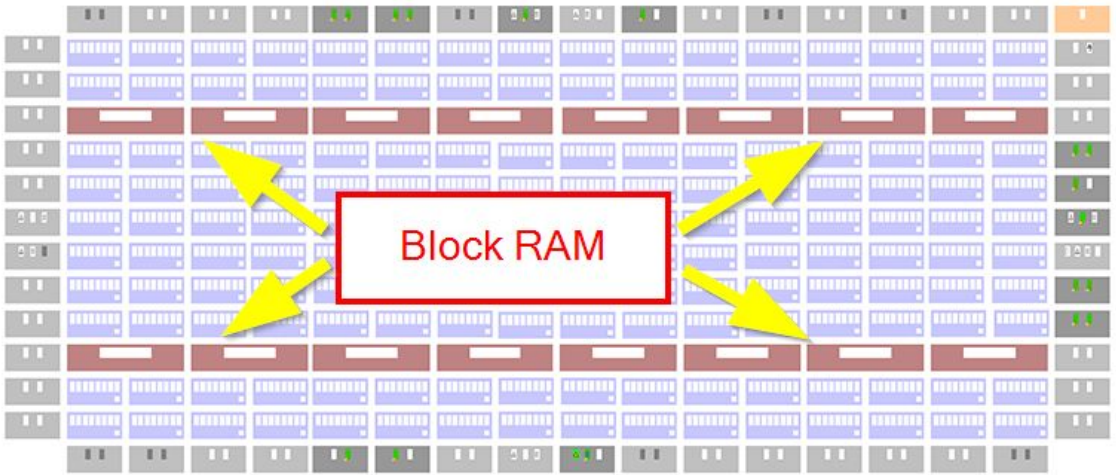
- High-performance multiply/add/accumulate operations
- Arria 10 DSP Block can do 32-bit IEEE-compliant floating-point multiply-add



(source: alteral)

Random Access Memory (RAM) Blocks

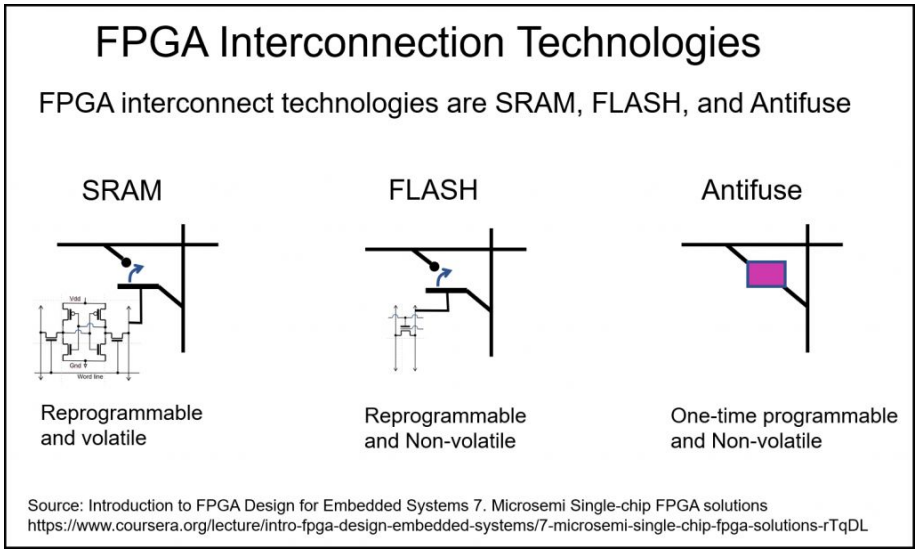
- on-chip memory structures to support design
- Typical sizes:
  - Single memory block is 20 Kilobits
  - MLABs are general-purpose dual-port memory SRAM array (640 bits)



(source:https://vhdlwhiz.com/terminology/block-ram/)

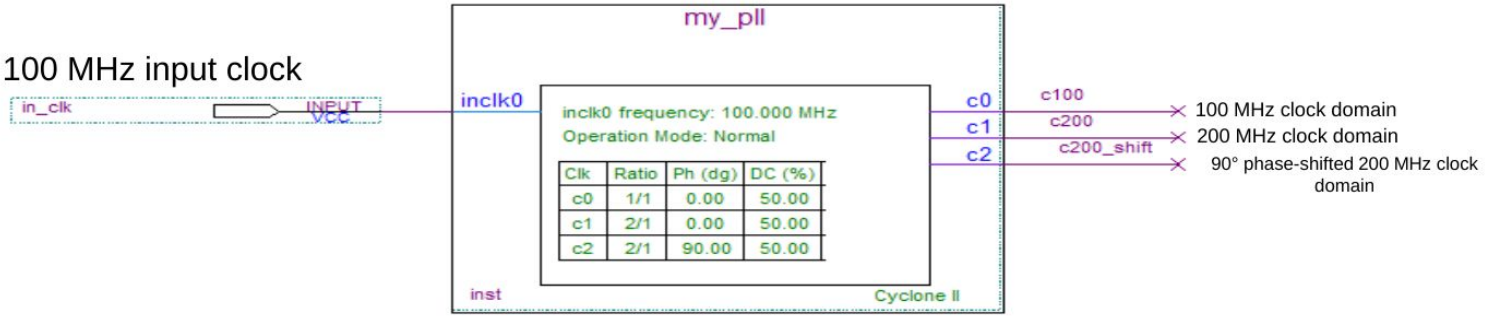
FPGA Interconnect

- Type:
  - SRAM : reprogrammable and volatile
  - FLASH: reprogrammable and non-volatile
  - Antifuse: One-time programmable and non-volatile
- Most FPGAs use SRAM cell technology to program interconnect and LUT function levels



Phase-Locked Loop (PLL)

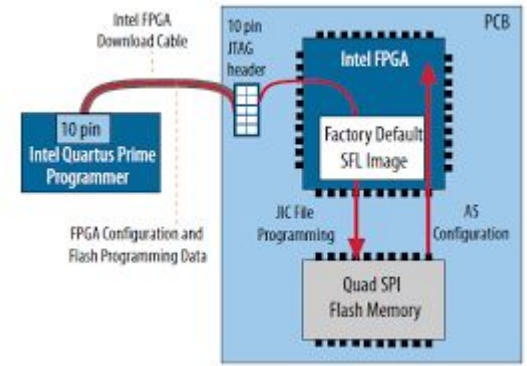
- Many FPGAs use a phase-locked loop (PLL) to increase the internal clock speed.
- Ex: The iCE40 on the IceStick allows you to run up to 275 MHz by setting the internal PLL with the onboard 12 MHz reference clock.



(source:Altera)

FPGA programming

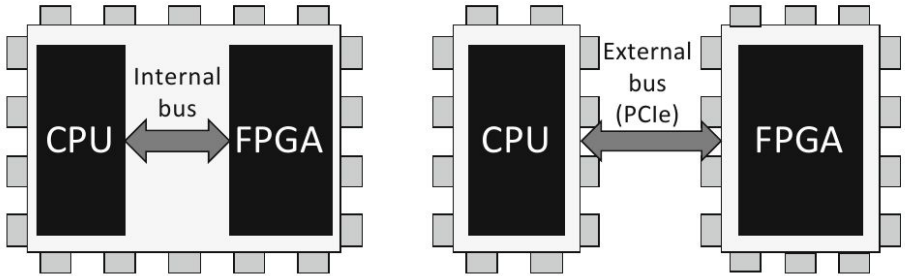
- Active mode: FPGA controls programming sequence automatically at power on
- Passive mode: CPU controls programming
- Program stored using either EEPROM, CPLD, SRAM, etc …



(source:Intel -- Cyclone 10 GX FPGA

FPGA systems (CPU-FPGA)

- Modern FPGA cards combine CPU and FPGA
- Internal bus: low-power embedded devices (System on Chip)
- External bus: PCIe for high-performance computing



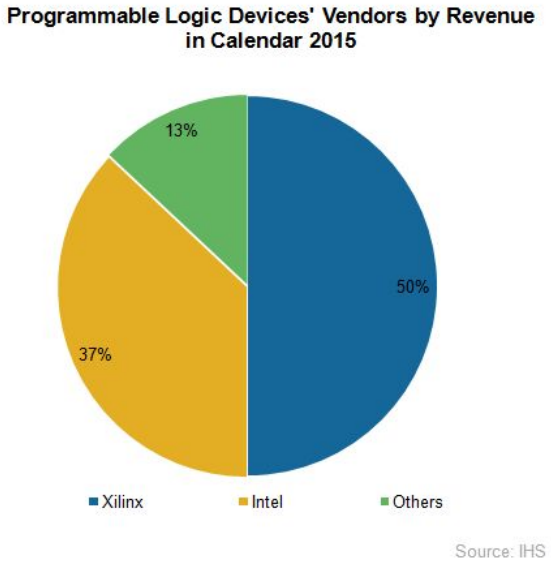
(source:Design of FPGA-Based Computing Systems with OpenCL)

FPGA on the market

UNIVERSITÉ DU LUXEMBOURG

FPGA vendors

- Two major FPGA vendors:
  - Intel [Altera]
  - Xillinx [AMD]
- Intel acquired Altera in 2015
- Xillinx is solely focusing on the FPGA market
- While Intel is a sum of many parts
- Both profiles are very interesting for heterogeneous computing
- Among the others:
  - Lattice Semiconductor
  - QuickLogic
  - Microchip Technology
  - Achronix
  - Efinix



Intel® FPGA family

- Intel® FPGAs are ideal for a wide variety of applications, from high-volume applications to state-of-the-art products.
- Each FPGA series with different features:
  - Embedded memory,
  - Digital Signal Processing (DSP) blocks,
  - High-speed transceivers,
  - High-speed I/O pins to cover a broad range of applications
- Intel® has four classes of FPGAs to meet market needs from the industry's highest density and performance to the most cost effective:
  - Agilex FPGA and SoC devices accelerate your delivery of the most advanced bandwith-intensive applications
  - Stratix 10 FPGA and SoC family enables you to deliver high-performance, state-of-the-art products to market faster with lower risk and higher productivity
  - Arria family delivers optimal performance and power efficiency in the midrange
  - Cyclone 10GX FPGA series is built to meet your low-power, cost-sensitive design needs,enabling you to get to market faster

Intel® FPGA family

- Fabric + Tiles: built using heterogeneous 3D system-in-package (SiP) technology

Intel® FPGA Family	Technology	Architecture
Intel® Agilex™ F-Series and I-Series	10 nm SuperFin	Fabric + Tiles
Intel® Stratix® 10	14 nm FinFet	Fabric + Tiles
Intel® Arria® 10	20 nm Planar	Monolithic
Intel® Cyclone® 10 GX	20 nm Planar	Monolithic

Intel® - Xilinx Device Comparison

Application	Xilinx Devices	Intel Devices
Highest performance	Versal Prime	Intel Agilex F-Series
	Versal Premium	Intel Agilex I-Series
High performance	Virtex UltraScale+ Kintex UltraScale+ Zynq UltraScale+	Intel Agilex F-Series Intel Agilex I-Series Intel Stratix 10 GX Intel Stratix 10 SX Intel Stratix 10 TX Intel Stratix 10 MX Intel Stratix 10 DX
Mid-range	Virtex UltraScale Kintex UltraScale Zynq-7000	Intel Stratix 10 GX Intel Stratix 10 SX Intel Stratix 10 TX Intel Stratix 10 MX Intel Stratix 10 DX Intel Arria 10 GX Intel Arria 10 SX
Low cost	Artix -7	Intel Cyclone 10 GX

HLS v.s. HDL

- **HLS:** High-Level Synthesis allows designers to describe hardware using high-level programming languages like C, C++, or SystemC. This means that HLS works at a higher level of abstraction, where developers can describe algorithms or logic without specifying the exact hardware details.

- **SystemVerilog/VHDL**: These HDLs require a more detailed specification of the hardware, providing a gate-level or Register Transfer Level (RTL) description. They require knowledge of the specific hardware constructs, like registers, flip-flops, etc.

Productivity

- **HLS:** Usually, HLS offers faster development time since engineers can write code using familiar programming paradigms. Automated synthesis tools then translate the high-level code into RTL, allowing quicker prototyping.
- **SystemVerilog/VHDL**: Writing in HDLs typically takes more time as developers have to manually describe the low-level hardware details. This can result in more control and optimization but is generally more time-consuming.

Flexibility & Optimization

- **HLS:** While HLS can accelerate development, it often provides less control over the final hardware implementation. This might result in less efficient utilization of FPGA resources or higher latency compared to hand-crafted RTL code.
- **SystemVerilog/VHDL:** Since these languages allow developers to describe hardware at a more granular level, there is usually greater opportunity for manual optimization of the design.

Learning Curve

- **HLS:** Generally, HLS has a lower learning curve for software engineers or those familiar with C/C++. This makes it more accessible to developers who might not have a hardware background.
- **SystemVerilog/VHDL:** Learning these languages typically requires a deeper understanding of hardware concepts. Thus, there's a steeper learning curve, but it can provide more expertise in hardware design.

Verification

- **HLS:** Verification might be less comprehensive compared to what can be achieved with traditional HDLs, although tools are evolving to bridge this gap.
- **SystemVerilog/VHDL:** These languages offer robust verification methodologies and frameworks that are often used in industry for rigorous verification of complex designs.

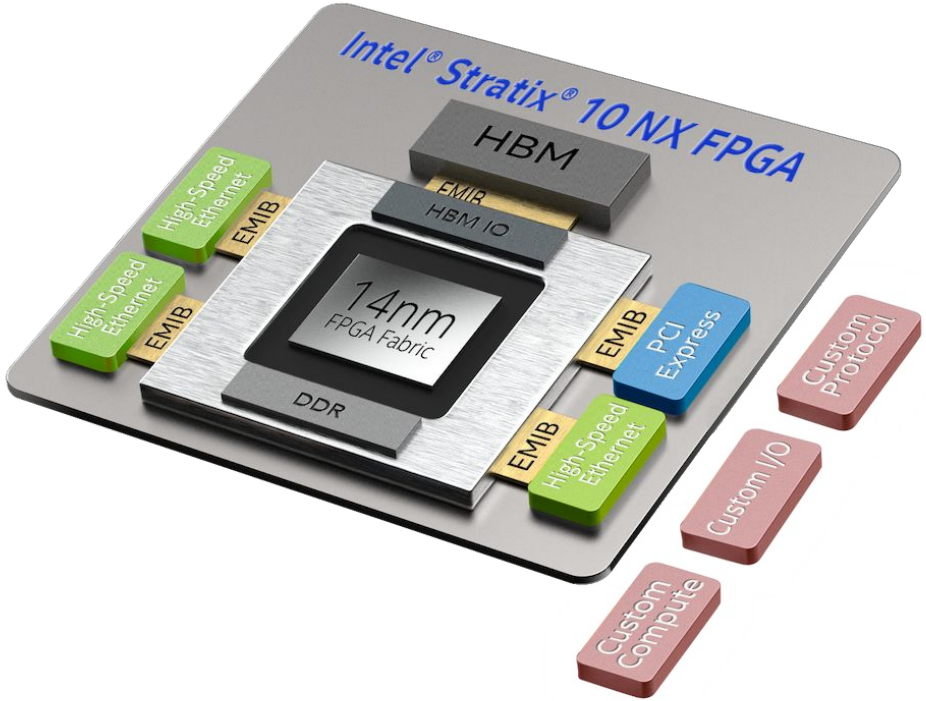
Use cases

- **HLS:** Often preferred for algorithm development, data flow designs, and when a software prototype exists that needs to be converted into hardware. => hardware acceleration
- **SystemVerilog/VHDL:** Used for more traditional hardware design, where control over implementation details and optimizations is critical. => chip design

Applications

- Historically present in embedded devices
- Small devices with dedicated functions
- Applications:
  - Consumer Automotive
  - Measurement
  - Communications
  - Military & Industrial
  - Computer & Storage
  - Etc …

FPGA-based HPC accelerators



Mapping program to FPGA

Example provided by Altera (Intel)

CPU instructions

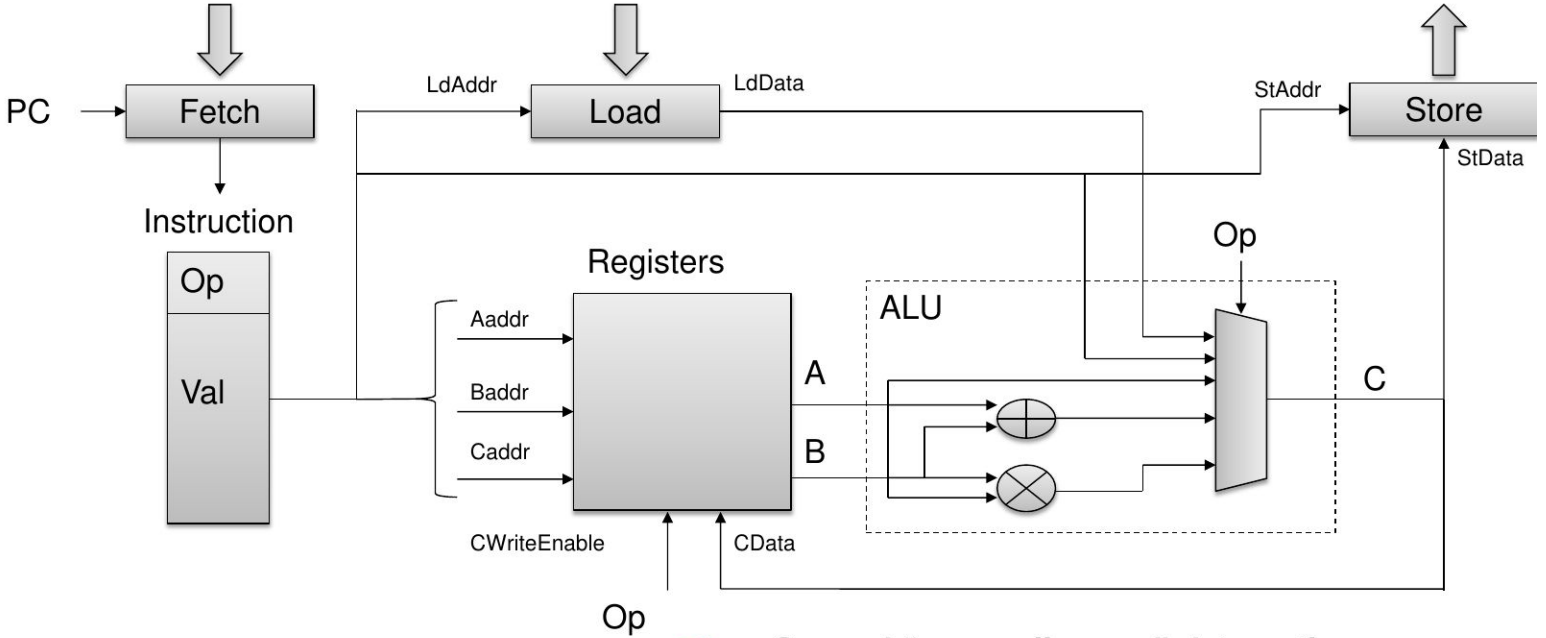
High-level code

```
Mem[100] += 42 * Mem[101]
```

CPU instructions

```
R0 ← Load Mem[100]
R1 ← Load Mem[101]
R2 ← Load #42
R2 ← Mul R1, R2
R0 ← Add R2, R0
Store R0 → Mem[100]
```

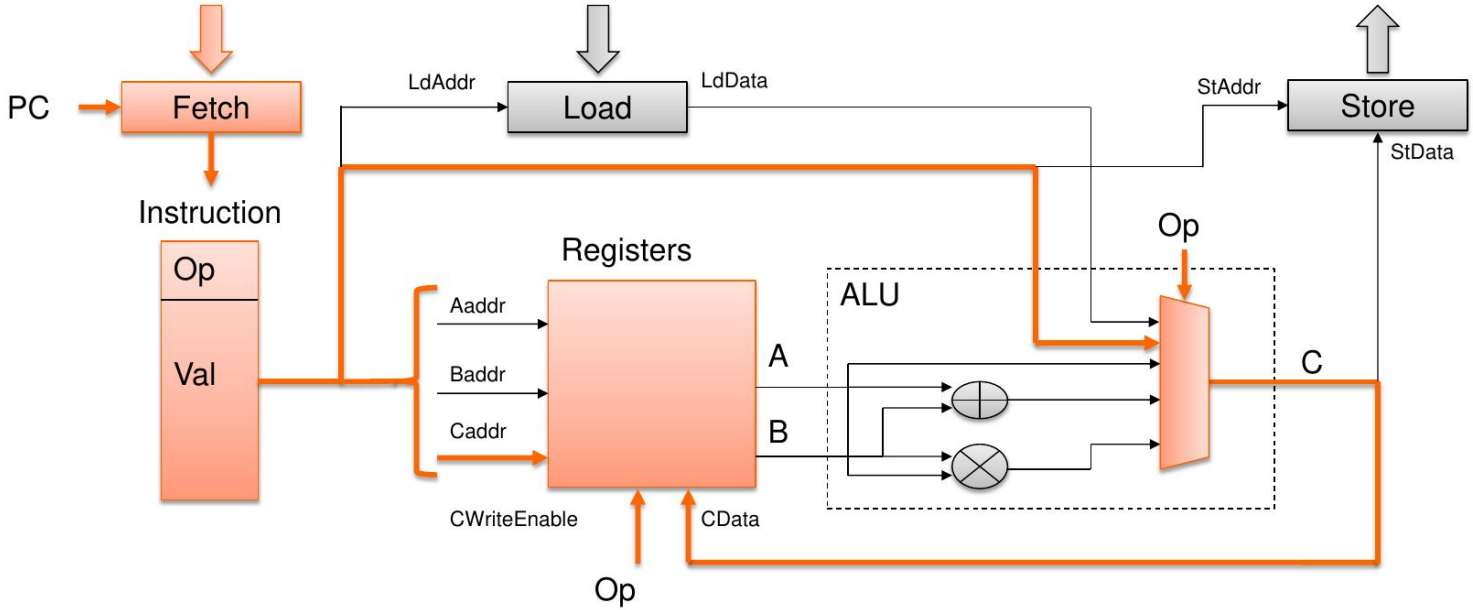
Example of simple CPU



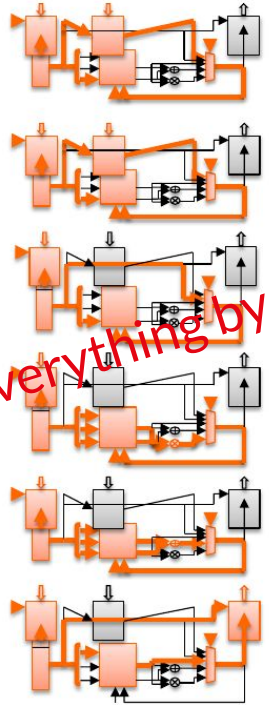
Fixed and general architecture:

- **General "cover-all-cases" data-paths**
- **Fixed data-widths**
- **Fixed operations**

Load constant value into register

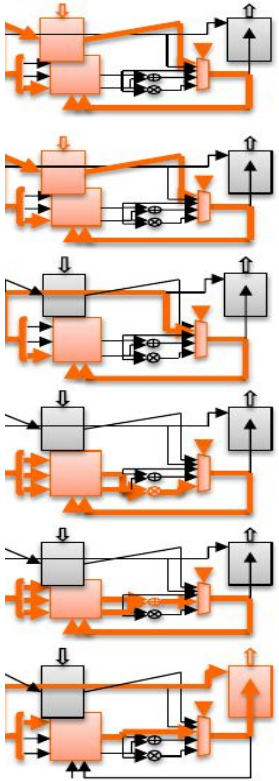


CPU activity, step by step

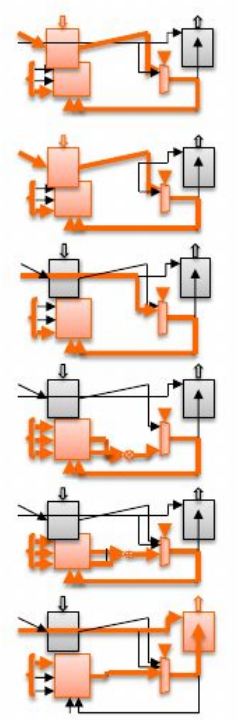


In FPGA, we specialize everything by unrolling the hardware

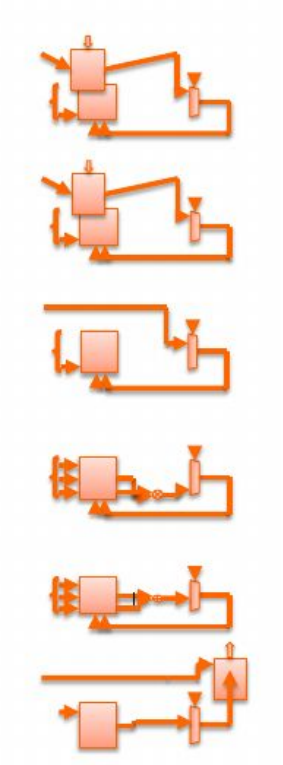
Remove Fetch operation



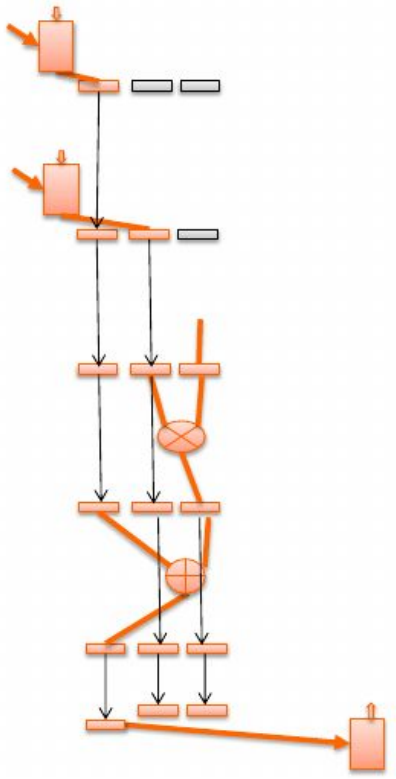
Remove unused ALUs



Remove unused Load / Store

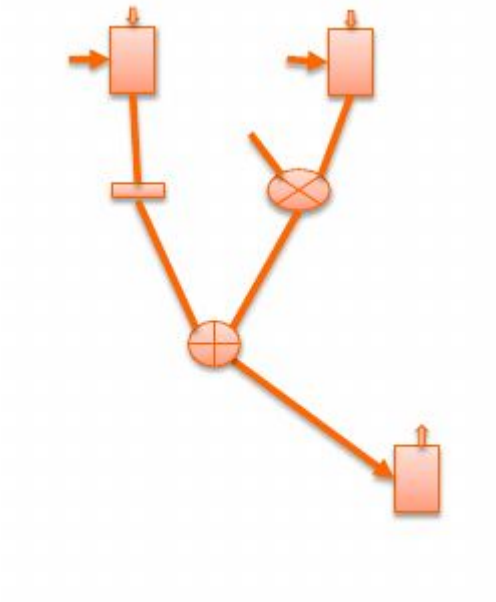


Remove unused Load / Store



- Removed fetch
- Removed unused ALUs
- Removed unused Load/Store
- Connect elements and propagate state

Simplify workflow



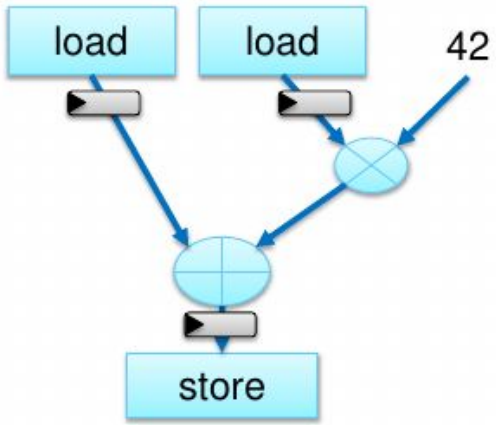
- Removed fetch
- Removed unused ALUs
- Removed unused Load/Store
- Connect elements and propagate state
- Schedule and simplify workflow

Custom data-path for the program

High-level code

```
Mem[100] += 42 * Mem[101]
```

Custom data-path



Build exactly what you need:

Operations

Data widths

Memory size & configuration

Efficiency:

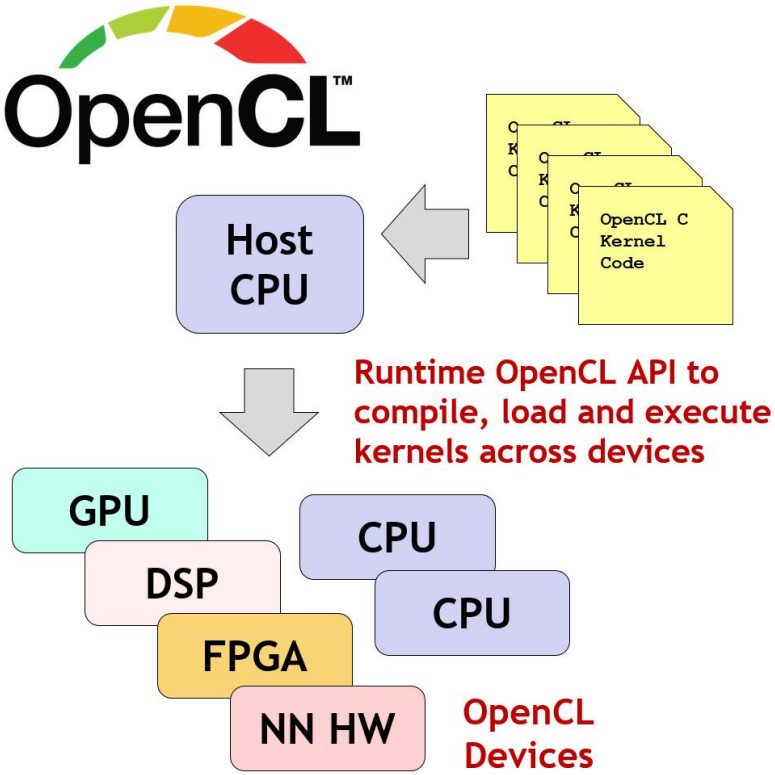
Throughput / Latency / Power

Models for Heterogeneous Computing

OpenCL : Low-level Heterogeneous Parallel Programing

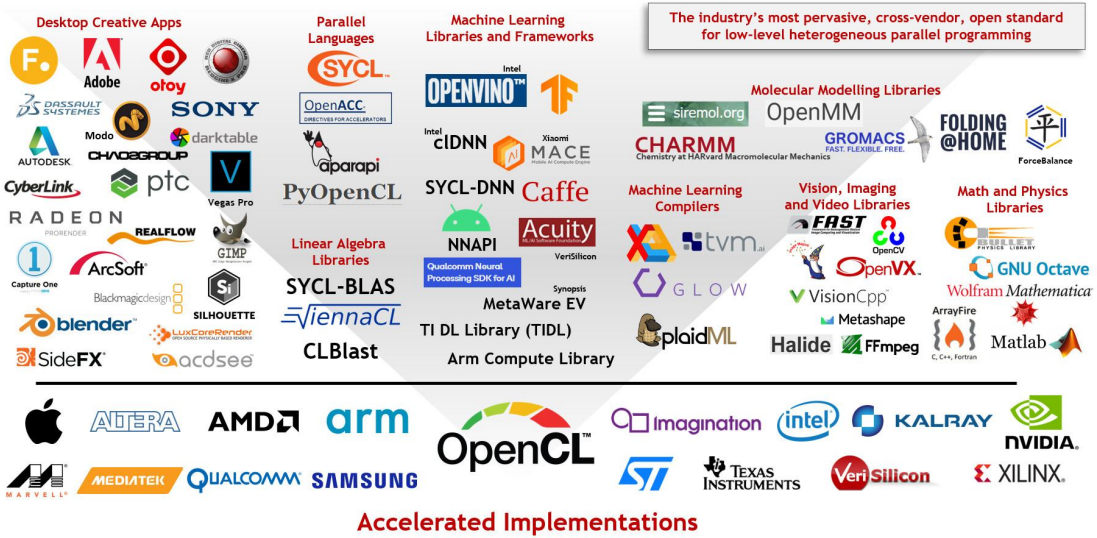
What is OpenCL

- OpenCL (Open Computing Language)
- Framework dedicated to heterogeneous computing:
  - Host API and Kernel language
  - Low-level programming style
  - Use for Hardware Acceleration (not only FPGA)
- Open and royalty-free
  - Initially developed by Apple and technical teams (AMD, IBM
  - First release in August, 2009
  - Maintained by Khronos
  - Khronos also maintain other standard
  - (e.g., OpenGL, SYCL, EGL, etc . . .)
  - Official website for the OpenCL standard
  - OpenCL guide: a github repository

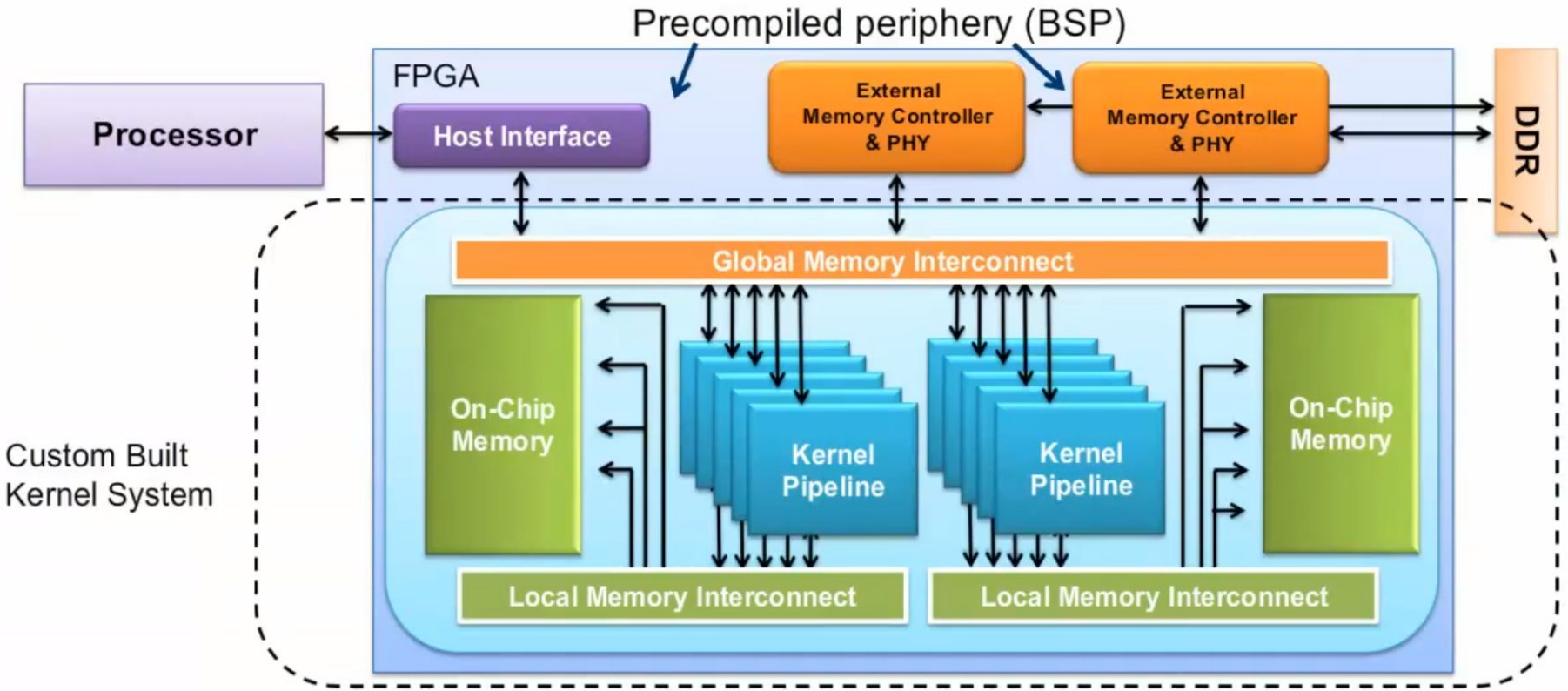


Adopters

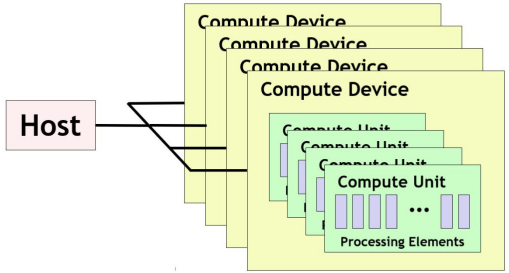
- OpenCL is widely used throughout the industry
- Many silicon vendors ship OpenCL with their processors, including GPUs, DSPs and FPGAs
- Intel is a strong contributor



FPGA architecture for OpenCL implementation

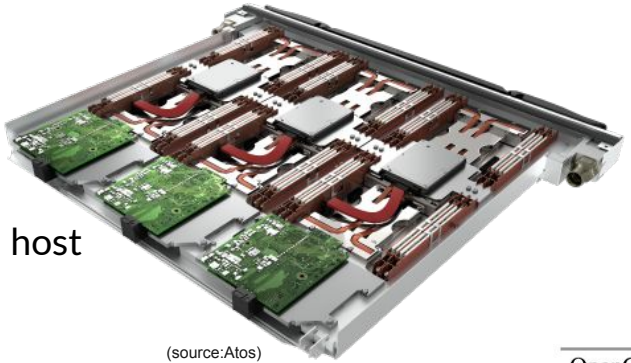


Platform Model



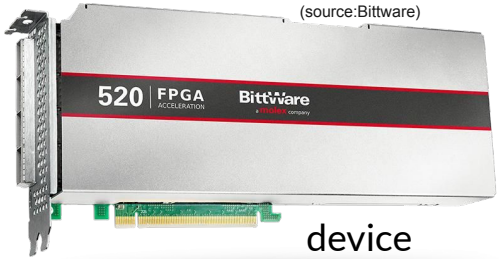
(source:Khronos)

- Abstract Hardware Model
- The platform consists of a **single host** and **multiple devices**
- Multiple platforms can coexist on a system but they are generally isolated
- There is a one-to-one mapping between the platform and a vendor provided SDK



(source:Atos)

SDK

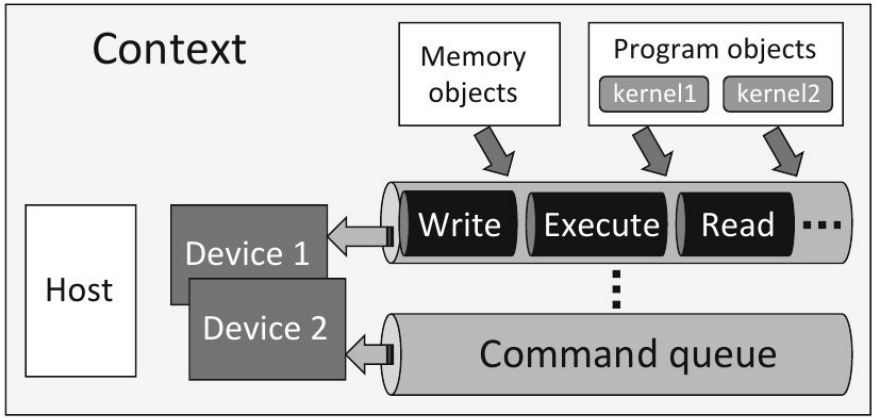


(source:Bittware)

OpenCL platform	SDK version
CPU (host) and CPU (device)	Intel SDK for OpenCL applications
CPU (host) and GPU (device)	Nvidia SDK for OpenCL
CPU (host) and GPU (device)	AMD APP SDK 3.0 for 64-bit Linux
CPU (host) and FPGA (device)	Intel FPGA SDK for OpenCL

Execution Model

- Execution Model:
  - Define how host and devices communicate
  - An OpenCL context is created with one or more devices
  - Provides an environment for host-device interaction

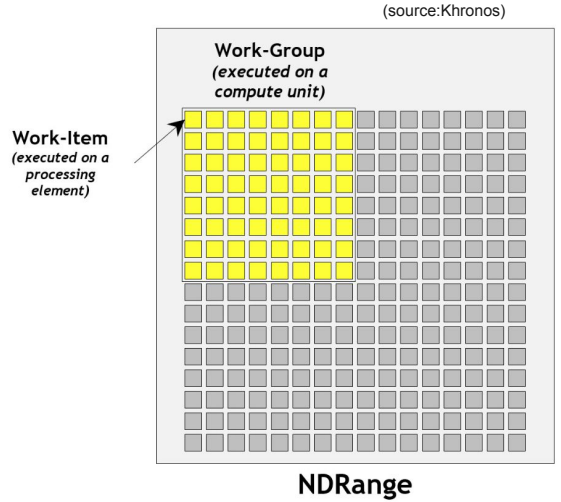


(source:Design of FPGA-Based Computing Systems with OpenCL)

Kernel Programming Model

- Kernels are functions executed on an OpenCL device
- The execution unit is the **work-item**
- Work-items are organized into **work-groups**
- Collection of work-items is called a **NDRange**, i.e., a multidimensional grid (max N=3)
- Sizes of NDRange and work-groups are specified by the host program
- Work-item identified by its global work-item ID and local work-item ID

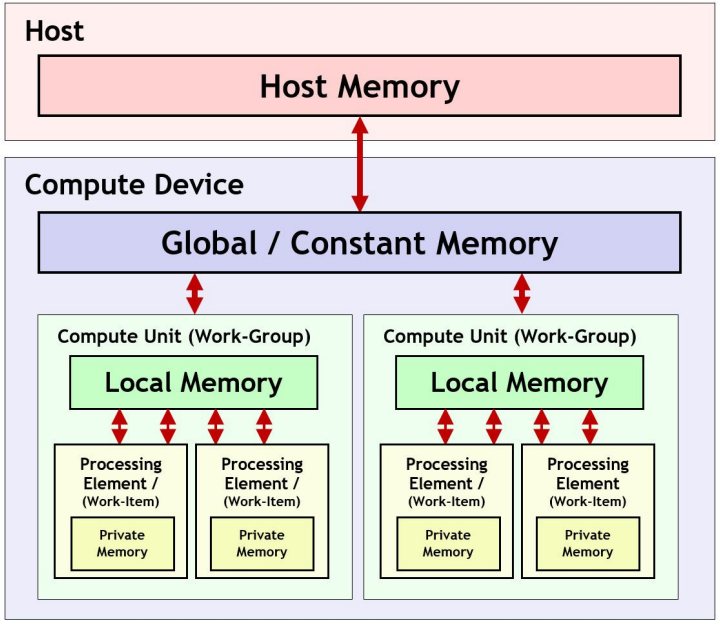
(source:Khronos)



Memory Model

- **Host memory** is accessible only to the host
- Data should be transferred from the host to the global memory of the device
- **Constant memory** is a read-only memory for the device
- **Local memory** belongs to a particular work-group
- **Private memory** belongs to a work-item

(source:Khronos)



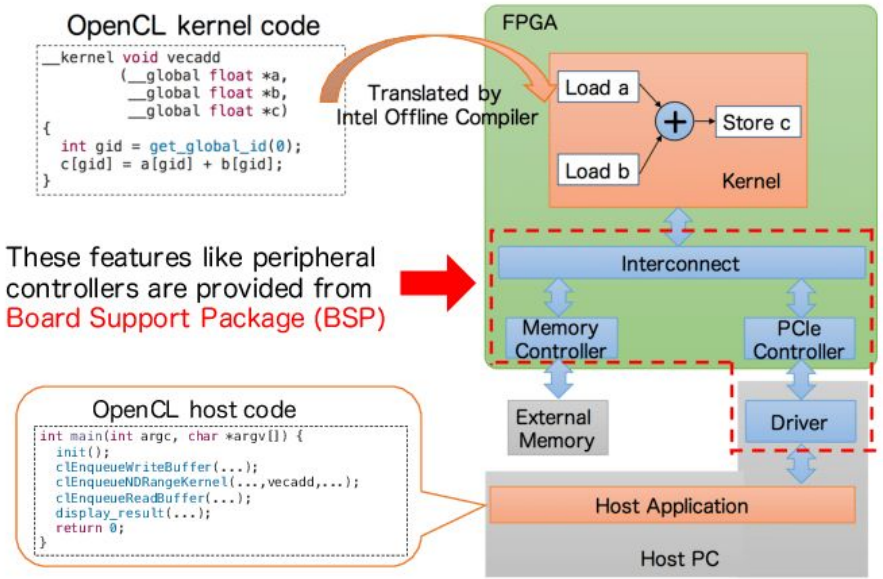
Why OpenCL ?

- Advantages:
  - C-like programming
  - Explicit parallelism → you code parallel kernels
  - Support for I/O controllers (e.g., memory PCIe, DMA)
    - HDL designers need to design everything from scratch
  - Require less hardware knowledge as OpenCL is more abstract
  - Compatible and Re-usable on different type of FPGA
  - Design methodologies using C languages are more efficient than HDL-based ones

- Drawbacks:
  - Synthesis is time-consuming
  - No control on the hardware architecture
  - Cannot design a specific clock frequency
  - Difficult to control resource utilization

OpenCL paradigm

- Two programming sides:
  - Kernel code (*.cl) translated by Intel Offline Compiler
  - Host code (*.c, *.cpp) compiled with host compiler:
    - Intel
    - GCC
    - Etc …
- Board support package (BSP) contains:
  - logic and memory information, and also
  - I/O controllers such as DDR3 controller
  - PCI controller, etc
- BSP provided by vendors
  - Can provide your own BSP
  - Need a high-level of expertise
  - Intel provides documentation for it



(source:OpenCL-ready High Speed FPGA Network for Reconfigurable High Performance Computing)

OpenCL parallelism

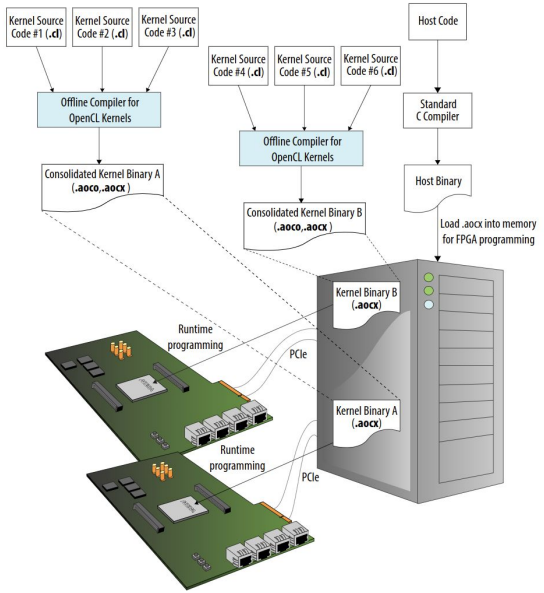
- Parallelism depends on the platform
  - FPGA ≠GPU
  - GPU starts multiple threads in //
  - FPGA intensively use pipelining
- Parallelism is set explicitly by the developer:
  - **Task parallelism** is obtained using the queues and event coordination
  - **Data parallelism (a.k.a SIMD)** is the simultaneous execution of parallel work-items (threads) on the same function across the elements of a dataset
  - **Loop pipeline parallelism** is achieved when the offline compiler analyzes dependencies between iterations of a loop and is able to pipeline each iteration for acceleration
- Data management
  - Explicit
  - Managed by the programmer
  - Up to the programmer to check memory and bandwidth efficiency

Compilation flow -- Intel FPGA SDK for OpenCL

- Kernels are compiled using an offline compilers (AOC for Intel)
- AOC executes the following tasks:
  - Parsing the OpenCL kernel source code
  - Circuit performance analysis
  - Synthesis or hardware compilation

- Kernels are first translated to a **aoco** object file
  - representing the hardware system

- A **aocx** executable file is finally created
  - use to program the FPGA



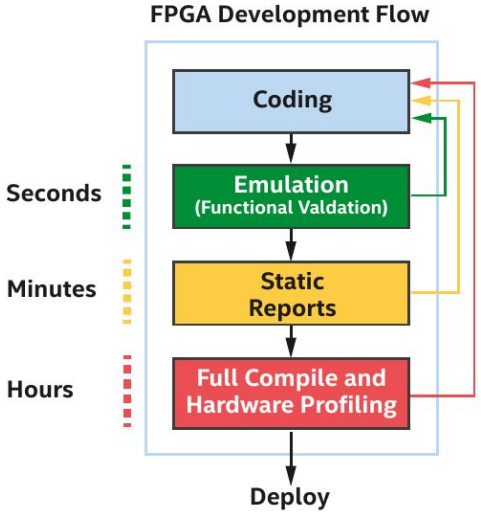
(source:Intel)

FPGA OpenCL kernels -- configuration time

- Hardware synthesis can be very long
- Emulation is a practical way of testing your OpenCL kernels

Design properties	Estimated time	Estimated memory
Low resource utilization (<10% in Kernel System) Simple memory interface (Global interconnect for < 10 global loads + stores) Loops with low to medium latency (<500 cycles)	2-4h	45 GB
Medium resource utilization (<40% ALUTs and FFs, and <60% RAMs and DSPs in Kernel System) Simple to medium memory interface (Global interconnect for < 20 global loads + stores) Loops with low to medium latency (<500 cycles)	8-12h	60-90 GB
High resource utilization (>50% ALUTs and FFs, or >70% RAMs and DSPs in Kernel System) Simple to medium memory interface (Global interconnect for < 20 global loads + stores) Loops with low to medium latency (<500 cycles)	12-20h	90-120 GB
Any resource utilization Simple to medium memory interface (Global interconnect for > 100 global loads + stores) or Loops with high to very high latency (>2000 cycles)	30-60h	120+ GB

(source: wikis.uni-paderborn.de)



(source:DPC++ book)

Why using FPGA OpenCL kernels

- While the long compilation time of FPGA designs is a genuine concern and can be a barrier in some contexts
- FPGAs offer in terms of customization, power efficiency, flexibility, and more may justify this trade-off in many situations.
- Once FPGA image has been compiled:
  - High Throughput for Fixed Functions that can be specialized and doesn't change frequently
  - Low Latency as they can process data in parallel without the overhead of a general-purpose processor.
  - Customization for hardware design tailored to specific tasks
  - Power efficiency due to less generated hardware

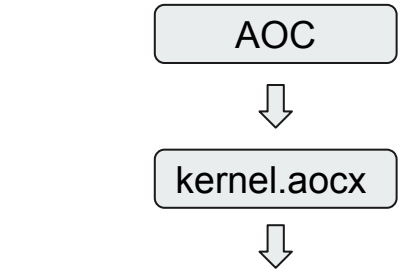
Compilation kernels

```
#define N (1024*1024)

__kernel void first_kernel ( __global const int * restrict k_din,
                             __global int * restrict k_dout )
{

        for(unsigned int i=0; i<N; i++)
          k_dout[i] = k_din[i] + 40;


}
```



- `aoc -list-boards`
  - List available boards within the current package

```
[u100057@mel3009 first_code]$ aoc -list-boards
Board list:
  p520_hpc_m210h_g3x16 (default)
     Board Package: /apps/USE/easybuild/staging/2022.1/software/520nmx/20.4
     Memories:      HBM0, HBM1, HBM2, HBM3, HBM4, HBM5, HBM6, HBM7, HBM8, HBM9, HBM10, HBM11, HBM12, HBM1
3, HBM14, HBM15, HBM16, HBM17, HBM18, HBM19, HBM20, HBM21, HBM22, HBM23, HBM24, HBM25, HBM26, HBM27, HBM2
8, HBM29, HBM30, HBM31

  p520_max_m210h_g3x16
     Board Package: /apps/USE/easybuild/staging/2022.1/software/520nmx/20.4
     Memories:      HBM0, HBM1, HBM2, HBM3, HBM4, HBM5, HBM6, HBM7, HBM8, HBM9, HBM10, HBM11, HBM12, HBM1
3, HBM14, HBM15, HBM16, HBM17, HBM18, HBM19, HBM20, HBM21, HBM22, HBM23, HBM24, HBM25, HBM26, HBM27, HBM2
8, HBM29, HBM30, HBM31
     Channels:      kernel_input_ch0, kernel_output_ch0, kernel_input_ch1, kernel_output_ch1, kernel_inpu
t_ch2, kernel_output_ch2, kernel_input_ch3, kernel_output_ch3
```

- `aoc -board=<board> <kernel file>`
  - Compile the kernel for a specific board
  - Generate the kernel hardware system
  - Call Intel Quartus Prime software to create the aocx file

```
[u100057@mel3009 first_code]$ aoc -board=p520_hpc_m210h_g3x16 first_kernel.cl
```

(source:Intel)

Compilation outputs files

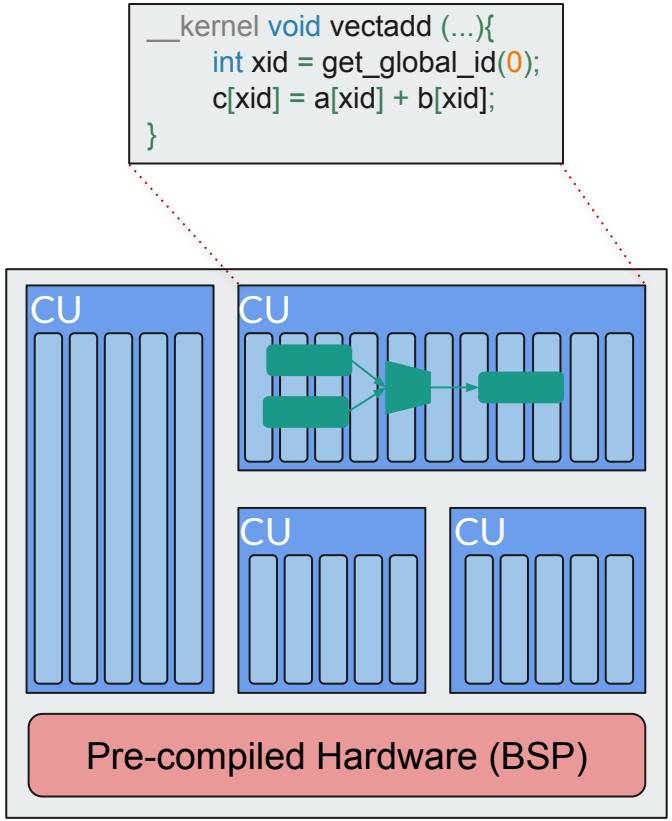
- `<kernel file>.aoco`
  - List available boards within the current package
- `<kernel file>.aocx`
  - Kernel executable file to program FPGA
- `<kernel file> folder`
  - `<kernel file> folder/reports/report.html`
    - Interactive HTML report
    - Static report showing optimization and architectural information
  - `<kernel file>.log`
    - Kernel compilation log
  - `<kernel file> folder/{*.tcl,*.v,*.qsf,*.qsys, etc …}`
    - Numerous intermediate files
    - Generated by Intel Quartus Prime

```
[u100057@mel3009 first_code]$ ls first_kernel
acds_version_rom.hex     first_kernel_sys.v          quartuserr.tmp
acds_version_rom.mif     first_kernel.v              quartus.ini
adjust_floorplan.py      flat.qsf                    quartus_sh_compile.log
automigration.rpt        hbm_bottom                  quartus_version.id
base.aocx                hbm_bottom.qsys             reports
base_bak.qar             hbm_logic_lock.qsf          root_partition.qarlog
base_bak.qarlog          hbm_top                     root_partition.qdb
base.kernel.pmsf         hbm_top.qsys                scripts
base.pof                 hw_iface.iipx               sw_iface.iipx
base.qar                 iface.ipx                   sys_description.hex
base.qarlog              ip                          sys_description.legend.txt
base.qdb                 ip_include.tcl              sys_description.txt
base.qsf                 kernel_hdl                  tmp-clearbox
base.sdc                 kernel_pll_refclk_freq.txt  top.fit.finalize.rpt
base.sof                 kernel_report.tcl           top.fit.place.rpt
base.static.msf          kernel_system.qip           top.fit.plan.rpt
board                    kernel_system.v             top.fit.retime.rpt
board.qsys               llc.err                     top.fit.route.rpt
board_spec.xml           opencl_bsp_ip.qsf           top.flow.rpt
compiler_metrics.out     opencl.ipx                  top.pin
compile_script.tcl       out_directory_tmp.txt.tmp   top_post.sdc
control_aer.sh           pr_base.id                  top.qpf
cra_ring_rom.params      pr_region_logic_lock.qsf    top.qsf
device_opn.tcl           pwrmgt.qsf                  top.sdc
device.tcl               qar_info.json               top.syn.rpt
first_kernel.bc.xml      qdb                         top.syn.summary
first_kernel.log         qdb.qar                     top.v
first_kernel_sys_hw.tcl  qdb.qarlog
```

(source:Intel)

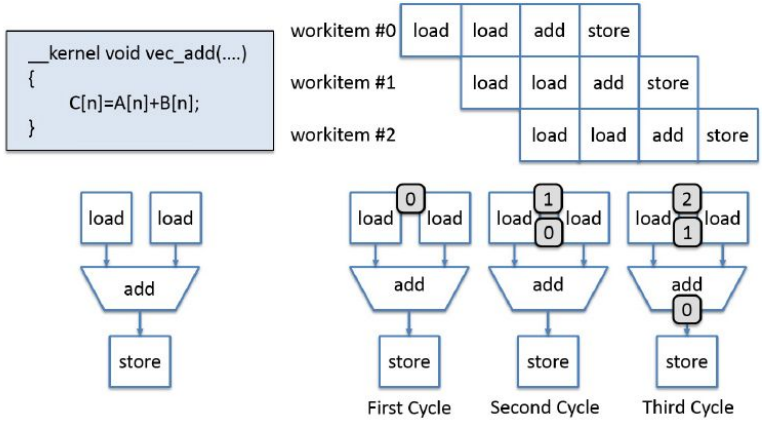
Kernel as Custom Hardware

- Each kernel become a **Compute Unit (CU)**
- Some element have been precompiled
  - Provided by the BSP
  - Ex: Memory Controller, I/O controller, etc …
- C operation in the kernel are converted to circuits:
  - Use HDL existing library and ip cores
  - Create LOAD/STORE units for read/write operations
  - Connections of all elements to follow the dataflow
  - Elements or full circuit can be replicated multiple times
- Memory:
  - Global → DDR, QDR
  - Local → On-chip memory
  - Private → BRAM, register



Pipelining Parallelism

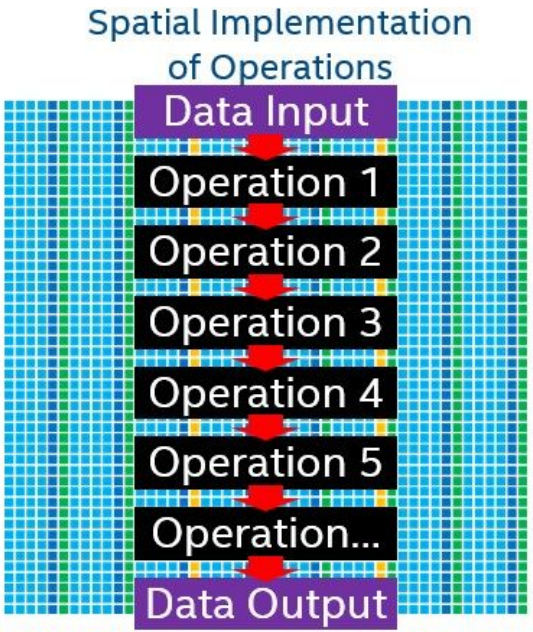
- FPGA Parallelism is clearly very different from GPU
- GPU are better for data parallelism
  - Independent tasks with almost no dependencies
- FPGA takes advantage of pipelining parallelism
  - Create a deep pipeline of the kernel
  - Stages can be executed concurrently by work-items
- Two main approaches for FPGA:
  - **NDRange kernel** is executed by multiple work-items
  - **Single work-item kernel** where loop-iterations are computed in different pipeline stages
- Use **NDRange**:
  - when there are no data sharing between work-items
  - when kernel will be used on GPU and FPGA platforms
- Use **single-work item:**
  - when you have data dependencies
  - When you want to port CPU code to FPGA



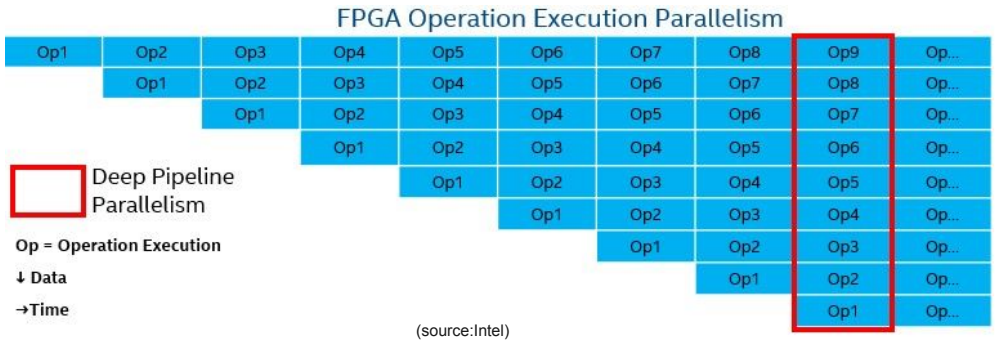
(source:OpenCL-Based Design of an FPGA Accelerator for Phase-Based Correspondence Matching)

NDRange to FPGA

- Replicating hardware for each work-item is suboptimal
- Need to take into account that FPGA are efficient for pipeline
- Why:
  - FPGA are different from GPU (lots of thread started at the same time)
  - Impossible to replicate a million time of kernel for a FPGA card
  - This is wasteful as you can be sure that all stages of all pipelines won't be busy
  - How many work-items do you finally need ?



(source:Intel)

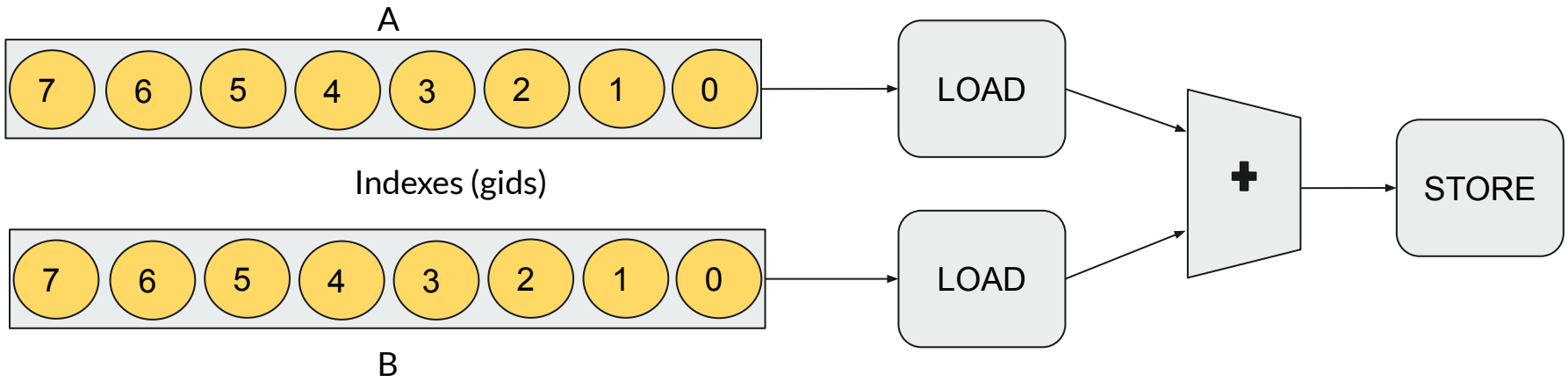


(source:Intel)

Pipelining example for NDRange

- Vector addition
- 8 work-items
- Each clock cycle, all parts of the pipeline process different items

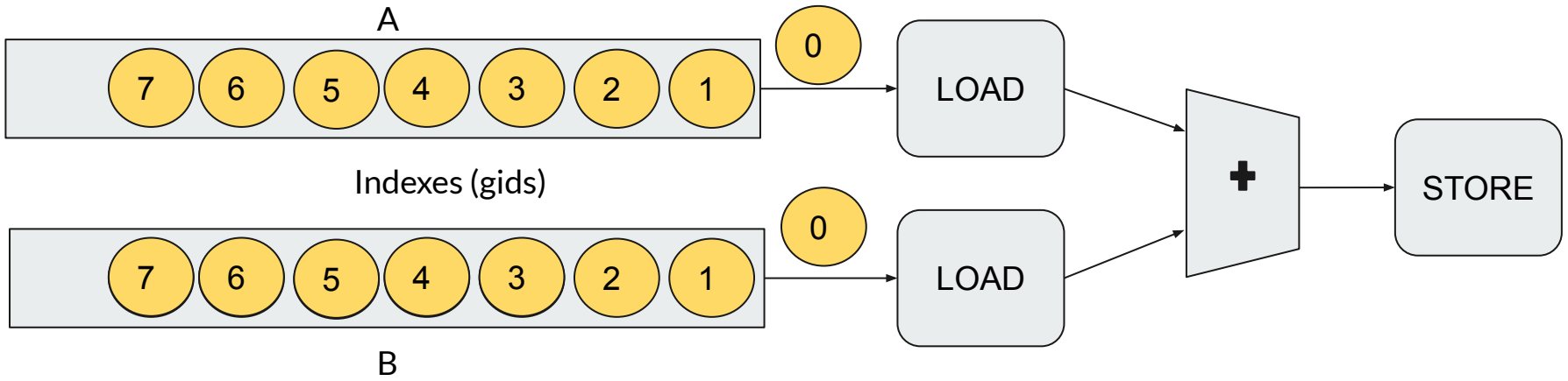
```
__kernel void vecadd(...){
   int gid = get_global_id(0);
   C[gid] = A[gid] + B[gid];
}
```



Pipelining example for NDRange

- Vector addition
- 8 work-items
- Each clock cycle, all parts of the pipeline process different items

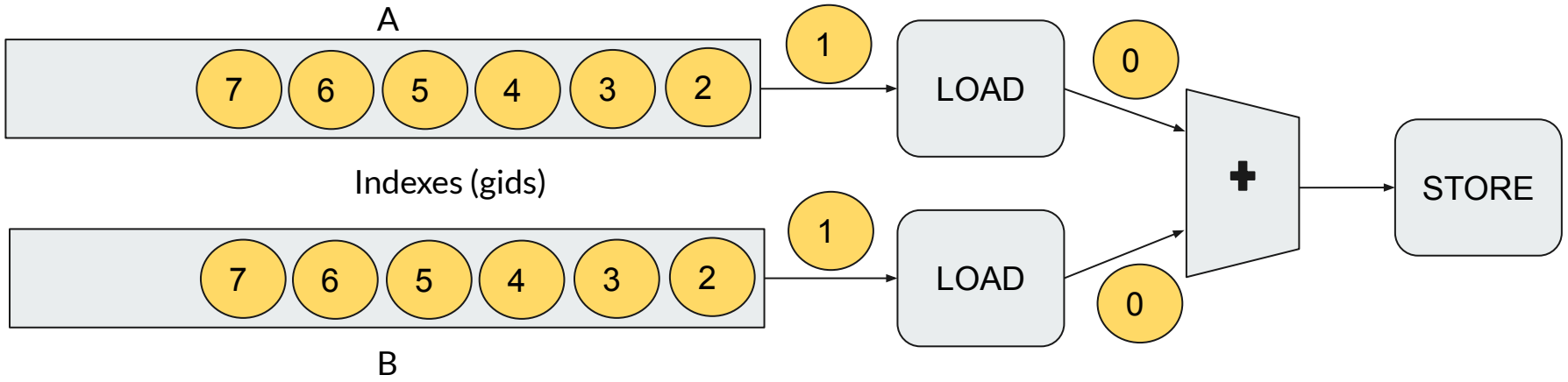
```
__kernel void vecadd(...){
    int gid = get_global_id(0);
    C[gid] = A[gid] + B[gid];
}
```



Pipelining example for NDRange

- Vector addition
- 8 work-items
- Each clock cycle, all parts of the pipeline process different items

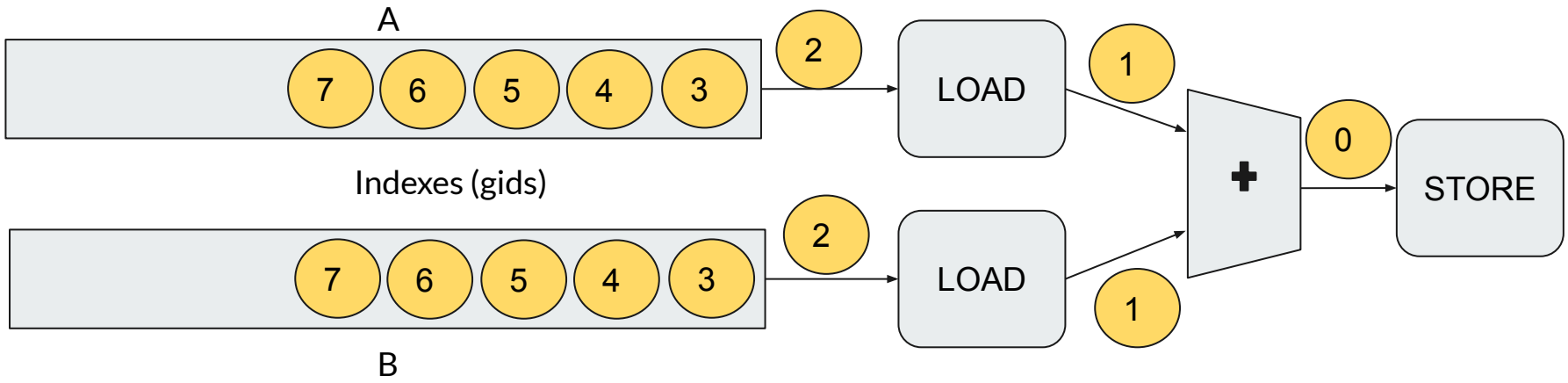
```
__kernel void vecadd(...){
    int gid = get_global_id(0);
    C[gid] = A[gid] + B[gid];
}
```



Pipelining example for NDRange

- Vector addition
- 8 work-items
- Each clock cycle, all parts of the pipeline process different items

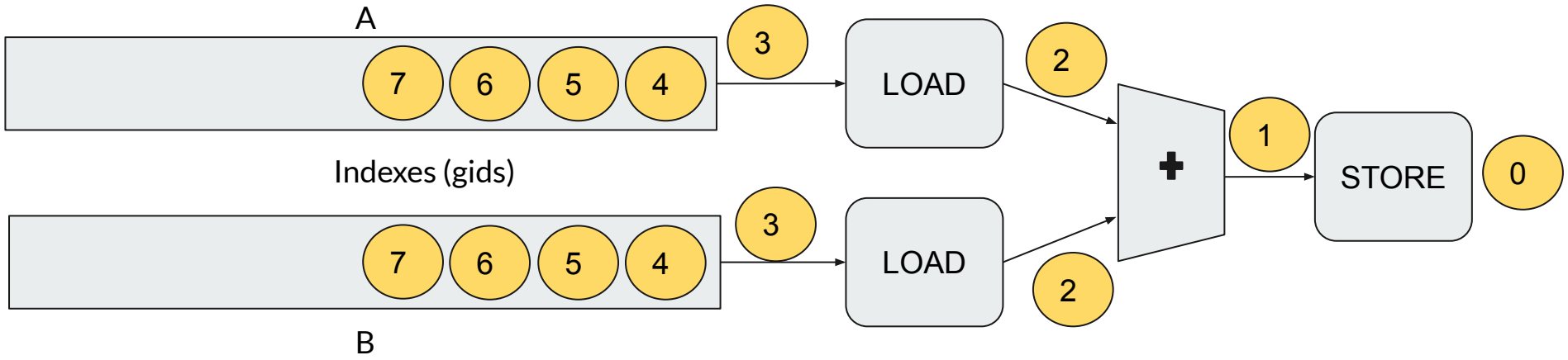
```
__kernel void vecadd(...){
    int gid = get_global_id(0);
    C[gid] = A[gid] + B[gid];
}
```



Pipelining example for NDRange

- Vector addition
- 8 work-items
- Each clock cycle, all parts of the pipeline process different items

```
__kernel void vecadd(...){
   int gid = get_global_id(0);
   C[gid] = A[gid] + B[gid];
}
```



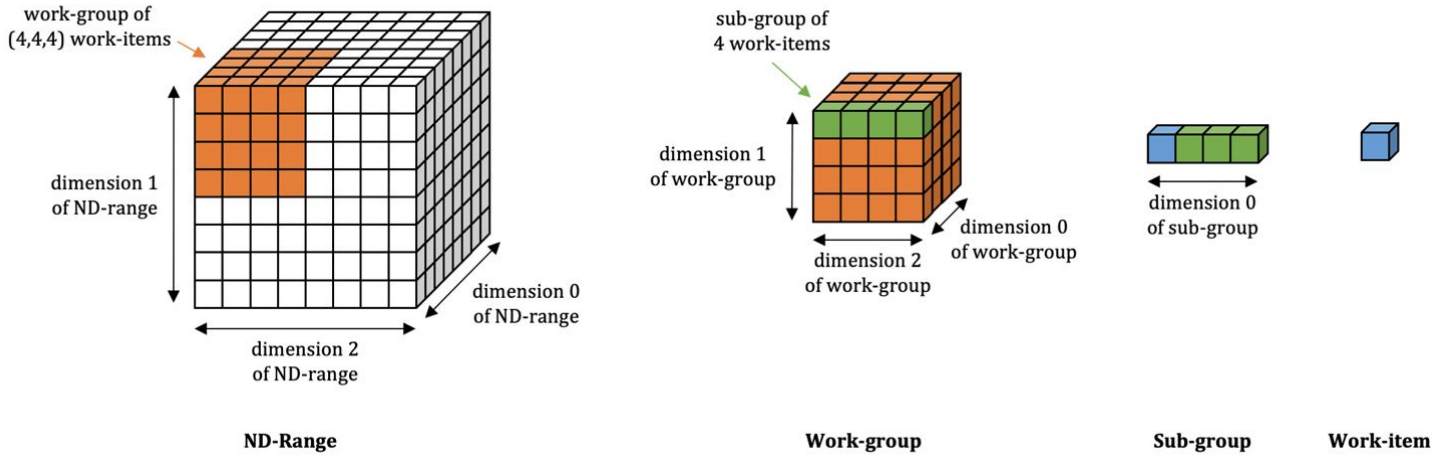
NDRange

- Useful when you can create a deep pipeline representation
- Each clock you send an new input data which is processed by the pipeline
- Fine-grained parallelism
- A kernel with a thousand of stages will concurrently execute a thousand of work-items
- Best suited for applications with independent loops (no data dependencies)
- Barriers should be used to avoid race conditions and have an additional hardware cost

```
int gid = get_global_id(0);
C[gid] = A[gid] + B[gid];
barrier();
C[N-gid] = C[M-xid] + A[gid]
```

Single-Work Item

- Atomic element of a NDrange ⇔ task
- Kernel executing on a compute unit by exactly one work-item

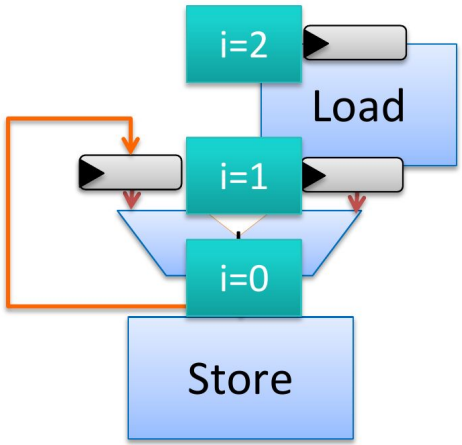


Single-Work Item

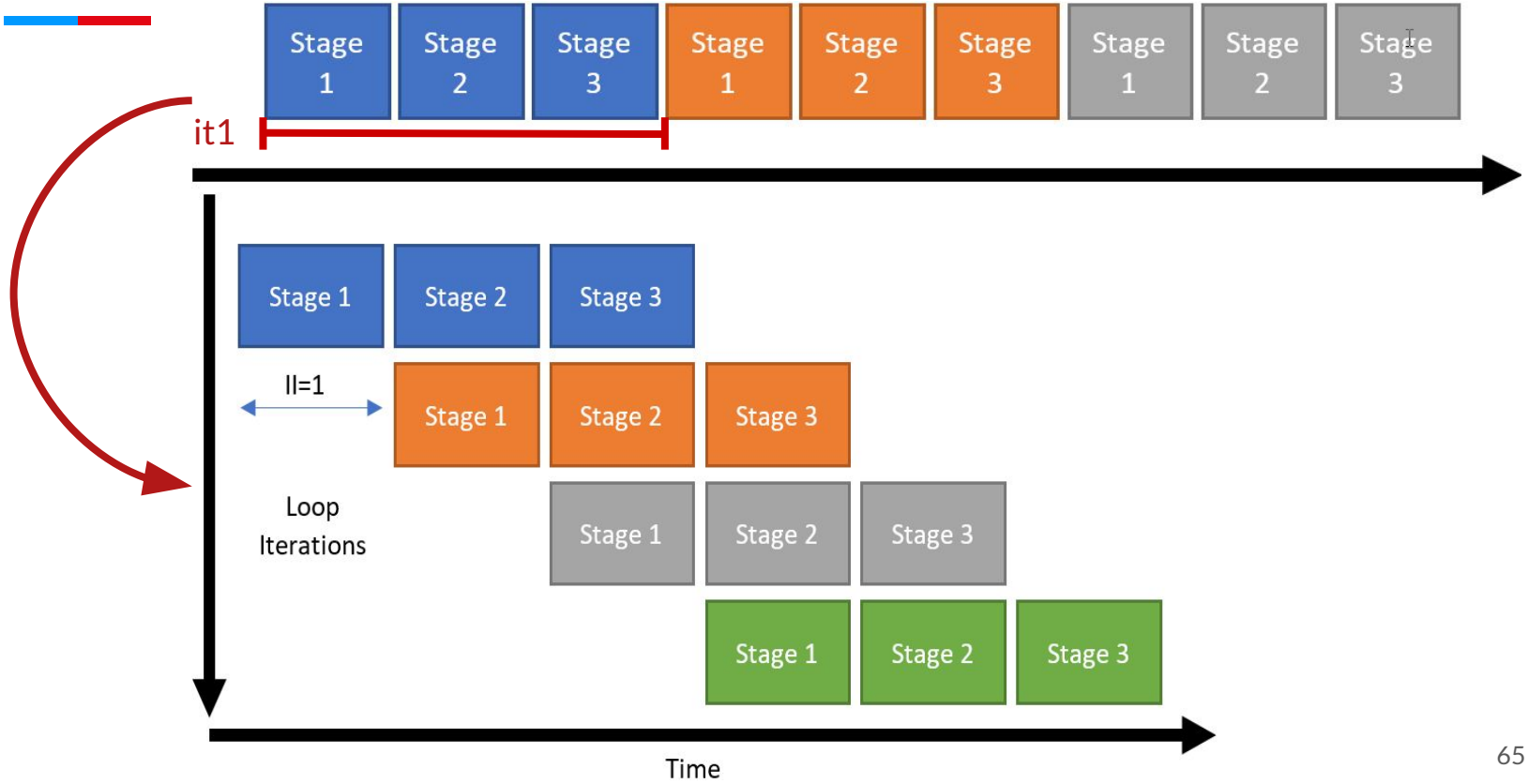
- Implementation of a single-work item is very close to classical C program
- A single-work contains loops which have *multiple loop-iterations*
- Loop pipelining execute the multiple loop-iterations in different pipeline stages in parallel
-

```
#define SIZE 1024
__kernel void vectoradd_single_work_item ( __global const int *A,
                                           __global const int *B,
                                           __global int *C)

{
for(int i=1, i<SIZE; i++)
   C[i] = A[i-1] + B[i];
}
```



Single-Work Item (trivial example)



Single-Work Item

- The offline compiler analyze each iteration of the loop
- It detects any dependencies
- Schedule and launch operations a.s.a.p

```
float array[M];
for (unsigned int i = 0; i < N; i++)
{
        for (unsigned int j = 0; j < M-1; j++)
            array[j] = array[j+1];
        array[M-1] = a[i];

        for( unsigned int j = 0; j < M; j++){
            answer[i] += array[j] + coefs[j];
}
```

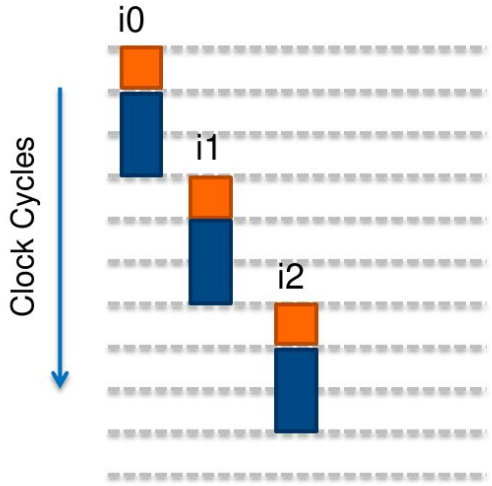
Shift register array

Reduction on array

(source:Intel)

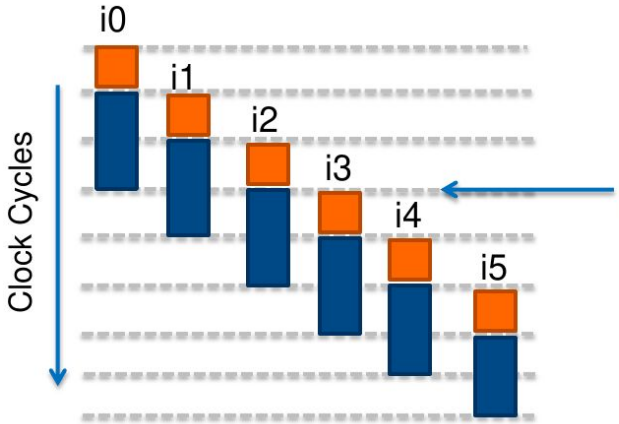
Single-Work Item

No Loop Pipelining



No Overlap of Iterations!

With Loop Pipelining



Looks almost like multi-threaded execution!

Finishes Faster because Iterations Are Overlapped

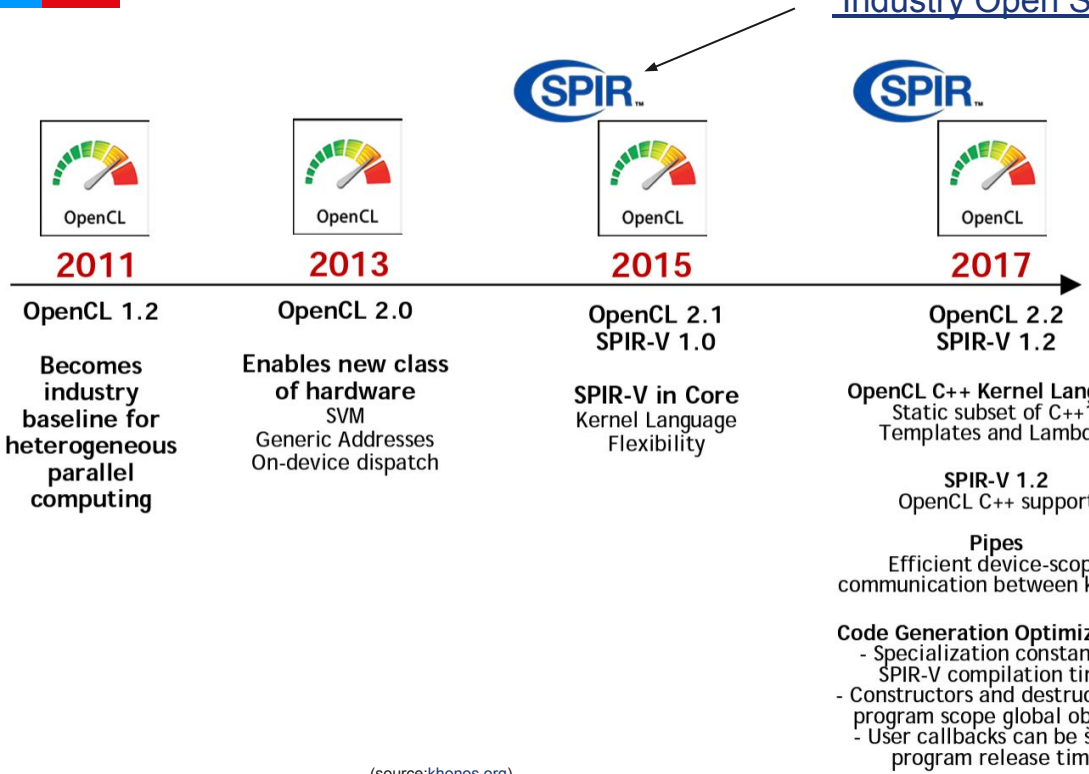
(source:Intel)

SYCL : High-level Heterogeneous Parallel Programing

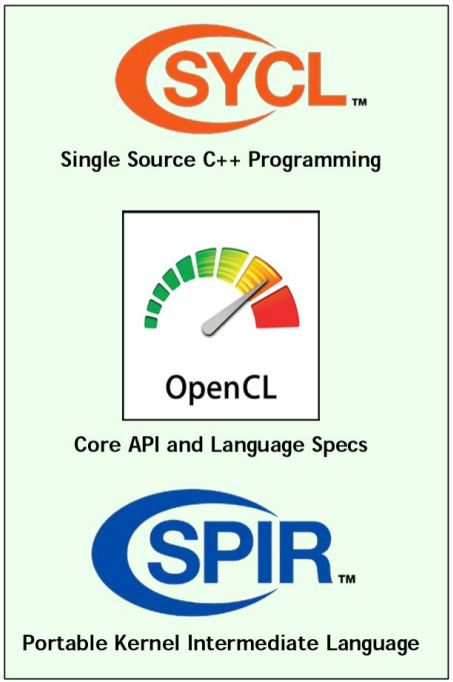
UNIVERSITÉ DU LUXEMBOURG

Why SYCL ?

Industry Open Standard Intermediate Language



(source:khonos.org)



What is SYCL ?

- High-level C++ abstraction layer for OpenCL
- Full coverage for all OpenCL features
- Interop to enable existing OpenCL code with SYCL
- Single-source compilation
- Automatic scheduling of data movement

Developer Choice

The development of the two specifications are aligned so code can be easily shared between the two approaches

C++ Kernel Language

Low Level Control

'GPGPU'-style separation of device-side kernel source code and host code

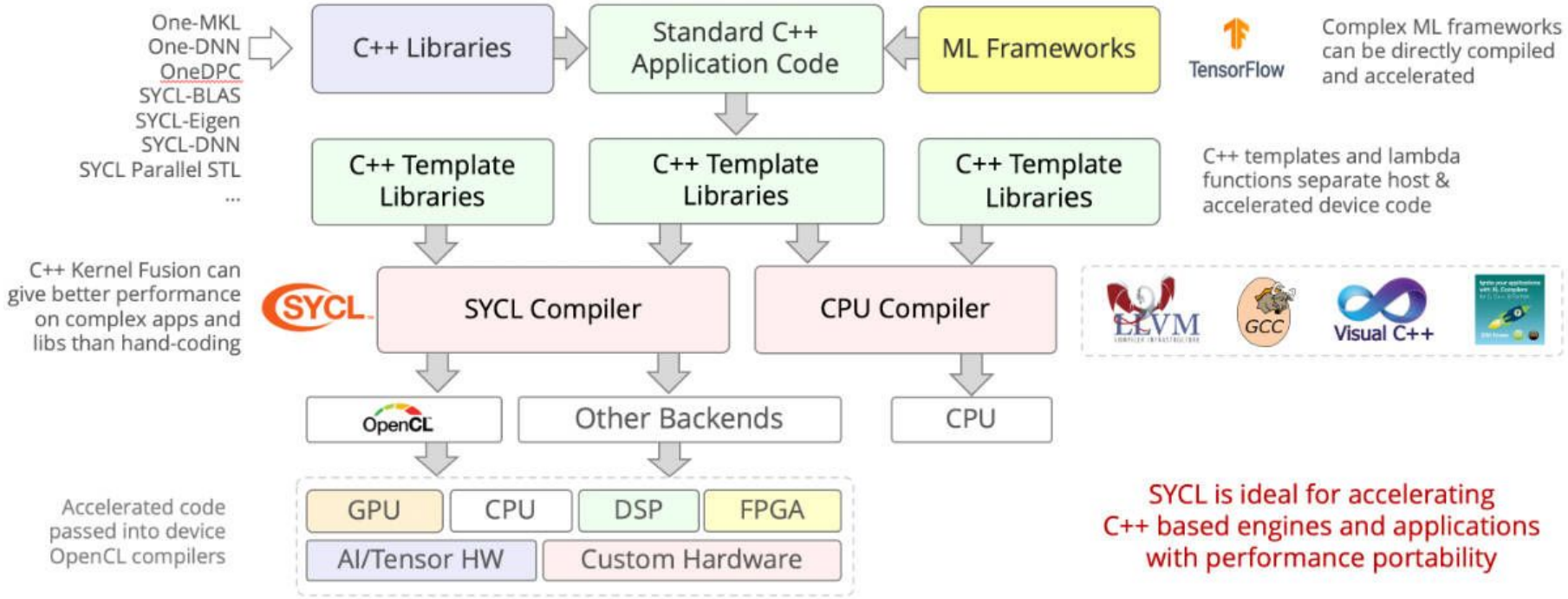
Single-source C++

Programmer Familiarity

Approach also taken by C++ AMP and OpenMP

(source:khonos.org)

What is SYCL ?



(source:khonos.org)

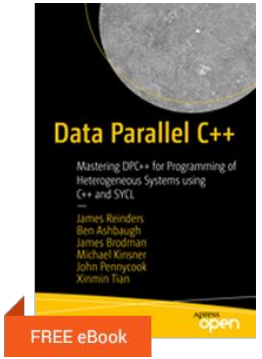
SYCL code example

```
// Copyright (C) 2020 Intel Corporation
// SPDX-License-Identifier: MIT
#include <sycl/sycl.hpp>
#include <iostream>
using namespace sycl;
const std::string secret {"Ifmmp-!xpsme\"\012J(n!tpssz-!Ebwf/!" "J(n!bgsbje!J!dbo(u!ep!uibu/!.!IBM \01"
};
const auto sz = secret.size();
int main() {
  queue Q;
  char *result = malloc_shared<char>(sz, Q);
  std::memcpy(result,secret.data(),sz);
  Q.parallel_for(sz,[=](auto& i) {
    result[i] -= 1;
  }).wait();
  std::cout << result << "\n";
  return 0;
}
```

(source:DPC++ book)

SYCL resources

- Data Parallel C++: Mastering DPC++ for Programming of Heterogeneous Systems using C++ and SYCL
- SYCL academy
- ENCCS Heterogeneous programming with SYCL
- More resources & tutorials on the Khronos website



Libraries/Frameworks with SYCL

- SYCL-BLAS - An open source implementation of BLAS using the SYCL open standard for acceleration on OpenCL devices
- SYCL-DNN - An open source neural network operations library written using the SYCL API
- SYCL-ML - An open source C++ library implementing classical machine learning algorithms in SYCL
- SYCL-ParallelSTL - An open source Parallel STL implementation
- Tensorflow - An implementation of TensorFlow using SYCL

SYCL implementation

- ComputeCpp - SYCL v1.2.1 conformant implementation by Codeplay Software
- Intel LLVM SYCL oneAPI DPC++ - an open source implementation of SYCL that is being contributed to the LLVM project
- hipSYCL - an open source implementation of SYCL over NVIDIA CUDA and AMD HIP
- triSYCL - an open-source implementation led by Xilinx

Intel LLVM SYCL oneAPI DPC++

Let’s go back to https://ekieffer.github.io/oneAPI-FPGA