

**ALL PROGRAMMABLE**

ANY MEDIA

5G

4K/8K

ANY STANDARD

ANY MACHINE

ANY NETWORK

5G Wireless • SDN/NFV • Video/Vision • ADAS • Industrial IoT • Cloud Computing



## P0367R0: Accessors — wrapper classes to qualify accesses

Ronan Keryell (Xilinx) & Joël Falcou (NumScale)

2016/06/25

# Outline

- 1 Heterogeneous architectures
- 2 Accessors
- 3 Conclusion

# Power wall & speed of light: the final frontier...

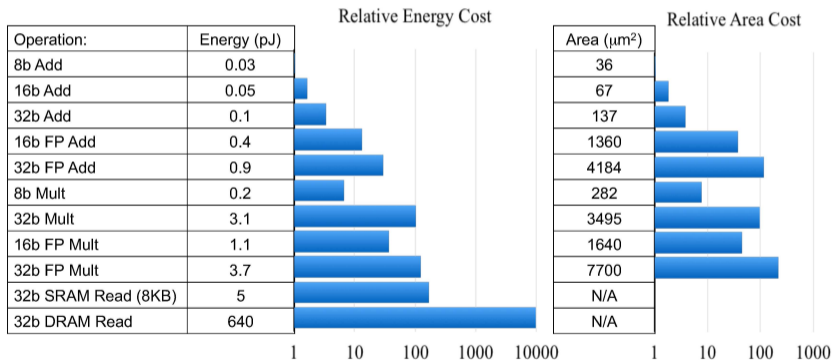
- Current physical limits
  - ▶ Power consumption
    - Cannot power-on all the transistors without melting (*dark silicon*)
    - Accessing memory consumes orders of magnitude more energy than a simple computation
    - Moving data inside a chip costs quite more than a computation
  - ▶ Speed of light
    - Accessing memory takes the time of  $10^4+$  CPU instructions
    - Even moving data across the chip (cache) is slow at 1+ GHz...

# (Rather old) 45nm technology characteristics

Tutorial on “*High-Performance Hardware for Machine Learning*”, William Dally at NIPS, December 7th, 2015

<https://media.nips.cc/Conferences/2015/tutorialslides/Dally-NIPS-Tutorial-2015.pdf>

## Cost of Operations



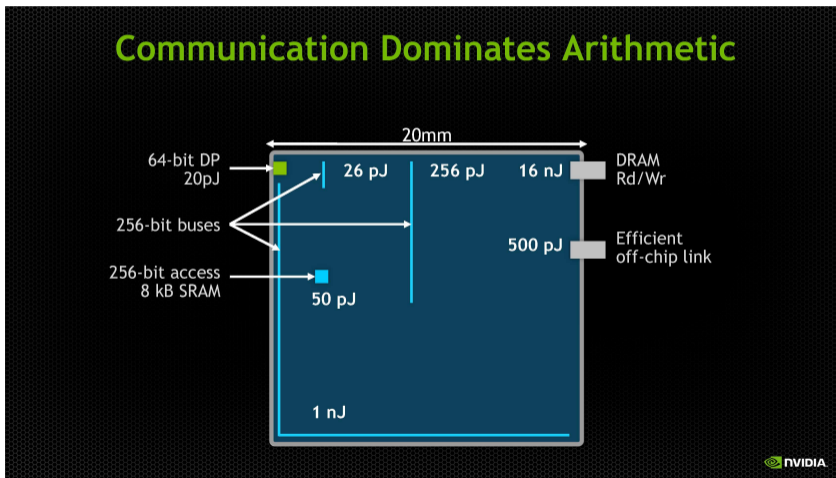
Energy numbers are from Mark Horowitz “Computing’s Energy Problem (and what we can do about it)”, ISSCC 2014

Area numbers are from synthesized result using Design Compiler under TSMC 45nm tech node. FP units used DesignWare Library.



# Space-time traveling

“Challenges for Future Computing Systems”, William J. Dally, January 19, 2015, HiPEAC 2015.

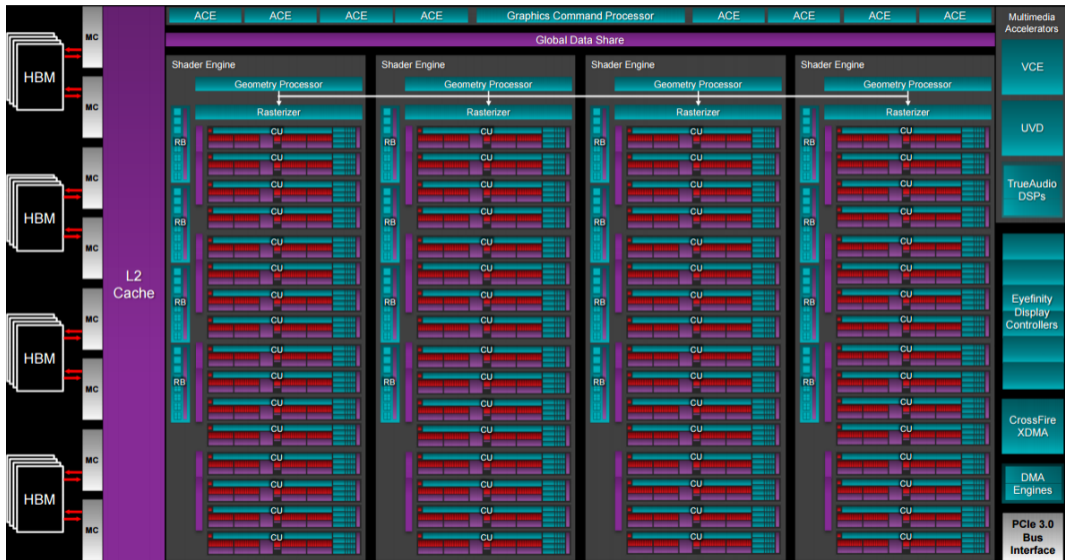
<http://www.cs.colostate.edu/~cs575d1/Sp2015/Lectures/Dally2015.pdf>



# Power wall & speed of light: implications

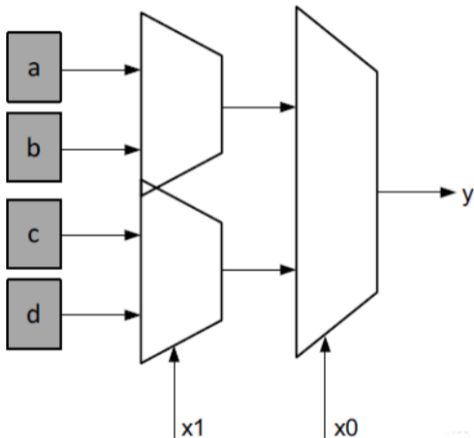
- Specialize architecture
- Use locality & hierarchy
- Massive parallelism
- NUMA & distributed memories
  - ▶  New memory address spaces
  - ▶  PiM (Processor-in-Memory), Near-Memory Processing...
- Power-on-demand only what is required

# From AMD Fiji XT GPU (2015)...

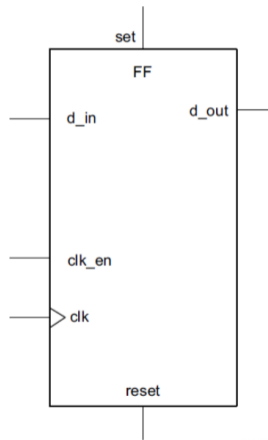


■ P0367R0: Accessors — wrapper classes to qualify accesses

# Basic architecture = Lookup Table + Flip-Flop storage + Interconnect

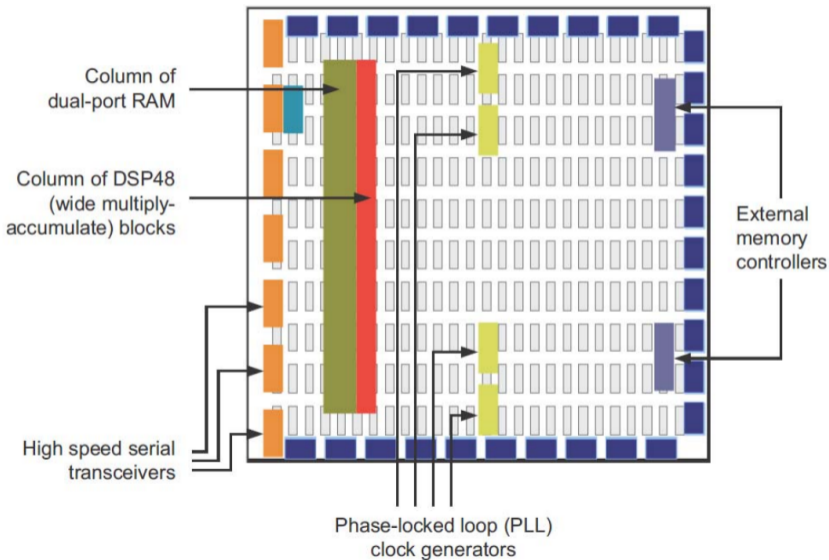


Typical Xilinx LUT have 6 inputs

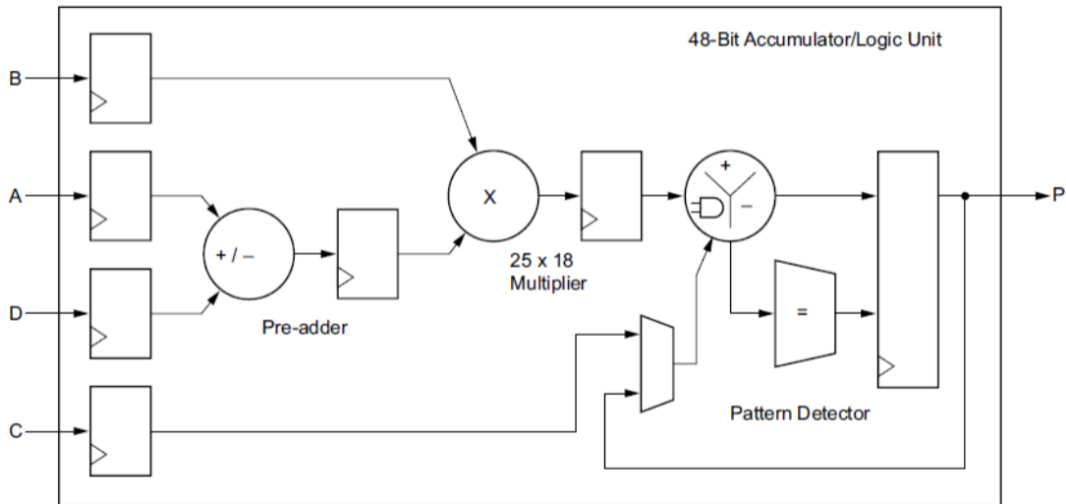




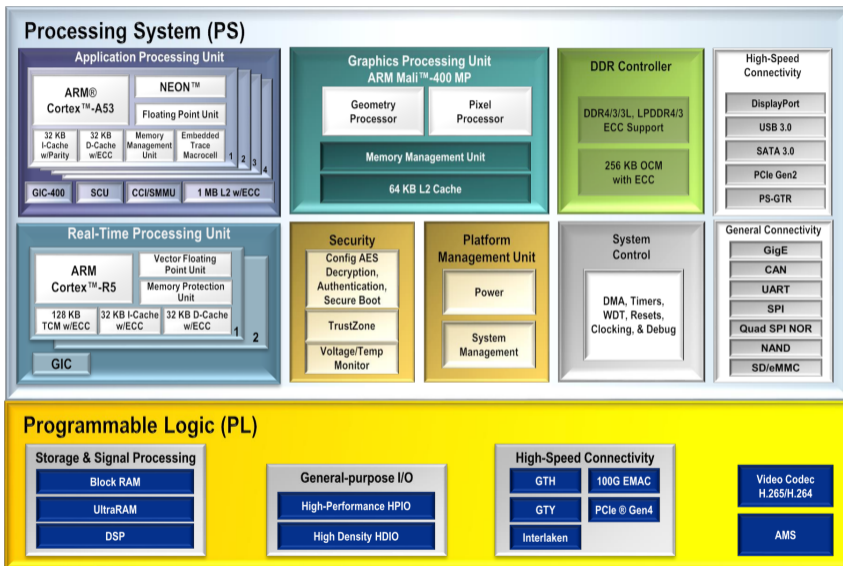
# Global view of programmable logic part



# DSP48 block overview



# Xilinx Zynq UltraScale+ MPSoC Overview




# Motivation

- C++ targets both high-level programming and hard-core optimization
- More and more heterogeneous architectures

## No performance portability...

- ...But C++ should provide the usual fine tuning framework
- ...Even on heterogeneous architectures!

# How to express details?

- Generalized attributes
  - ▶ Allow terse incremental programming
  - ▶ Might not change semantics
  - ▶ Might be ignored by the compiler
- #pragma
  - ▶ Allow terse incremental programming
  - ▶ Simple to do simple things
  - ▶ Might not compose nicely with generic programming
- Use wrapper objects
  - ▶ Normal classes  integrated in type system
  - ▶ Poor (wo)man introspection with type traits
    - Specialization possible according to wrapper kind
  - ▶ Allow RAI
  - ▶ Plain C++: do not require a compiler to prototype
  - ▶ Require deeper program change and generic programming
- Language extensions
- ...

# Example of classes/methods around C++

- Constructors
- Destructors
- Operators
- Functors
- Allocators
- Views
- Spans
- array\_ref
- Executors
- Relocators
- ...

~> ¿Accessors?

# Outline

- 1 Heterogeneous architectures
- 2 Accessors
- 3 Conclusion

# Example of accessors from OpenCL SYCL C++

```

#include <CL/sycl.hpp>
#include <iostream>

using namespace cl::sycl;

constexpr size_t N = 2;
constexpr size_t M = 3;
using Matrix = float[N][M];

// Compute sum of matrices a and b into c
int main() {
    Matrix a = { { 1, 2, 3 }, { 4, 5, 6 } };
    Matrix b = { { 2, 3, 4 }, { 5, 6, 7 } };

    Matrix c;

    { // Create a queue to work on default device
      queue q;
      // Wrap some buffers around our data
      buffer<> A { a };

```

```

      buffer<> B { b };
      buffer<> C { c };
      // Enqueue some computation kernel task
      q.submit([&](handler& cgh) {
        // Define the data used/produced
        auto ka = A.get_access<access::mode::read>(cgh);
        auto kb = B.get_access<access::mode::read>(cgh);
        auto kc = C.get_access<access::mode::write>(cgh);
        // Create & call kernel named "mat_add"
        cgh.parallel_for<class mat_add>(range<2> { N, M },
          [=](id<2> i) { kc[i] = ka[i] + kb[i]; }
        );
      }); // End of our commands for this queue
    } // End scope, so wait for the buffers to be released
    // Copy back the buffer data with RAII behaviour.
    std::cout << "c[0][2]_=" << c[0][2] << std::endl;
    return 0;
}

```

Accessing data from another memory space on accelerators (may hide host-device communications...)



# Accessor

- Wrapper object
- Change the way wrapped object is accessed
- Behaves basically like the wrapped type...
- ...But with added twisted properties

```
template <typename T, typename ... access_kind>  
struct std::accessor<T, access_kind... >;
```

# Speculative execution

```
auto result = heavy_computation(lengthy_io());
```

- But what if often the I/O result is 42...?
- Could be rewritten as

```
auto result = heavy_computation(make_accessor<likely> { lengthy_io(), 42 });
```

- Implementation may launch `lengthy_io()` and start `heavy_computation(42)` if no undoable side-effect
  - ▶ When `lengthy_io()` finishes, compare to 42 and serve or compute result accordingly
- (Should we add in C++ standard the value 42 as a default default answer value? 😊)
- Instruct compiler to instrument code for statistical analysis

```
// Execute f() ahead after some PGO analysis to figure out common values
auto result = f(make_accessor<pgo_likely> { some_io() });
```

# Read/write qualification

```
auto io = get_some_memory_mapped_io();  
// Basic type inferred by constructor  
std::accessor<write_only> use_it { io };  
// Now we are sure this is a write-only usage  
some_random_code(use_it);
```

- `write_only` to access to a write-only data
- `read_only` to prevent writing back data
- `discard_read` to read data if they are written first and do not read the initial value
  - ▶ Typical use case: data locally generated that can be read back locally
- `discard_write` to write data to be locally read but won't be written back
- `noaccess` by symmetry
  - ▶ Typically to detect inappropriate behaviours in a program

# Non unified memory

- C++ assumes uniform memory address space
  - ▶ Not true for HPC Top 500-class machines
  - ▶ Not true for embedded systems
  - ▶ Not true with some accelerators
- $\exists$  PGAS (Partitioned Global Address Space) languages or libraries, such as Coarray C++ or UPC++
  - ▶ Physically distributed memory remotely accessed with explicit language or library support
  - ▶ No transparent virtual shared memory

```
std::array<double, N> local;
accessor<remote, write_only> remote = get_some_runtime_remote(local);
```

```
std::copy(std::begin(local), std::end(local), std::begin(remote));
```

- Typical use case: SHMEM, Portal4, SYCL, Coarray C++, RDMA, iWARP, MPI...
- Real power from remote accessors would come from the combination with the array\_ref P0009R2

# Non temporal access

- To access data that won't be accessed again
  - ▶ Typically will not use a cache
  - ▶ Decrease cache transactions and leave cache for some more useful usage
- Example when generating huge amount of data to an array
  - ▶ Everything else would be evicted from the cache without any chance to read the array back from the cache anyway

```
// An array of 1 TB
double a[2<<37];
auto ac = make_accessor<non_temporal, write_only>(a);
// Initialize a, starting from the end with increasing
// integer starting at 42
std::iota(std::par_vec, ac.rbegin(), ac.rend(), 42);
```

# Aliasing

- Data structure aliasing prevents the compiler from doing aggressive optimizations

```
double a[N], b[N];
// Dummy class declarations to be used as alias set tags
class ta;
class tb;

auto ca = make_accessor<alias_tag<ta>>(build_complex_data_structure(a));
auto cb = make_accessor<alias_tag<tb>>(build_complex_data_structure(b));
auto oa = make_accessor<alias_tag<ta>>(other_complex_data_structure(a));
auto ob = make_accessor<alias_tag<tb>>(other_complex_data_structure(b));
// No aliasing:
correlation(ca, cb);
correlation(oa, ob);
// Aliasing:
correlation(ca, oa);
correlation(cb, ob);
```

- Alternative to N3988 but reuse same alias tag concept
- Compared to N3988 possible to use different algorithms just with metaprogramming

# Sequential access

- In some case the programmer knows that accesses to an array are strictly sequential but the compiler cannot prove it

```

{
  std::accessor<sequential, discard_read, discard_write> a { some_array };
  for (int i; i = 0; i != N; ++i)
    a[i] = i;
  for (int i; i = 0; i != N; ++i)
    b[i] = a[i]*2;
}

```

- Compiler could decide either
  - Remove array access (array scalarization) & fuse the 2 loops
  - Remove array access & generate 2 Kahn's processes
    - 1 producer and 1 consumer
    - Efficient hardware FIFO in between and eluding the memory transfer on `a[i]`
    - Pipelined implementation on FPGA

# Prefetching

- Latency often performance killer
- If we know in advance we will read some array elements, we could prefetch it...
- Prefetching is actually independent from caching
  - ▶ Can be combined with non temporal access

```
int a[N];  
// Instruct the memory prefetcher to look 16 elements ahead  
std::accessor<prefetch<16>> a_p { a };  
auto result = std::accumulate(std::begin(a_p), std::end(a_p), 0);
```



# Burst mode

- Most memory interfaces work better with coarse grain transfers

```
int a[N*20];  
// Instruct the memory prefetcher to use a burst mode of 20 elements  
std::accessor<burst<20>> a_p { a };  
  
// To generate random integers between 0 and N - 1  
std::default_random_engine r;  
std::uniform_int_distribution<int> d { 0, N - 1 };  
for (int i = 0; i != N; ++i) {  
    // Randomly write blocks of 20 elements  
    p = std::begin(a_p) + 20*d(r);  
    std::fill(p, p + 20, 0);  
}
```

# DMA

- Take advantage from DMA to transfer data

```
int far_far_away[N];  
{  
  int near_and_fast[N];  
  auto p = make_accessor<dma> { far_far_away , near_and_fast };  
  std::generate(p, p + 20, 0);  
}
```

- Useful to combine with `write_only`, `read_only`, `discard_read` & `discard_write`

# Finer hardware control

- Bus type

```
char a;
float b;
std::accessor<bus<axi4>> a_a { a };
std::accessor<bus<axi4>> b_a { b };
std::accessor<bus<main> a_m { a };
std::accessor<bus<main> b_m { b };
// Use different buses in parallel to improve bandwidth
auto sum = a_a + b_m;
auto prod = a_m + b_a;
```

- Access width

```
double d;
// Transfer 8-bit at a time
std::accessor<bit_width<8>> d_b { d };
auto a = d_b;
```

- Address modes outside of `std::accessor<address_mode<normal>>` (PC-relative...)

# Generic proxy

- Delegates all read and write operations to some user-provided functors

```
void instrument(int v[]) {  
    std::accessor<proxy> p { v, read_functor, write_functor };  
    // Call f on p instead of v to intercept the read and write  
    f(p);  
}
```

## Useful for

- Virtualizing some non-existent memory or hardware
- Implementing transactional memory in user mode
- Testing with some non-existing software part by interacting with a mock-up hidden behind an accessor
- Override memory operations to do fault injection for fault-tolerance evaluation
- Security testing with fuzzing of inputs

# Modulo addressing

- Circular buffers are common in data processing
- DSP have modulo addressing mode

```
unsigned char buffer[N];  
// b is a kind of infinite array, but with only buffer storage repeated  
std::accessor<modulo> b { buffer };
```

- A specialized version of P0059R1 (ring) could use this

# Translation

- In embedded systems, some level of shared memory
- Might be mapped physically at different addresses

```
unsigned char frame_buffer[N];  
size_t offset = &display - frame_buffer;  
// The screen memory on the display controller  
std::accessor<translate> b { frame_buffer, offset };  
set_graphics(b);  
draw_some_stuff(b);
```

# Address bit setting

- Some architectures encode in the address bus some semantics
  - ▶ Not part of the address itself
  - ▶ Supervisor mode
  - ▶ Non executable mode
  - ▶ ...

```
int a[N];  
a[0] = 2;  
// On the target architecture , the address space is duplicated on the  
// half upper 32KB space for a cacheless access  
std::accessor<bit_set<or<(1 << 15)>> cacheless_access { a };  
// as a[123] but do not use the cache  
cacheless_access[123] = 4;
```

# Transactional memory

```
int shared_data[N];
```

```
void f() {  
    std::accessor<transaction> a { shared_data };  
    foo(a);  
}
```

```
void g() {  
    std::accessor<transaction> b { shared_data };  
    bar(b);  
}
```

- May be executed from 2 different threads
  - ▶ Use transactional memory behaviour if available
  - ▶ Otherwise fallback on lock-based solution



# Type traits

- Type traits to introspect accessors at compile time
  - ▶ `std::is_accessor<property>(acc)`
  - ▶ `std::is_accessor_v<property>(acc)`
  - ▶ `std::get_accessor<property>(acc)`
  - ▶ `std::get_accessor_v<property>(acc)`

```
auto correlation = [] (auto data1, auto data2) {
    if constexpr (std::is_accessor_v<aliasing_with>(data1, data2))
        slow_conservative_correlation(data1, data2);
    else
        crazy_aggressive_correlation(data1, data2);
};
```

# Implicit accessor with attribute?

- Explicit accessor objects may be painfully intrusive in some cases
- Allow something lighter?

```
// An array of 1 TB
double a[2<<37] [[std::accessor<non_temporal, sequential >]];
```

```
// An implicit accessor is wrapped around all uses of a
std::iota(a.begin(), a.end(), 0);
```

But also on any scope, such as class, block, namespace... as for example to add a behaviour of a transactional memory on all a class:

```
template <typename T>
class message_queue [[std::accessor<transaction >]] {
    T read() {...}

    void write(T &&t) {...}
}
```

# Outline

- 1 Heterogeneous architectures
- 2 Accessors
- 3 Conclusion

# Conclusion

- Ubiquitous heterogeneous computing: manycores, SIMD, GPU, FPGA, DSP, PiM...
- Distributed memories and different address spaces
- Go on with C++ success: high-level + efficiency
- Need ways to control new (and old) low level features
- Accessors are a way to express it at type level without language extension
  - ▶ Accessor  $\approx$  span + access properties
  - ▶ Combine with other proposals: `array_ref`, `span`, `atomic_view`...
  - ▶ Seamless templated single source programming
- Give control back to programmer *only when needed*
  - ▶ Only pay for what you need: C++ adage pushed into... hardware and electricity bill!

<http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2016/p0367r0.pdf>

<p><b>1</b> Heterogeneous architectures</p> <ul style="list-style-type: none"> <li>Outline</li> <li>Power wall &amp; speed of light: the final frontier...</li> <li>(Rather old) 45nm technology characteristics</li> <li>Space-time traveling</li> <li>Power wall &amp; speed of light: implications</li> <li>From AMD Fiji XT GPU (2015)...</li> <li>Basic architecture = Lookup Table + Flip-Flop storage + Interconnect</li> <li>Global view of programmable logic part</li> <li>DSP48 block overview</li> <li>Xilinx Zynq UltraScale+ MPSoC Overview</li> <li>Motivation</li> <li>How to express details?</li> <li>Example of classes/methods around C++</li> </ul> <p><b>2</b> Accessors</p> <ul style="list-style-type: none"> <li>Outline</li> <li>Example of accessors from OpenCL SYCL C++</li> <li>Accessor</li> <li>Speculative execution</li> </ul>	<p>2</p> <p>3</p> <p>4</p> <p>5</p> <p>6</p> <p>7</p> <p>8</p> <p>9</p> <p>10</p> <p>11</p> <p>12</p> <p>13</p> <p>14</p> <p>15</p> <p>16</p> <p>17</p> <p>18</p>	<p>Read/write qualification</p> <p>Non unified memory</p> <p>Non temporal access</p> <p>Aliasing</p> <p>Sequential access</p> <p>Prefetching</p> <p>Burst mode</p> <p>DMA</p> <p>Finer hardware control</p> <p>Generic proxy</p> <p>Modulo addressing</p> <p>Translation</p> <p>Address bit setting</p> <p>Transactional memory</p> <p>Type traits</p> <p>Implicit accessor with attribute?</p> <p><b>3</b> Conclusion</p> <p>Outline</p> <p>Conclusion</p> <p><b>You are here !</b></p>	<p>19</p> <p>20</p> <p>21</p> <p>22</p> <p>23</p> <p>24</p> <p>25</p> <p>26</p> <p>27</p> <p>28</p> <p>29</p> <p>30</p> <p>31</p> <p>32</p> <p>33</p> <p>34</p> <p>35</p> <p>36</p> <p>37</p>
--	---	--	---