Study of the feasibility of accelerating MC generation with FPGAs

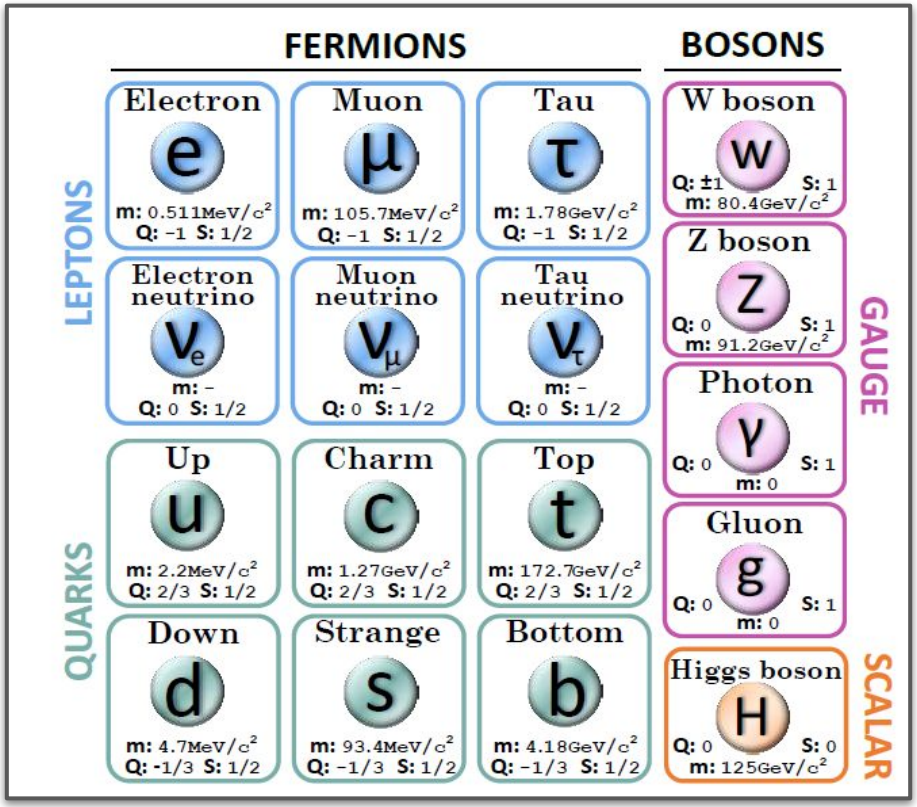
Jornadas ICTEA 2025

Santiago Folgueras[1], Carlos Vico Villalba[1], Pelayo Leguina López[1]
Luca Fiorini[2], Héctor Gutiérrez Arance[2], Alberto Valero Biot[2], Francisco Hervás Álvarez[2],
Arantza De Oyanguren Campos[2], Jiahui Zhuo[2], Volodymyr Svintozelskyi[2], Valerii Kholoimov[2],

[1] Universidad de Oviedo, Asturias, Spain
[2] Instituto de Física Corpuscular (CSIC-UV), Valencia, Spain

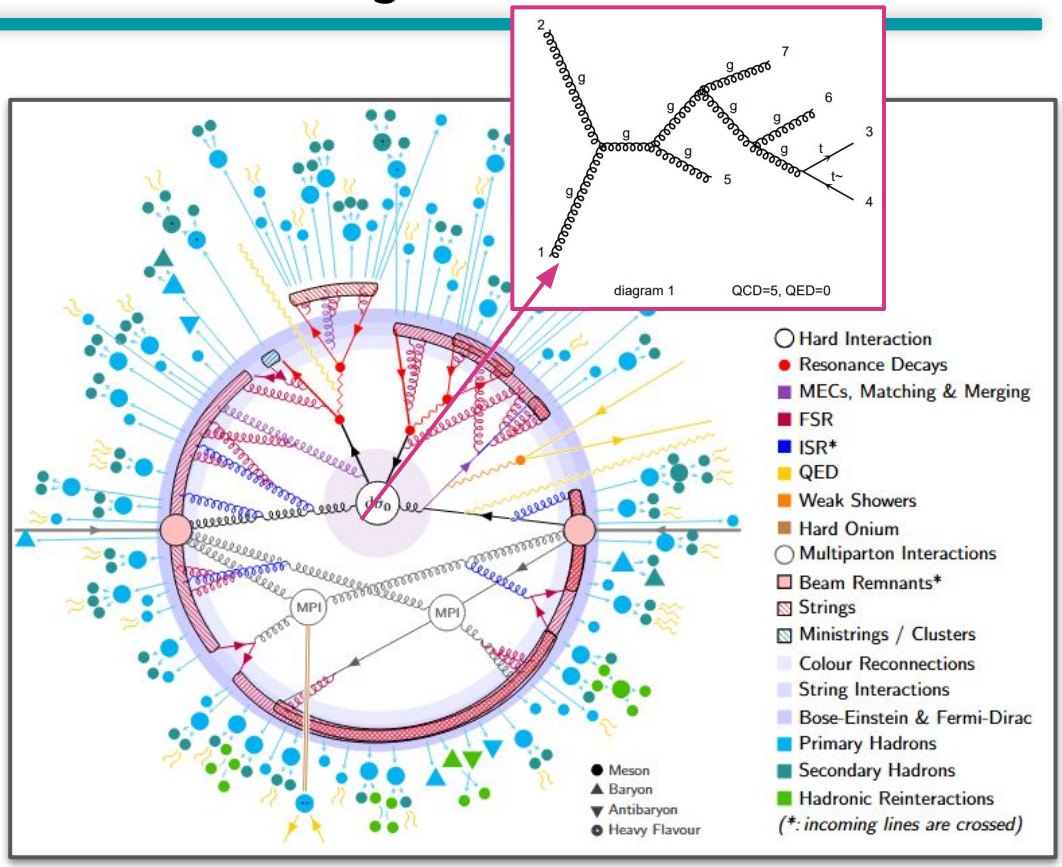
The Standard Model of Particle Physics

- The **SM** is a **renormalizable Quantum Field Theory**.
  - It **predicts** the nature of **high energy interactions between** the **particles** of the model.
- The **goal** of the experiments at the **LHC** is to
  - **Measure** the **properties** of **these particles**.
  - **Explore** further **extensions** in search of **new physics.**



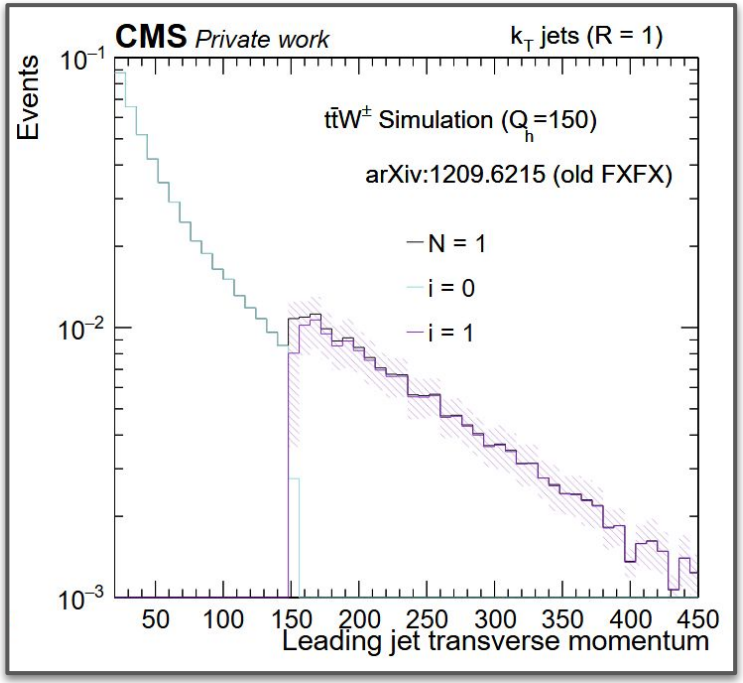
Montecarlo modeling

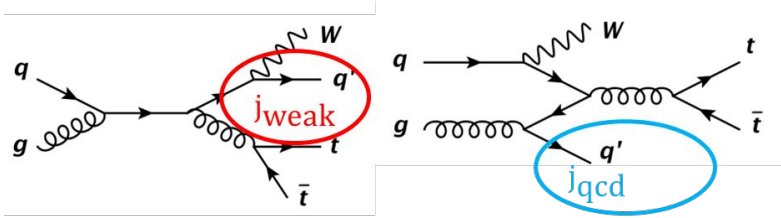
- The **SM predictive power** is **tested against** the recorded **data.**
  - **Montecarlo (MC) generation** approaches **allow to resembles** the nature of the **data** (counting experiments) **by generating events.**
- Based on a **factorized approach** that **divides** the whole **calculation into steps.**
  - Hard interactions: Leading Order (LO) + Next-to-LO (NLO) α_s expansion terms in a perturbative series at scale μ.
  - Soft interactions: approximation of collinear/soft QCD/weak radiation effects.



Montecarlo modeling

- **MC modeling** (how well the simulations describe the data) is **essential** for the **LHC physics programme.**
  - **For instance:** ttW (measured in Oviedo) production has shown slight tensions with the theory.
  - (Also in Oviedo) the modeling is thoroughly reviewed.

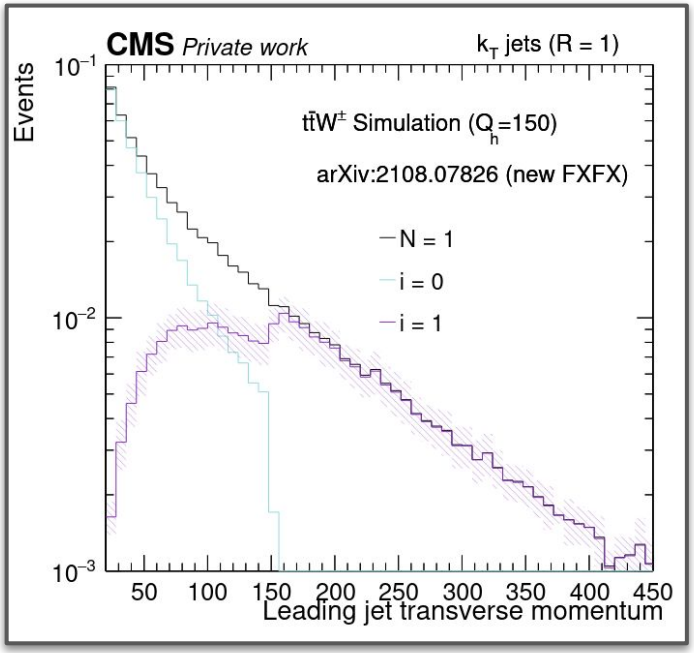


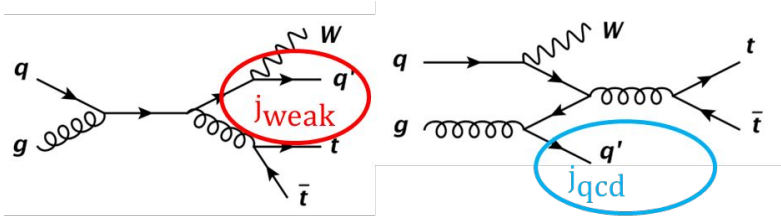


- **Left:** momentum in the transverse plane for the radiation emitted when producing a pair of quarks and a W boson.
- **Problem in 2021:** discontinuity at 150 GeV.
  - → Caused a **4 standard deviations disagreement** between experiment and theory!

Montecarlo modeling

- **MC modeling** (how well the simulations describe the data) is **essential** for the **LHC physics programme**.
  - **For instance:** ttW (measured in Oviedo) production has shown slight tensions with the theory.
  - (Also in Oviedo) the modeling is thoroughly reviewed.





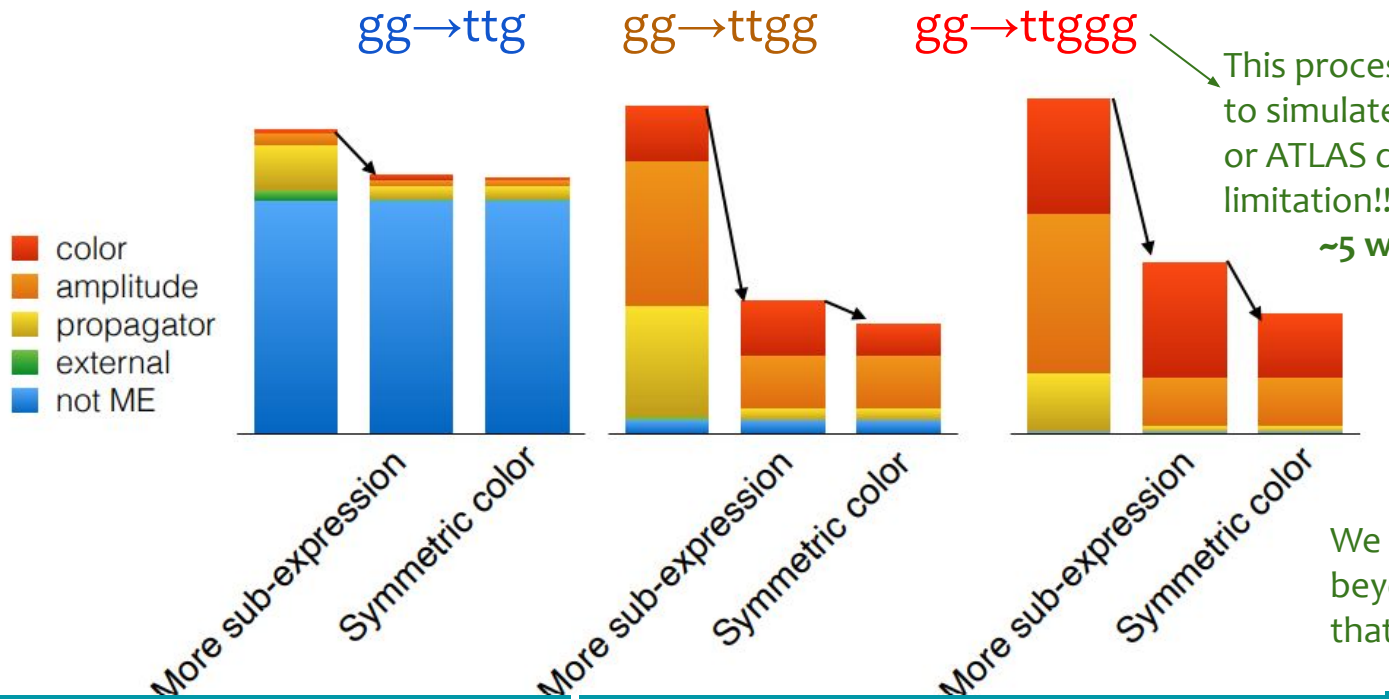
- **Left:** momentum in the transverse plane for the radiation emitted when producing a pair of quarks and a W boson.
- **Solution in 2022 (from MC builders):**
  - Proper modeling of weak radiation off of W boson.
  - A slight tension remains at ~2 standard deviations.
  - Much more controlled.

Current issues

- **MC modeling** is a challenge from multiple perspectives:
  - In the case of ttW → > 2 years of work just to get this MC simulation working in CMS!!
- **Timetable for this work:**
  - Few months of **compiling all the diagrams** that make up the **hard interaction**.
    - Each time we do this: ~1 day of running (`Intel(R) Core(TM) i7 CPU 970 @ 3.20GHz`)
  - Validation has to be made in a large enough sample of events (statistically meaningful predictions):
    - We can generate about 1M events in ~1 day
    - A reasonably large sample is made out of 10M events
    - This, in any case, runs relatively fast and efficient.
- **The bottleneck is in the compilation time**

Current issues

- **This issue** is actually more relevant for much more common processes → example: top-antitop pair production (measured in Oviedo).
- The problem grows as we add more radiation diagrams



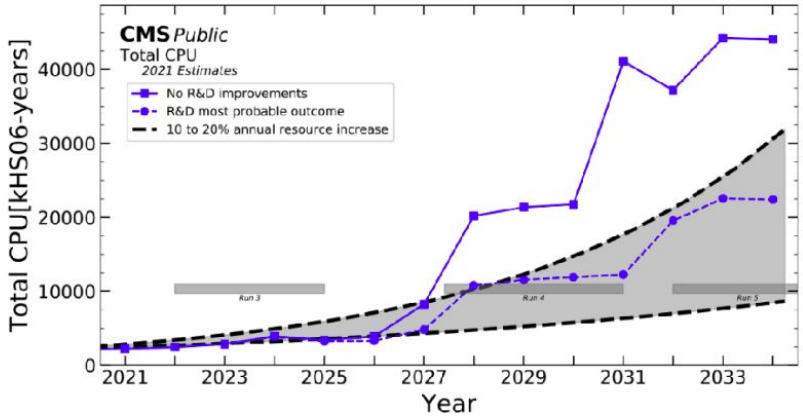
This process is **effectively impossible** to simulate in experiments like CMS or ATLAS due to a computing limitation!!!

~5 weeks of compilation

We normally approximate beyond the third radiation and that "does the trick"

Current issues

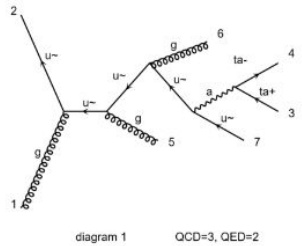
- **"Does the trick"** → assume the radiation is soft/collinear, take the modeling at degraded accuracy from the parton shower.
  - Works very well for smaller datasets → not a problem for the Run 2 + Run 3 results.
  - Let's not forget we are aiming at measuring properties of the Higgs in phase spaces with up to 4 jets (HH→4b channel) in the final state → this is extremely sensitive to these effects.
  - One needs to solve the bottleneck by better improving / pipelining the computation through improved running architectures such as **GPU**



Where the community is going

- **The developers** of one of the most commonly used MC software package (`Madgraph5_aMC@NLO`) are already working towards a solution to this problem.
  - As of now `Madgraph5_aMC@NLO` works in `Fortran`. Runs on CPU.
  - They ported the code to consider `CUDA/C++` which is significantly faster.

$g\bar{u} \to \tau^+\tau^- gg\bar{u}$
(subprocess of DY+3j)
100 Feynman diagrams
6x6 color matrix



$g\bar{u} \to \tau^+\tau^- gg\bar{u}$ (81920 weighted events)	MEs precision	madevent $t_{TOT} = t_{Mad} + t_{MEs}$ [sec]	madevent N_{events}/t_{TOT} [events/sec]	madevent N_{events}/t_{MEs} [MEs/sec]
Fortran(scalar)	double	52.2 = 17.0 + 35.2	1.57E3 (=1.0)	2.32E3 (=1.0)
C++/none(scalar)	double	50.9 = 17.0 + 33.9	1.61E3 (x1.0)	2.42E3 (x1.0)
C++/sse4(128-bit)	double	35.5 = 17.0 + 18.5	2.31E3 (x1.5)	4.44E3 (x1.9)
C++/avx2(256-bit)	double	24.5 = 16.9 + 7.6	3.34E3 (x2.1)	1.08E4 (x4.7)
C++/512y(256-bit)	double	23.9 = 16.9 + 7.0	3.43E3 (x2.2)	1.17E4 (x5.0)
C++/512z(512-bit)	double	26.7 = 17.0 + 9.6	3.09E3 (x2.0)	8.57E3 (x3.7)
CUDA/GPU	double	17.6 = 17.4 + 0.3	4.65E3 (x3.0)	3.26E5 (x140)
C++/none(scalar)	mixed	50.9 = 16.9 + 33.9	1.61E3 (x1.0)	2.41E3 (x1.0)
C++/sse4(128-bit)	mixed	33.9 = 16.9 + 17.0	2.41E3 (x1.5)	4.82E3 (x2.1)
C++/avx2(256-bit)	mixed	24.8 = 17.2 + 7.6	3.31E3 (x2.1)	1.08E4 (x4.7)
C++/512y(256-bit)	mixed	24.1 = 17.1 + 7.0	3.40E3 (x2.2)	1.18E4 (x5.0)
C++/512z(512-bit)	mixed	26.5 = 17.0 + 9.6	3.09E3 (x2.0)	8.57E3 (x3.7)
CUDA/GPU	mixed	17.7 = 17.4 + 0.3	4.64E3 (x3.0)	3.23E5 (x138)
C++/none(scalar)	float	50.1 = 16.9 + 33.1	1.64E3 (x1.0)	2.47E3 (x1.1)
C++/sse4(128-bit)	float	26.3 = 16.9 + 9.4	3.11E3 (x2.0)	8.70E3 (x3.7)
C++/avx2(256-bit)	float	20.8 = 16.9 + 3.9	3.94E3 (x2.5)	2.12E4 (x9.1)
C++/512y(256-bit)	float	20.6 = 16.9 + 3.6	3.99E3 (x2.5)	2.27E4 (x9.7)
C++/512z(512-bit)	float	21.7 = 16.9 + 4.8	3.78E3 (x2.4)	1.71E4 (x7.3)
CUDA/GPU	float	17.6 = 17.4 + 0.2	4.66E3 (x3.0)	4.46E5 (x191)

AV – CHEP2024

[By A. Valassi (CHEP 2024)]

For one typical subprocess of DY+3jets:
Fortran MEs ~ 67% of the total time
=> **Max overall speedup is x3 (Amdahl)**

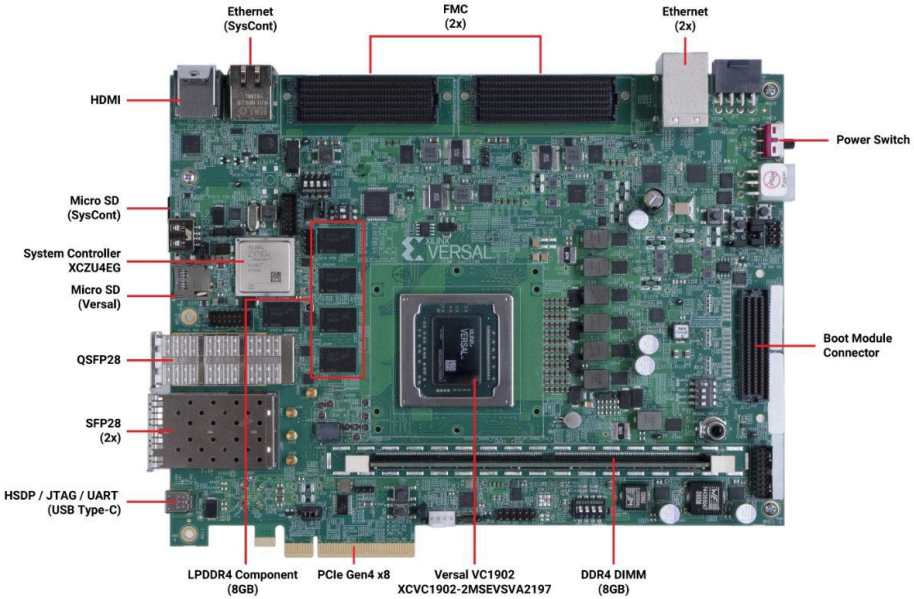
Achieved speedups (mixed FP precision):
- **x3.0 on GPU** (NVidia V100)
- **x2.2 on SIMD** ("512y" on Intel Silver)

In Oviedo we went for an alternative way :)

The project

- **One liner:** to perform viability studies on whether **MC generation** workflows can be accelerated using **Field Programmable Gate Arrays (FPGAs)**.

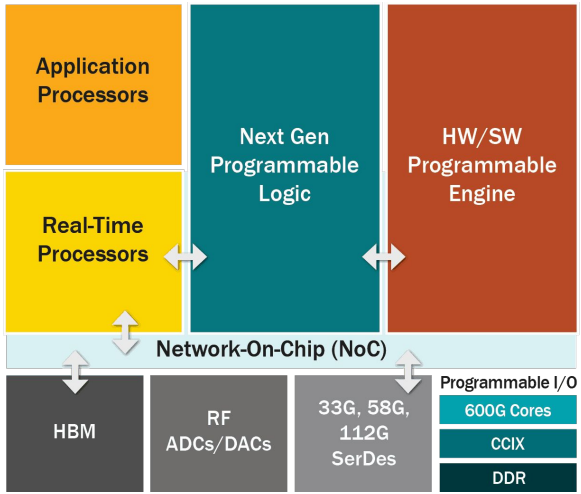
- **The project is a proof of concept**
  - Can we use FPGAs for MC generation?
  - At this stage, we have identified a task to be optimized in FPGA, and started making comparisons against CPU performance.
  - The ultimate goal is to test against GPU performance to see if FPGA can actually be competitive on this task.

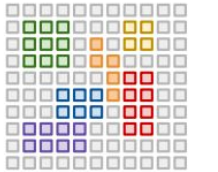


The project

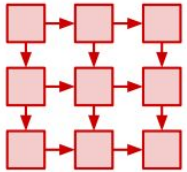
- **Technical description:** A ***Field Programmable Gate Array*** (FPGA) is an integrated circuit with a programmable hardware fabric that allows it to be reconfigured to behave like another circuit.
- **Non-technical description:** an electronic chip built inside a board that can be reprogrammed at any time (unlike GPUs/CPUs or any kind of ASIC).

- The advantages of an FPGA are:
  - **Fine-grained customization**
  - **Spatial compute**
  - **Hardware flexibility**
  - **Diverse IOs**
- The disadvantages are:
  - Little access to non-expert users.
  - "Limited" resources for complex operations (e.g divisions, exponentiations).

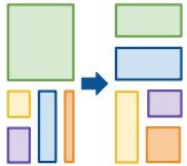




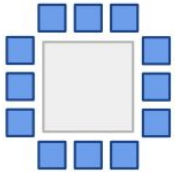
① Fine-grained Customization



② Spatial Compute



③ Hardware Flexibility



④ Diverse IOs

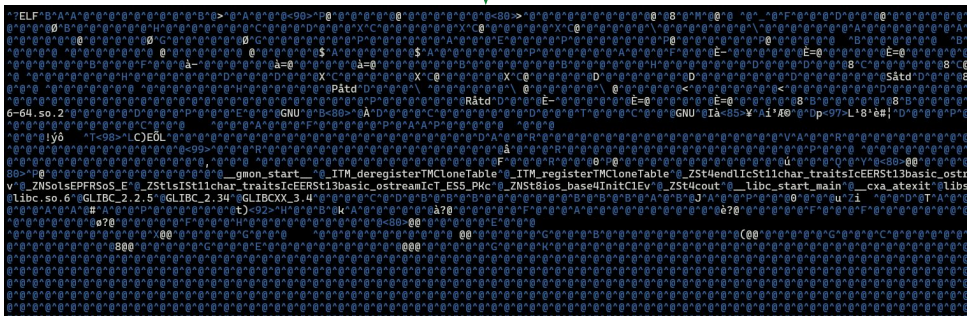
The project

- **In CPU programming:**
  - Write code in a given language (C, C++, fortran).
  - Compile the code.
  - Code is translated into instructions that can be understood by the CPU.
  - We execute the compiled code.

```cpp
#include <iostream>

int main(int argc, char* argv[]) {
    std::cout << "Hello world" << std::endl;
    return 0;
}
```

```
g++ -o code code.cpp
./code
Hello world
```

A bunch of gibberish

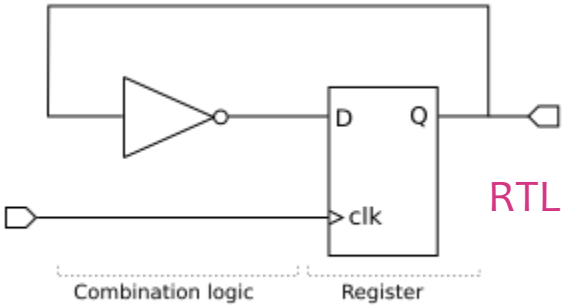


The project

- **In CPU programming:**
  - Write code in a given language (C, C++, fortran).
  - Compile the code.
  - Code is translated into instructions that can be understood by the CPU.
  - We execute the compiled code.
- **In FPGA programming:**
  - We compile code in a given language (VHDL, HLS).
  - The code is translated into Register Transfer Level language (RTL).
- **The RTL tells the FPGA what circuit do we have to build to solve the task.**

```
D ≤ not Q;

process(clk)
begin
    if rising_edge(clk) then
        Q ≤  D;
    end if;
end process;
```

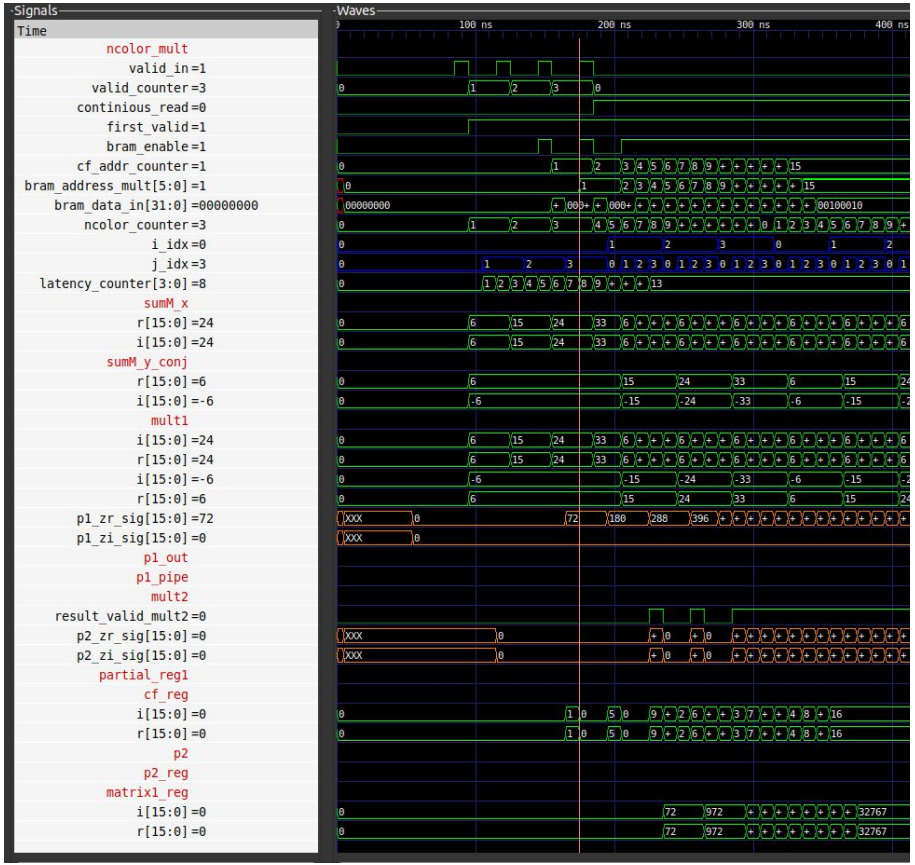
VHDL



RTL

The project

- **In CPU programming:**
  - Write code in a given language (C, C++, fortran).
  - Compile the code.
  - Code is translated into instructions that can be understood by the CPU.
  - We execute the compiled code.
- **In FPGA programming:**
  - We compile code in a given language (VHDL, HLS).
  - The code is translated into Register Transfer Level language (RTL).
- **The RTL tells the FPGA what circuit do we have to build to solve the task.**
  - Which resources from the FPGA are needed to solve the task at hand.



The project

- We focus all of our examples in one of the main processes used in experimental High Energy Physics: ttbar multijets (considering up to 3 jets).

These 10 lines of code actually consume ~60% of the total computation time!!!

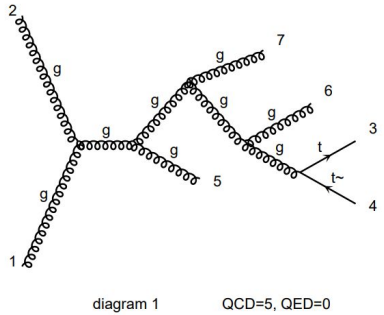
```
c       Save time at the beginning of color calculation
        call cpu_time(t_before_color)
        TS(K) = 0.D0
        DO M = 1, NAMPSO
          DO I = 1, NCOLOR
            ZTEMP = (0.D0,0.D0)
            DO J = 1, NCOLOR
              ZTEMP = ZTEMP + CF(J,I)*JAMP(J,M)
            ENDDO
            DO N = 1, NAMPSO

              TS(K) = TS(K) + ZTEMP*DCONJG(JAMP(I,N))

            ENDDO
          ENDDO
        ENDDO
c       Also after color
        call cpu_time(t_after_color)
        t_color = t_after_color - t_before_color
c       Sum the difference to the accumulator
        CUMULATED_TIMING_INCOLOR = CUMULATED_TIMING_INCOLOR + t_color
        print *, 'Time spent in this call:', t_color
```

t_{it}

C_t

`P1_gg_ttxggg`

- 119 integration channels after `gensym`.
- Focused on one (`G1`).
- We measured the time that is spent in integrating this channel.



- t_{it} is the time per iteration of the color matrix operation.
- C_t is the cumulated time spent in doing this operation
- $C_t = t_{it} \times N_{calls}$, where N_{calls} is the number of calls to functions that integrate the diagram)

The project

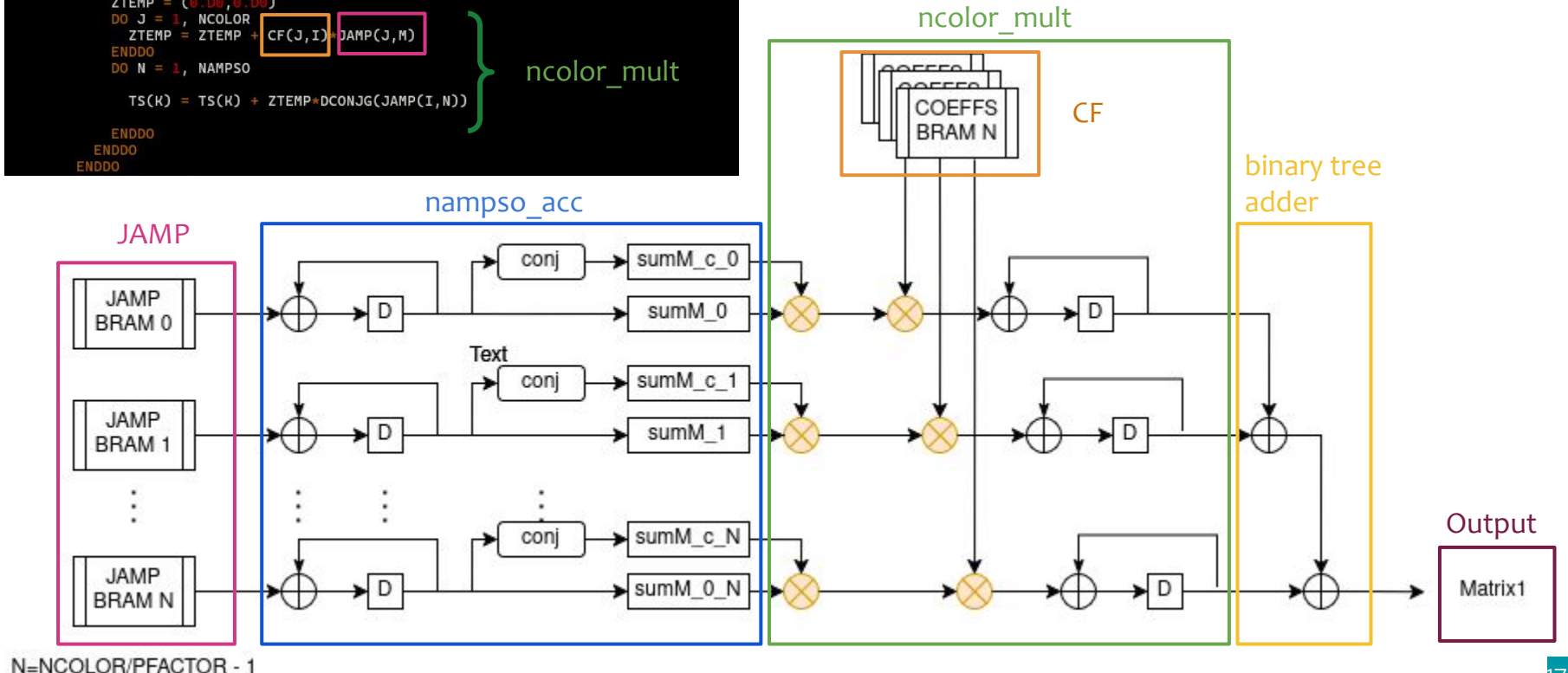
- **Translation to FPGA language**

```
TS(K) = 0.D0
DO M = 1, NAMPSO
  DO I = 1, NCOLOR
    ZTEMP = (0.D0,0.D0)
    DO J = 1, NCOLOR
      ZTEMP = ZTEMP + CF(J,I)*JAMP(J,M)
    ENDDO
    DO N = 1, NAMPSO

      TS(K) = TS(K) + ZTEMP*DCONJG(JAMP(I,N))

    ENDDO
  ENDDO
ENDDO
```

nampso_acc

ncolor_mult

Using an optimization for these kind of computations from [Sign Process Syst 95, 543–550 (2023)]



The project

- We have computed a first preliminary comparison of the performance between CPU/ FPGA DSPs / FPGA AI Cores.

Time spent in one call to the color matrix calculation for different architectures, expressed in microseconds (μs). The numbers for the `fortran` column have been obtained using the default output from `madgraph`. For the **CPU** we use an **`Intel(R) Core(TM) i7 CPU 970 @ 3.20GHz`**. For the **FPGA (DSP)** we use a **VCU13P** FPGA running at **645 MHz**. For the **FPGA (AI CORES)** we are using a **VERSAL** board.

N_{color}	t_{it} CPU (fortran)	t_{it} FPGA (DSP)	t_{it} FPGA (AI Cores)
6	1.00	0.043	2.7
27	2.00	0.14	Work in progress
120	29.00	0.45	Work in progress

Conclusions

- In this talk we have presented a **first temptative implementation** of the **color calculation in a FPGA** device.
  - How does an FPGA work.
  - *A priori* limitations.
- We have started to compare the performance between CPU and two different implementations of the color matrix calculation in FPGA.
- **Our metric so far:** How much time does it get an output per call.
  - Good improvement in timings between CPU and FPGA (DSP).
  - Further optimization needs to be done in the FPGA (`AI CORE`) implementation to get competitive result.
- **To be done in following iterations:**
  - Test another implementation in **High Level Synthesis** language (a C++ interface to VHDL).
  - **Comparisons with GPU performance!**
  - Test compatibility of results between CPU/FPGA.
  - Test against `cudacpp` CPU implementation (possibly faster?)
  - Test the overall time taken to perform the whole integration.

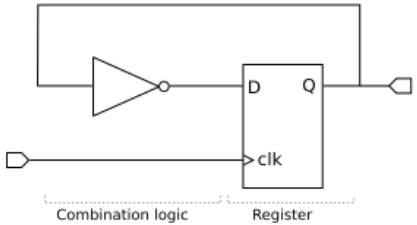
carlos.vico.villalba@cern.ch

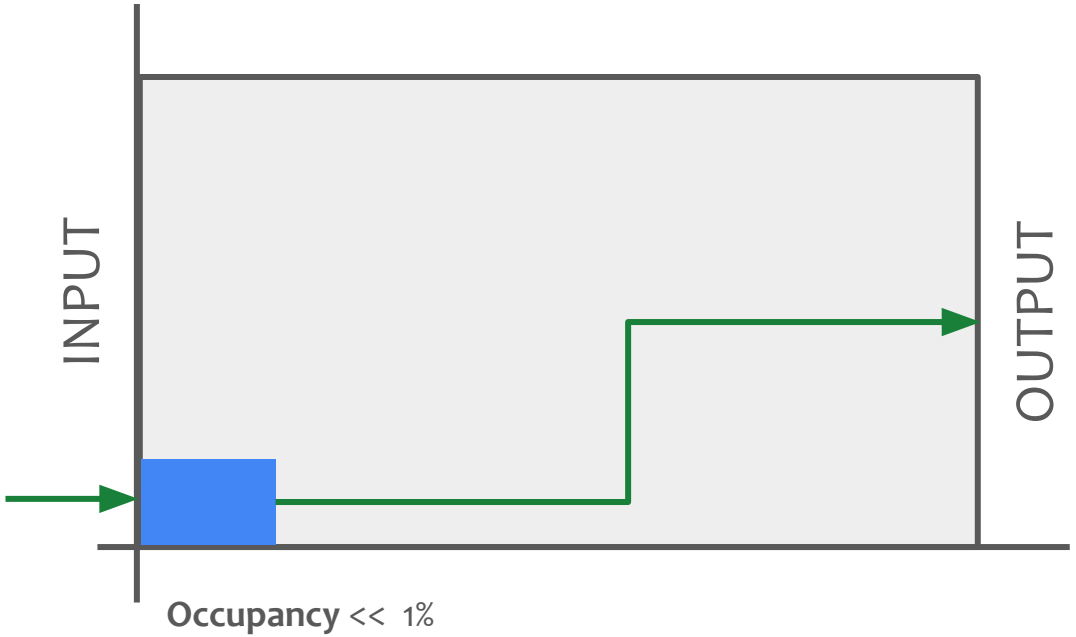
backup

Considerations when developing a FPGA application

- **There are two main considerations to be taken into account.**
  - **Area constraints:** do we have enough resources to perform the task?

```
D ≤ not Q;

process(clk)
begin
    if rising_edge(clk) then
        Q ≤  D;
    end if;
end process;
```





Considerations when developing a FPGA application

- **There are two main considerations to be taken into account.**
  - **Area constraints:** do we have enough resources (LUTs, FFs, BRAMS, DSPs, etc…) to perform the task?
    - The more operations we have to do (in parallel) → the more resources we consume.

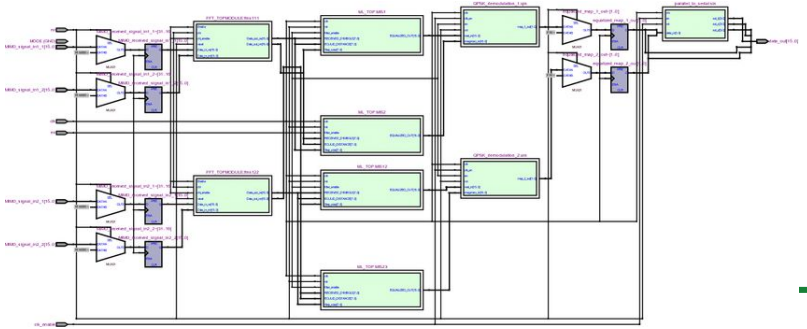
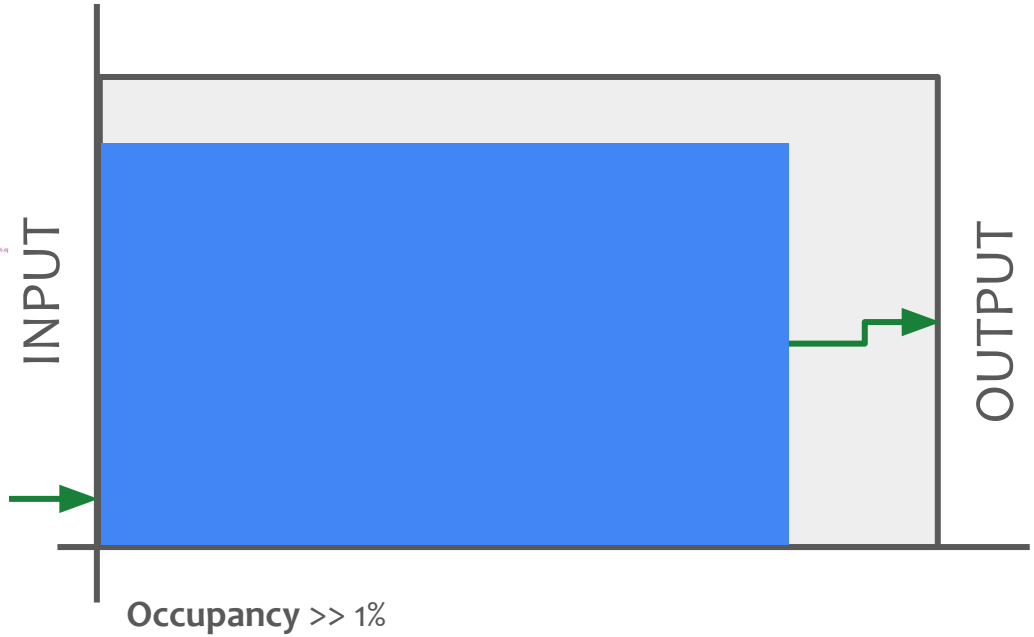


Image used for illustration purposes of a more complex operation.



Considerations when developing a FPGA application

- **There are two main considerations to be taken into account.**
  - **Area constraints:** do we have enough resources (LUTs, FFs, BRAMS, DSPs, etc…) to perform the task?
    - The more operations we have to do (in parallel) → the more resources we consume.

- **Things that significantly affect AREA:**
  - Divisions, exponentiations, …
  - Floating point operations.

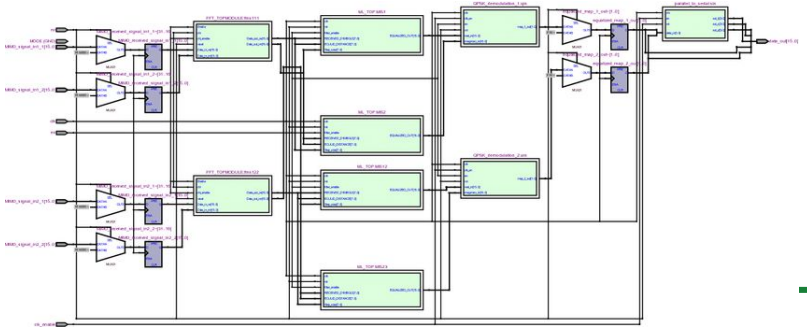
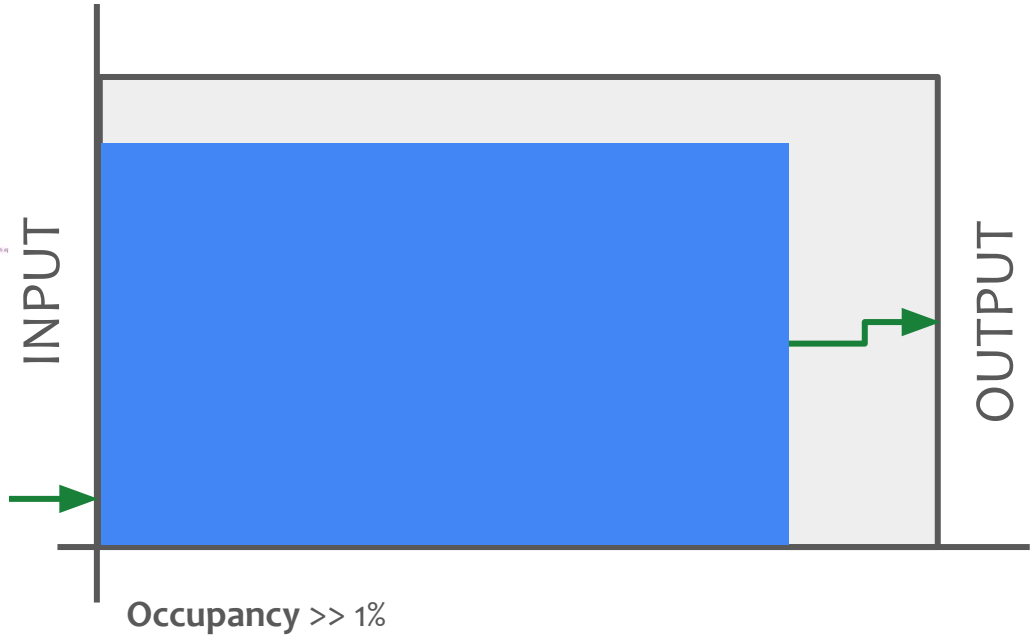


Image used for illustration purposes of a more complex operation.



Considerations when developing a FPGA application

- **There are two main considerations to be taken into account.**
  - **Area constraints:** do we have enough resources (LUTs, FFs, BRAMS, DSPs, etc…) to perform the task?
    - The more operations we have to do (in parallel) → the more resources we consume.

- **Things that significantly affect AREA:**
  - Divisions, exponentiations, …
  - Floating point operations.

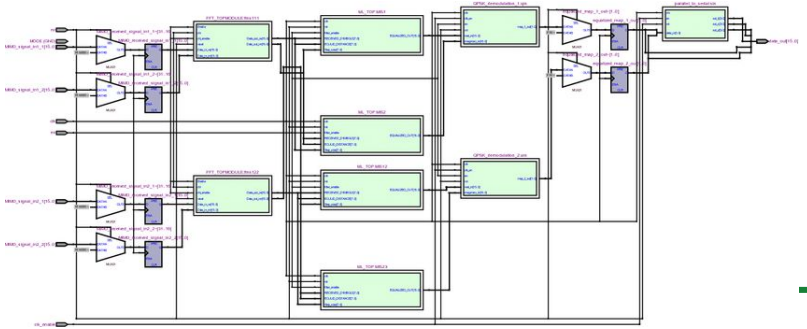
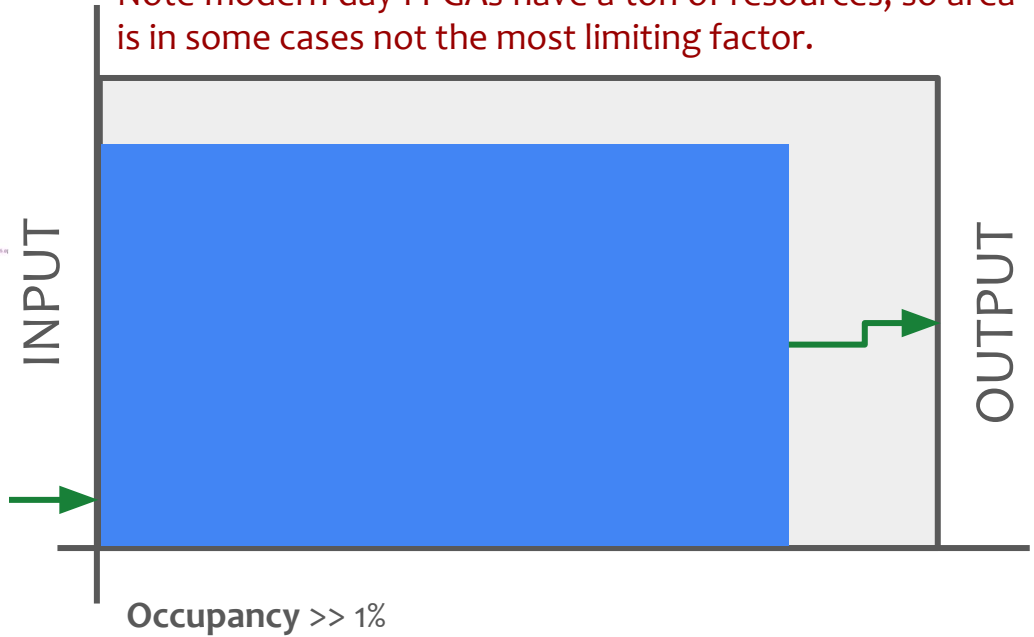


Image used for illustration purposes of a more complex operation.

Note modern day FPGAs have a ton of resources, so area is in some cases not the most limiting factor.

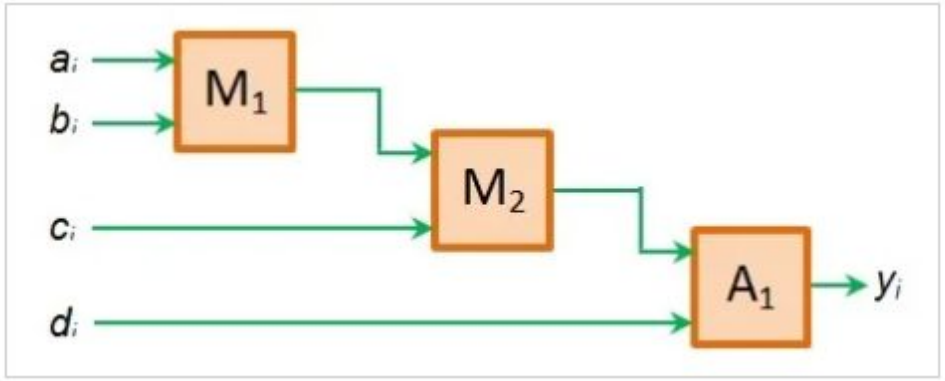


Considerations when developing a FPGA application

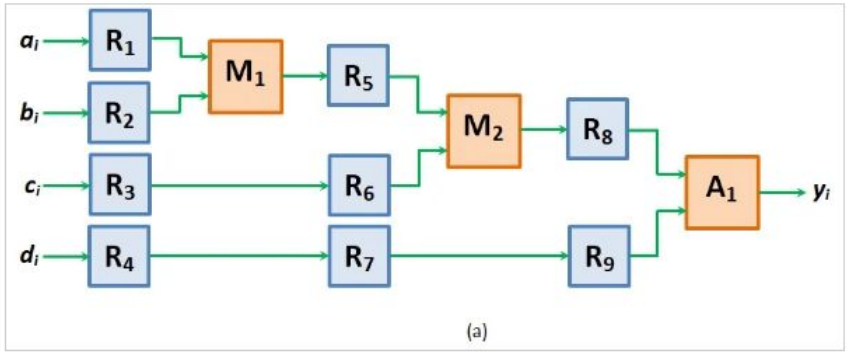
- **There are two main considerations to be taken into account.**
  - **Area constraints:** do we have enough resources (LUTs, FFs, BRAMS, DSPs, etc…) to perform the task?
    - The more operations we have to do (in parallel) → the more resources we consume.
  - **Timing constraints:** algorithm meets the required clock frequency while maintaining correct functionality?
    - **Pipelining:** Pipelining is a technique that **improves data processing speed** by **breaking operations into smaller sequential stages**, allowing partial results to be processed concurrently.
    - After a certain number of cycles, the pipeline reaches a steady state where one output is produced per clock cycle

Considerations when developing a FPGA application

- **There are two main considerations to be taken into account.**
  - **Area constraints:** do we have enough resources (LUTs, FFs, BRAMS, DSPs, etc…) to perform the task?
    - The more operations we have to do (in parallel) → the more resources we consume.
  - **Timing constraints:** algorithm meets the required clock frequency while maintaining correct functionality?
    - **Pipelining:** Pipelining is a technique that **improves data processing speed** by **breaking operations into smaller sequential stages**, allowing partial results to be processed concurrently.
    - After a certain number of cycles, the pipeline reaches a steady state where one output is produced per clock cycle



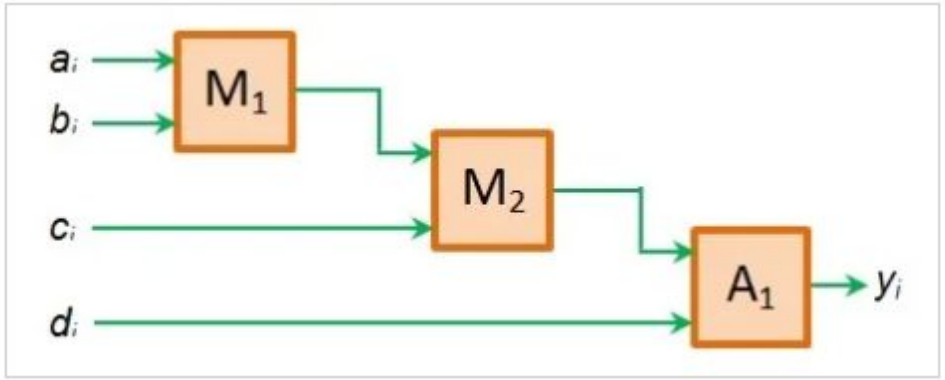
*Example of a **non-pipelined** designed. This design requires 3 clocks to finish, but only 1 is used, leading to time slack.*



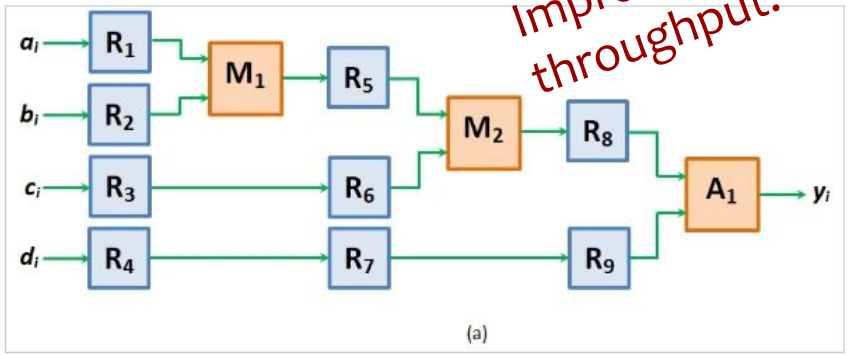
*Example of a **pipelined** designed. This design uses more registers to keep processing inputs while the final result of the first set of inputs is not finished yet. **Consequence: more area.***

Considerations when developing a FPGA application

- **There are two main considerations to be taken into account.**
  - **Area constraints:** do we have enough resources (LUTs, FFs, BRAMS, DSPs, etc…) to perform the task?
    - The more operations we have to do (in parallel) → the more resources we consume.
  - **Timing constraints:** algorithm meets the required clock frequency while maintaining correct functionality?
    - **Pipelining:** Pipelining is a technique that **improves data processing speed** by **breaking operations into smaller sequential stages**, allowing partial results to be processed concurrently.
    - After a certain number of cycles, the pipeline reaches a steady state where one output is produced per clock cycle



*Example of a **non-pipelined** designed. This design requires 3 clocks to finish, but only 1 is used, leading to time slack.*

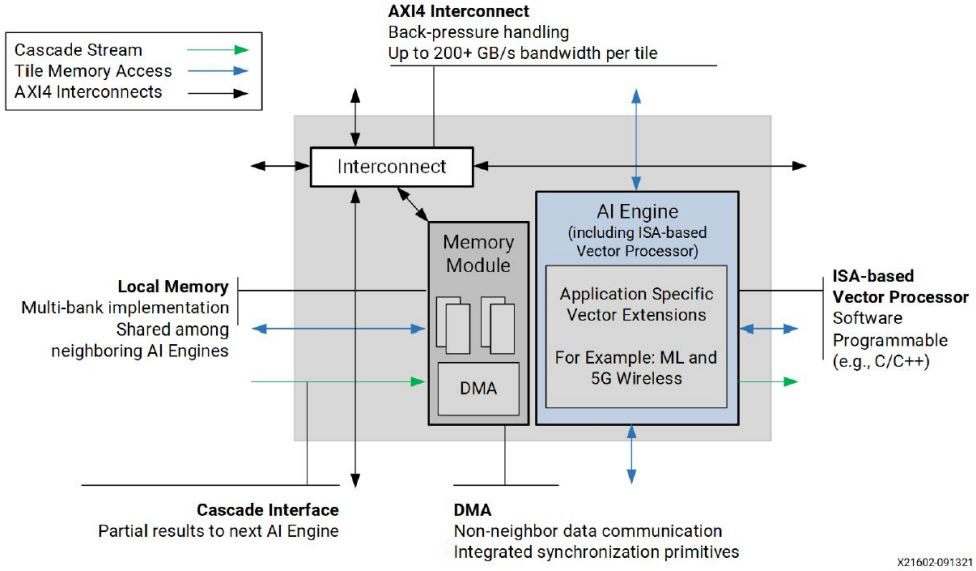


*Example of a **pipelined** designed. This design uses more registers to keep processing inputs while the final result of the first set of inputs is not finished yet. **Consequence: more area.***

Implementation of the color decomposition in FPGA AI Cores

- The previous implementation is done using **Digital Signal Processors** (DSP)**, Blocks of ram** (BRAM)**, Lookup Tables** (LUTs), and other kids of **hardware components.**
  - **Problem:** DSP do not handle well floating point precission.
- **Modern day FPGA boards** (see Versal FPGA) include what is called Adaptive Intelligence (AI) engines.
  - Specialized processing blocks optimized for high-throughput, low-latency workloads

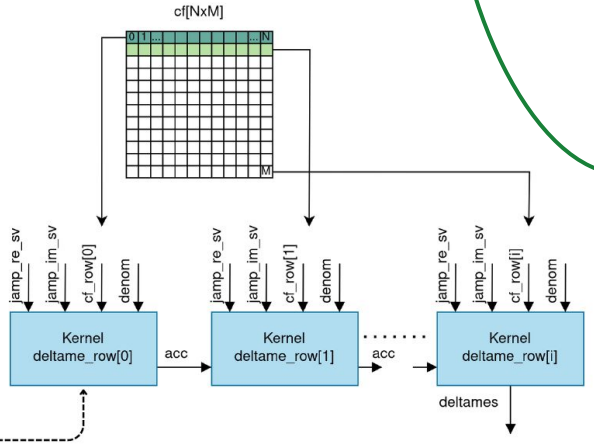
- AI engines are:
  - Arrays of cores for parallel processing.
  - Each core can process vectors in parallel at higher speeds (1.25 GHz).
- **Each core can handle:**
  - 32 KB data memory
  - 16 KB program memory
  - Efficient data flow.



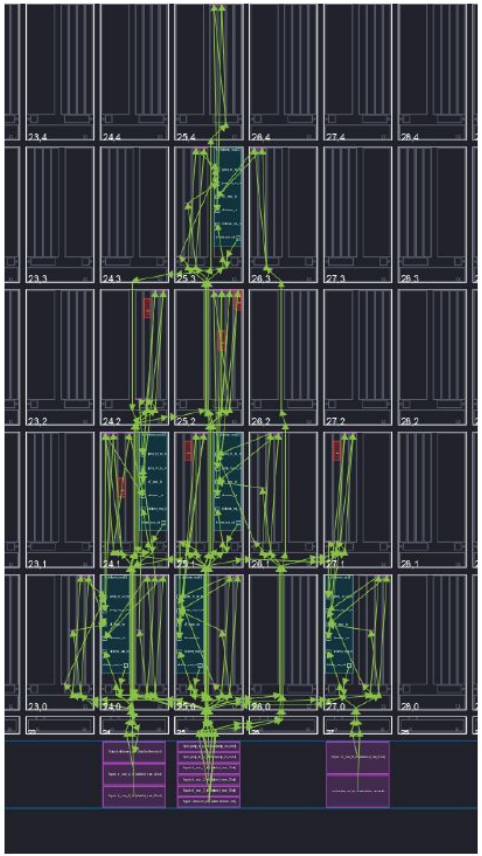
Implementation of the color decomposition in FPGA AI Cores

- Our versal board has ~400 AI engines.
- Even for gg→tt+3j (N_{color} = 120) we can use one kernel per matrix row.
- Each kernel output is passed to the next kernel for accumulating the `Matrix1` result
- Tested so far with gg→tt+1j (N_{color} = 6)

Occupation diagram



$$acc = acc + \frac{[\sum jamp\_re\_sv * cf\_row] * jamp\_re\_sv(i) + [\sum jamp\_im\_sv * cf\_row] * jamp\_im\_sv(i)}{denom(i)}$$



Comparison results

- We have computed a first preliminary comparison of the performance between CPU/ FPGA DSPs / FPGA AI Cores.

Time spent in one call to the color matrix calculation for different architectures, expressed in microseconds (μs). The numbers for the `fortran` column have been obtained using the default output from `madgraph`. For the **CPU** we use an **`Intel(R) Core(TM) i7 CPU 970 @ 3.20GHz`**. For the **FPGA (DSP)** we use a **VCU13P** FPGA running at **645 MHz**. For the **FPGA (AI CORES)** we are using a **VERSAL** board.

N_{color}	t_{it} CPU (fortran)	t_{it} FPGA (DSP)	t_{it} FPGA (AI Cores)
6	1.00	0.043	2.7
27	2.00	0.14	Work in progress
120	29.00	0.45	Work in progress

Implementation of the color decomposition in FPGA (VHDL)

- **Second step:** validation in simulation

Schematic view of the signals throughout one call to the `matrix color` code.

One has to take into account that in a FPGA, variables become signals.

This is the first working version for the computation with VHDL implementation.