Study of the feasibility of accelerating MC generation with FPGAs

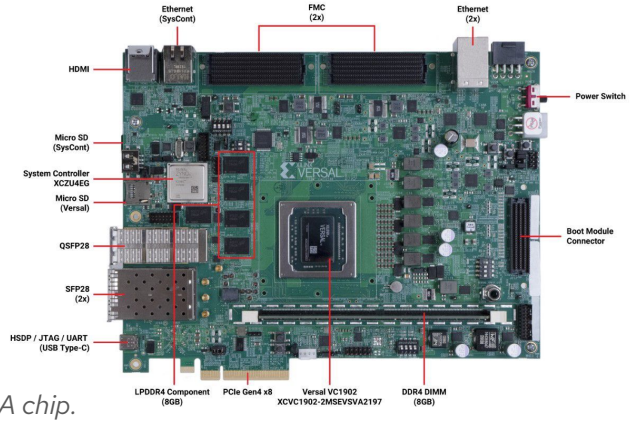
Madgraph5_aMC@NLO meeting at CERN

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 project

- **One liner:** to perform viability studies on whether **montecarlo (MC) generation** workflows can be accelerated **using *Field Programmable Gate Arrays*** (FPGAs).
- **Motivation:** ever demanding computational cost of MC generation lead the HEP community to develop MC@GPU (see talk from Stefan Roiser).
  - Very interesting results shown in CHEP2024 (see talk by A. Valassi – first release of the CUDACPP plugin).
  - Achieved overall speedups between ranging from x3 and x70!
  - In some cases not even dominated by the matrix element (ME) component of the computation.
- **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.

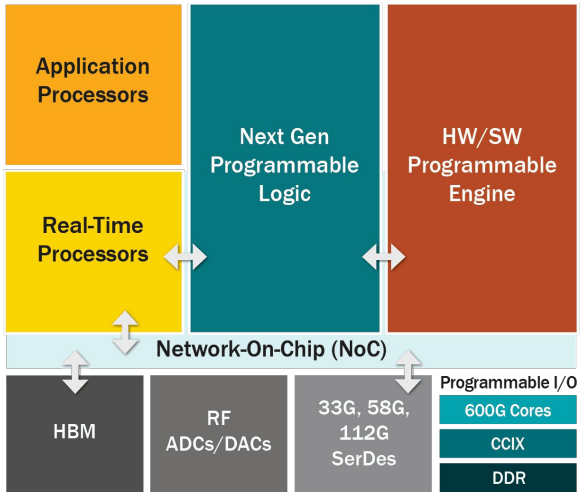


An electronic board with an FPGA chip.

What is an FPGA

- **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).



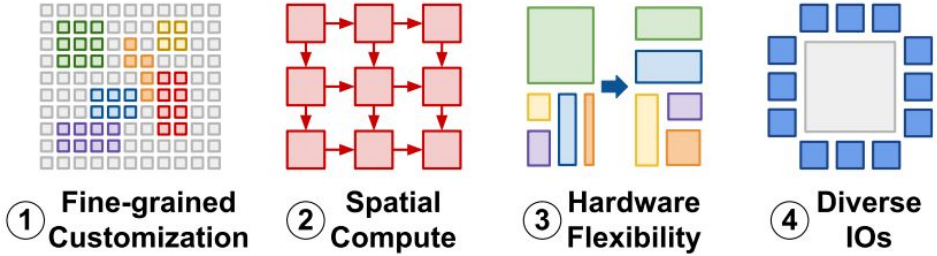


Figure from arXiv: 2404.10076 summarizing the advantages of FPGA solutions in Deep Learning. Applicable to any kind of task.

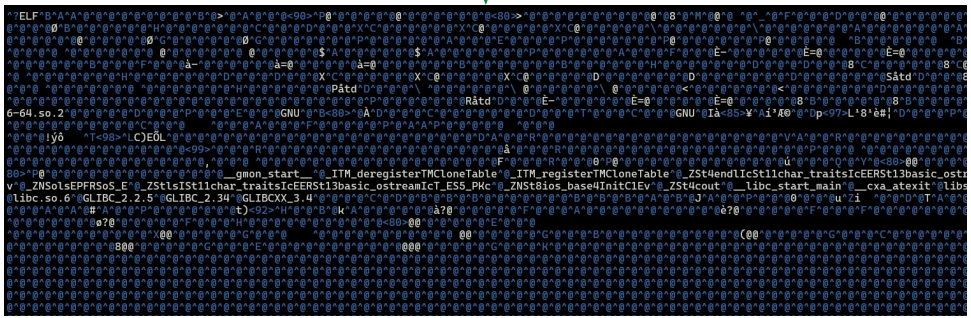
Rough idea on how an FPGA works

- **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

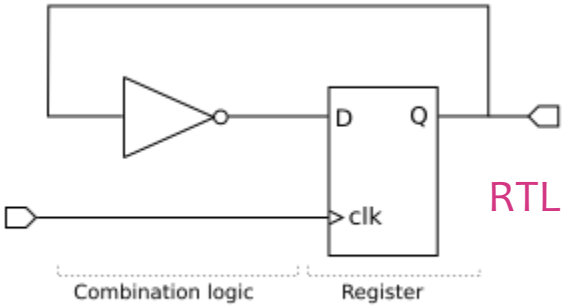


Rough idea on how an FPGA works

- **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;
```

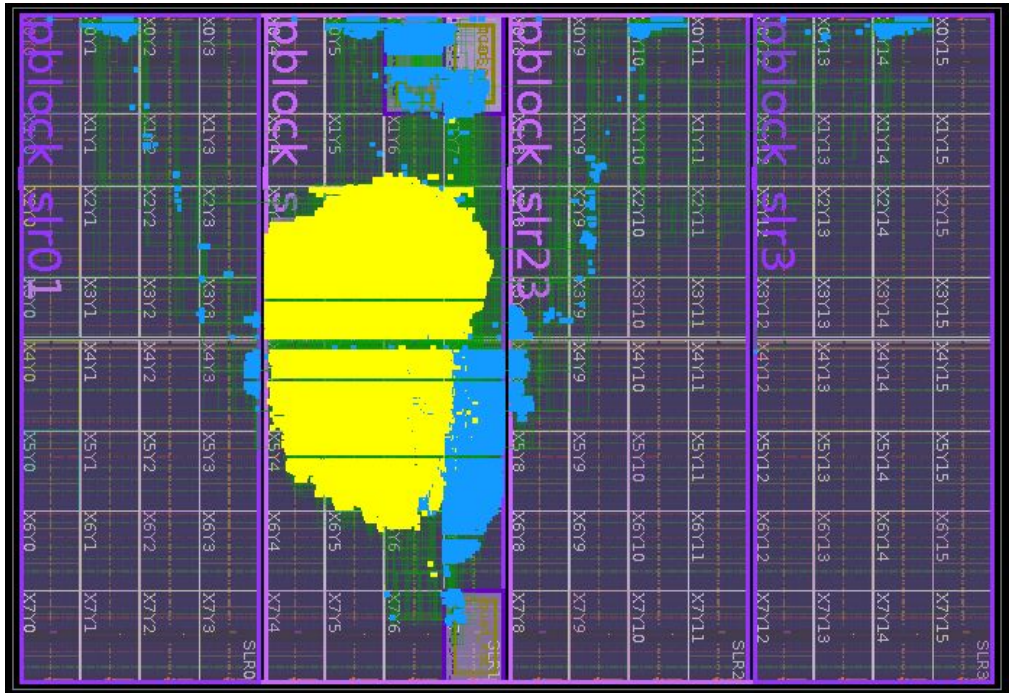
VHDL



RTL

Rough idea on how an FPGA works

- **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.

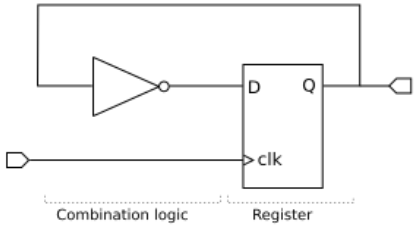


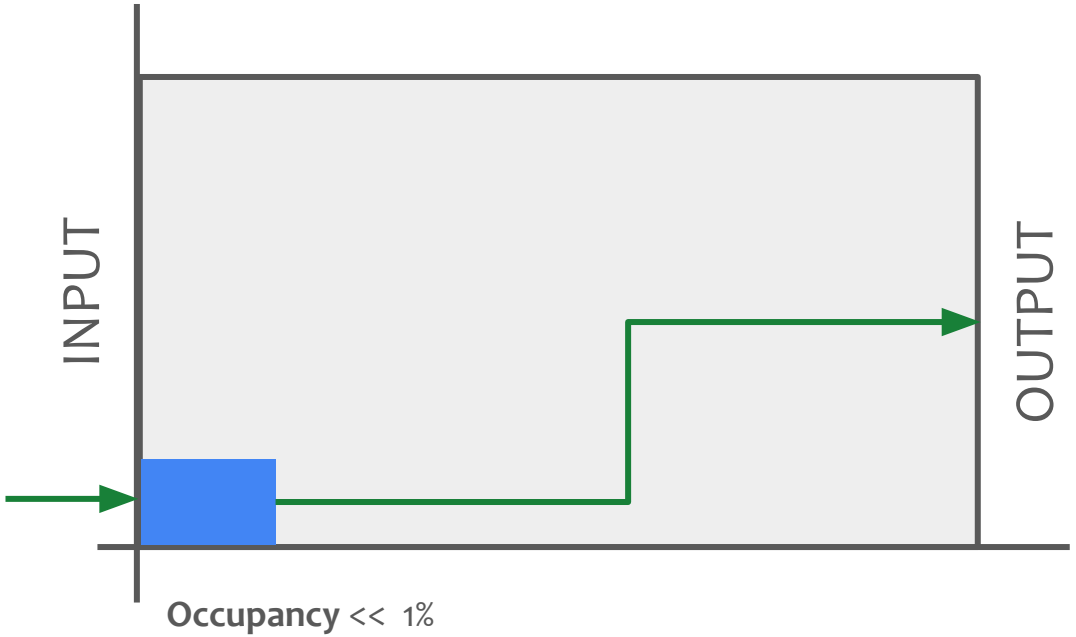
Example showing the implementation of an algorithm in a FPGA. The coloured regions show the occupancy of the FPGA. Blue: control framework, Yellow: algorithm.The FPGA will be using different hardware components such as Flip-Flops (FFs), Ram blocks (BRAMs), Digital Signal Processors (DSPs)

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.

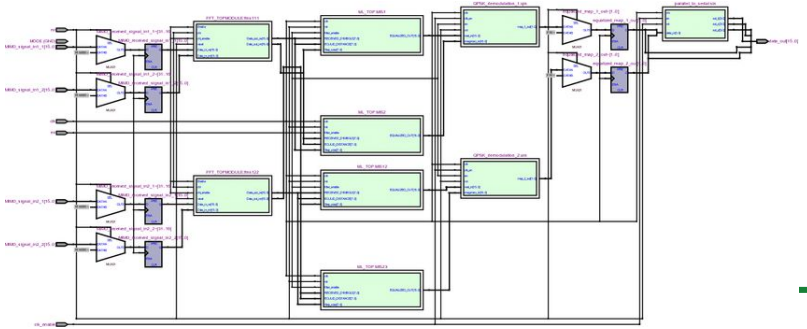
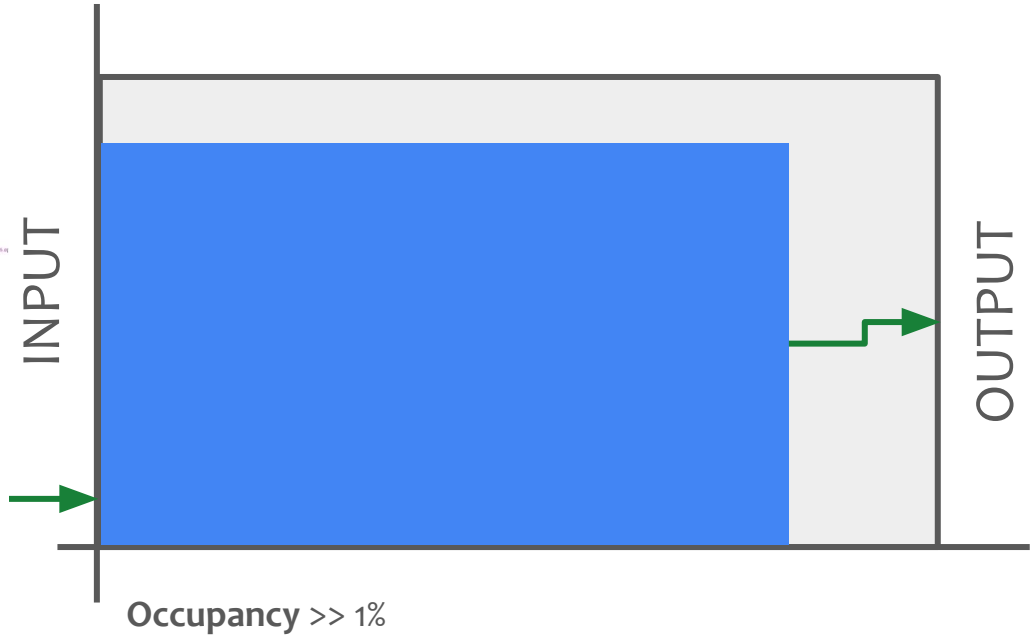


Image used for illustration purposes of a more complex operation.



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.

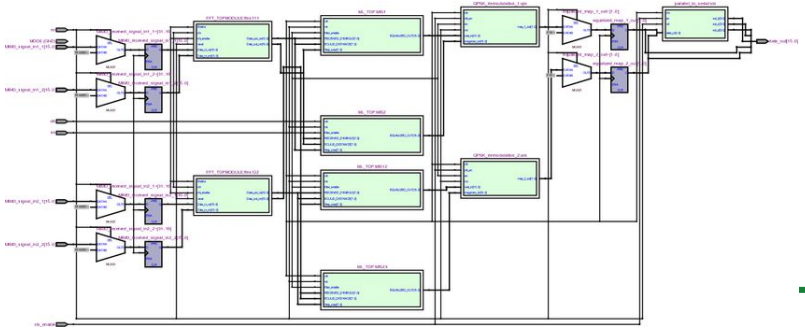
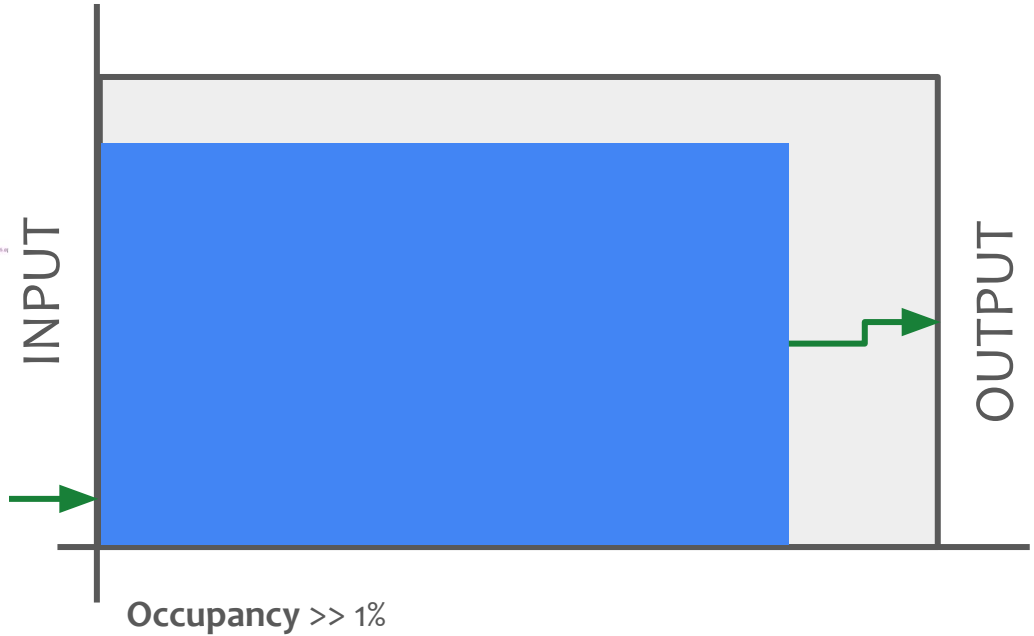


Image used for illustration purposes of a more complex operation.



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.

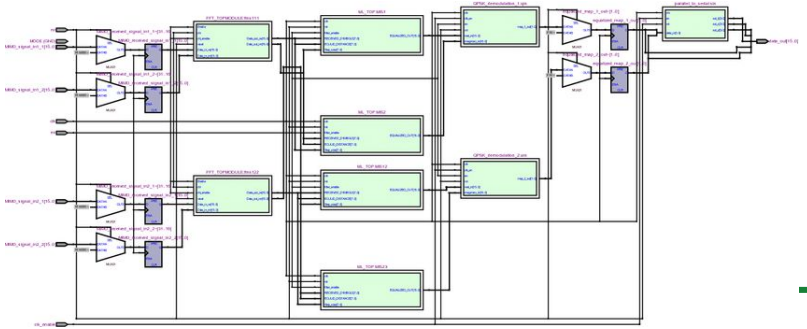
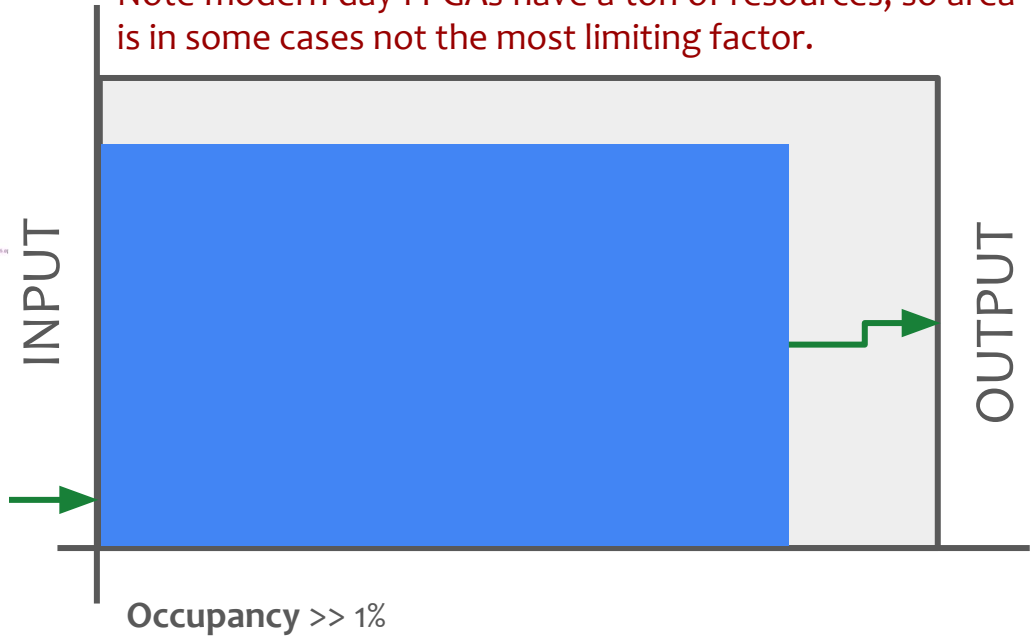


Image used for illustration purposes of a more complex operation.

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

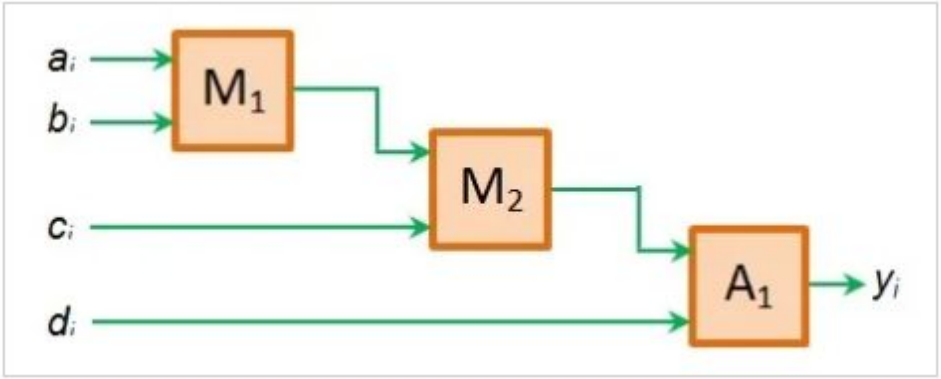


Considerations when developing a FPGA application

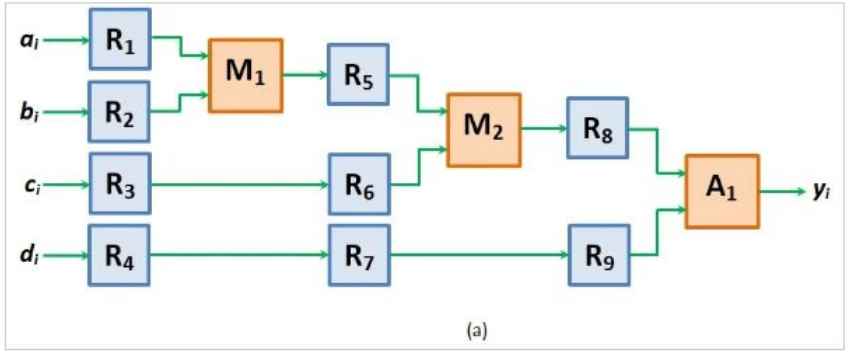
- **There are two main considerations to be taken into account.**
  - **Area constraints:** do we have enough resources (LUTs, FFs, BRAMS, DSPs, etc…) to perform the task?
    - The more operations we have to do (in parallel) → the more resources we consume.
  - **Timing constraints:** algorithm meets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



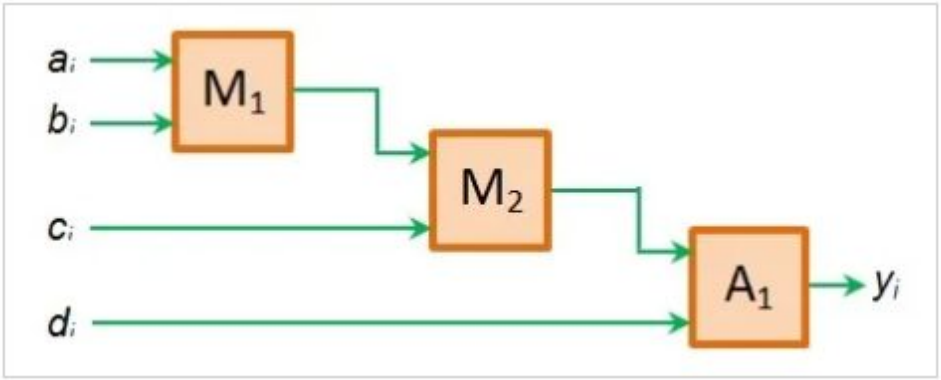
*Example of a **non-pipelined** designed. This design requires 3 clocks to finish, but only 1 is used, leading to time slack.*



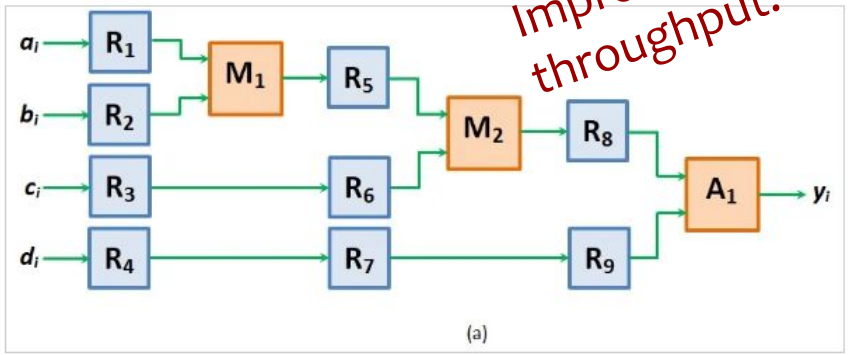
*Example of a **pipelined** designed. This design uses more registers to keep processing inputs while the final result of the first set of inputs is not finished yet. **Consequence: more area.***

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.*



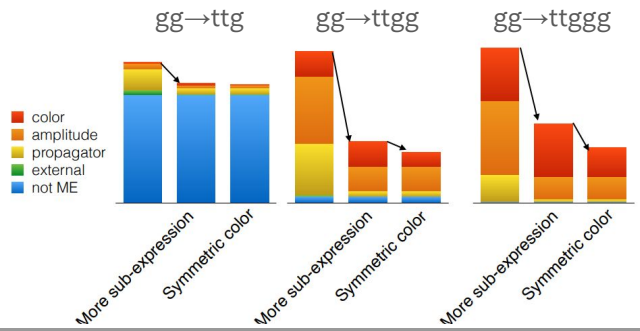
*Example of a **pipelined** designed. This design uses more registers to keep processing inputs while the final result of the first set of inputs is not finished yet. **Consequence: more area.***

Coming back to the physics

- Having covered what our considerations when developing an FPGA application. We come back to the problem at hand → **MC Generation**.
- The distributed computations in **`Madgraph5_aMC@NLO`** make **FPGAs** an – **a priori** – **suitable choice**.
  - ✓ Computation is divided into subfolders (`SubProcesses`) and integration channels.
  - ✓ The core driver runs a loop for the integration of each channel. This is something we can optimize using pipelining.
- **Our limitations can be:**
  - ✖ The final output requires double precision. Not impossible, but probably won't lead to as good as results as a GPU because of area and latency constraints of double precision operations in FPGAs.

- What we have explored (thanks to Olivier Mattelaer for the guidance!) → the color matrix computation.
  - Not affected by double precision ✓
  - Code is easy to port to VHDL (as it is just a few lines of fortran code that perform a loop over the decomposed color matrix). ✓
  - We can easily measure times in CPU and FPGA to compare performances. ✓

talk from Olivier at CHEP



A look into the computation (code wise)

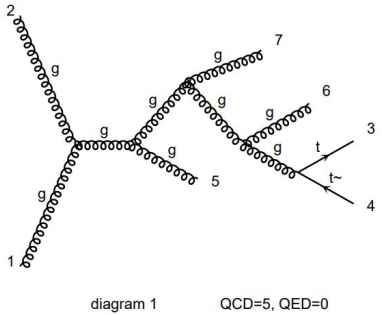
- We focus all of our examples in one of the main processes used in experimental High Energy Physics: ttbar multijets (considering up to 3 jets).

process card

```
import model sm-no_b_mass
generate g g > t t~ g g g
output gg_tt3j_LO
```

P1_gg_ttxggg

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




```
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

- 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 `MATRIX1`)

Implementation of the color decomposition in FPGA (VHDL)

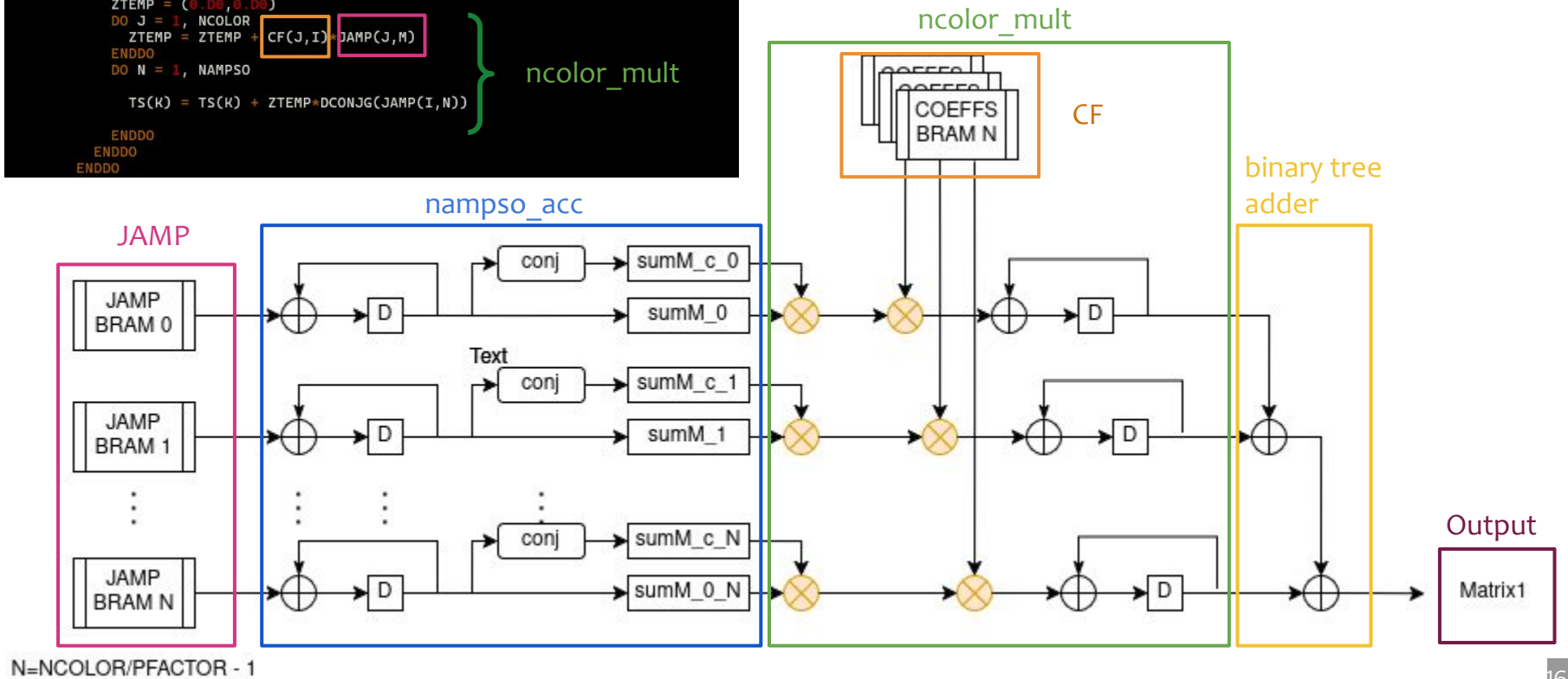
- **First step:** translation to VHDL

```
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

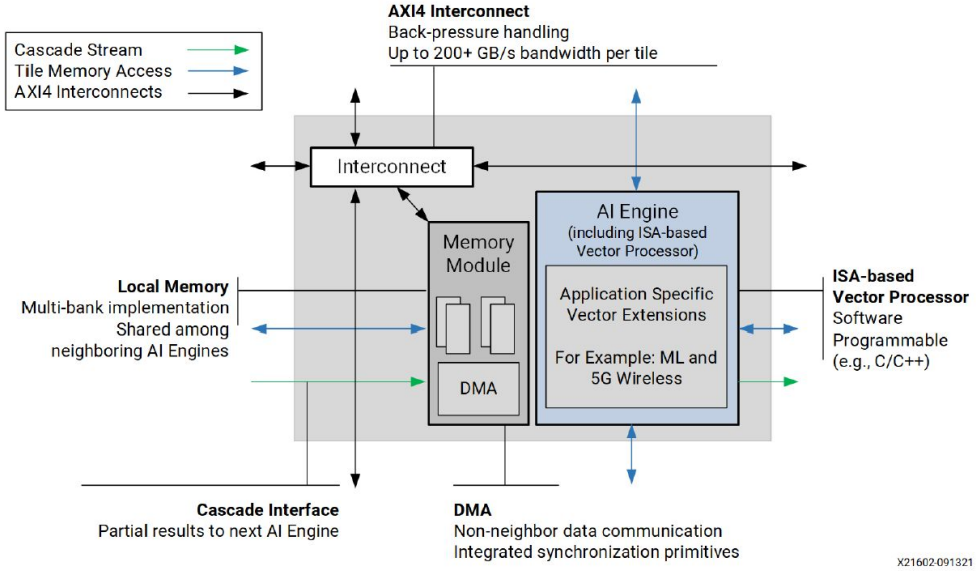
Using an optimization for these kind of computations from Sign Process Syst 95, 543–550 (2023)



Implementation of the color decomposition in FPGA AI Cores

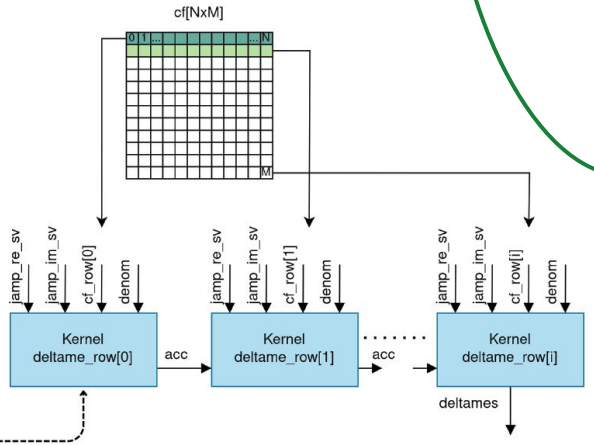
- The previous implementation is done using **Digital Signal Processors** (DSP)**, Blocks of ram** (BRAM)**, Lookup Tables** (LUTs), and other kids of **hardware components.**
  - **Problem:** DSP do not handle well floating point precission.
- **Modern day FPGA boards** (see Versal FPGA) include what 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.



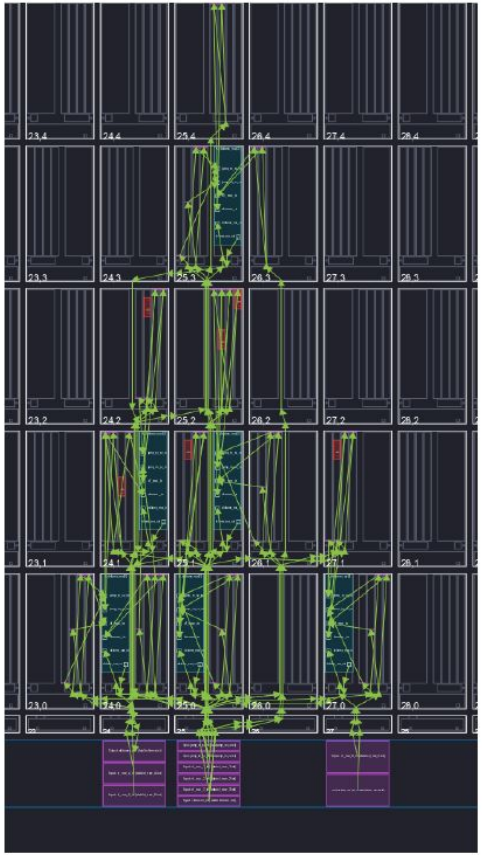
Implementation of the color decomposition in FPGA AI Cores

- 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)



$$acc = acc + \frac{[\sum jamp_re_sv * cf_row] * jamp_re_sv(i) + [\sum jamp_im_sv * cf_row] * jamp_im_sv(i)}{denom(i)}$$

Occupation diagram



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

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.
- **Any feedback from Madgraph experts is more than welcome!**

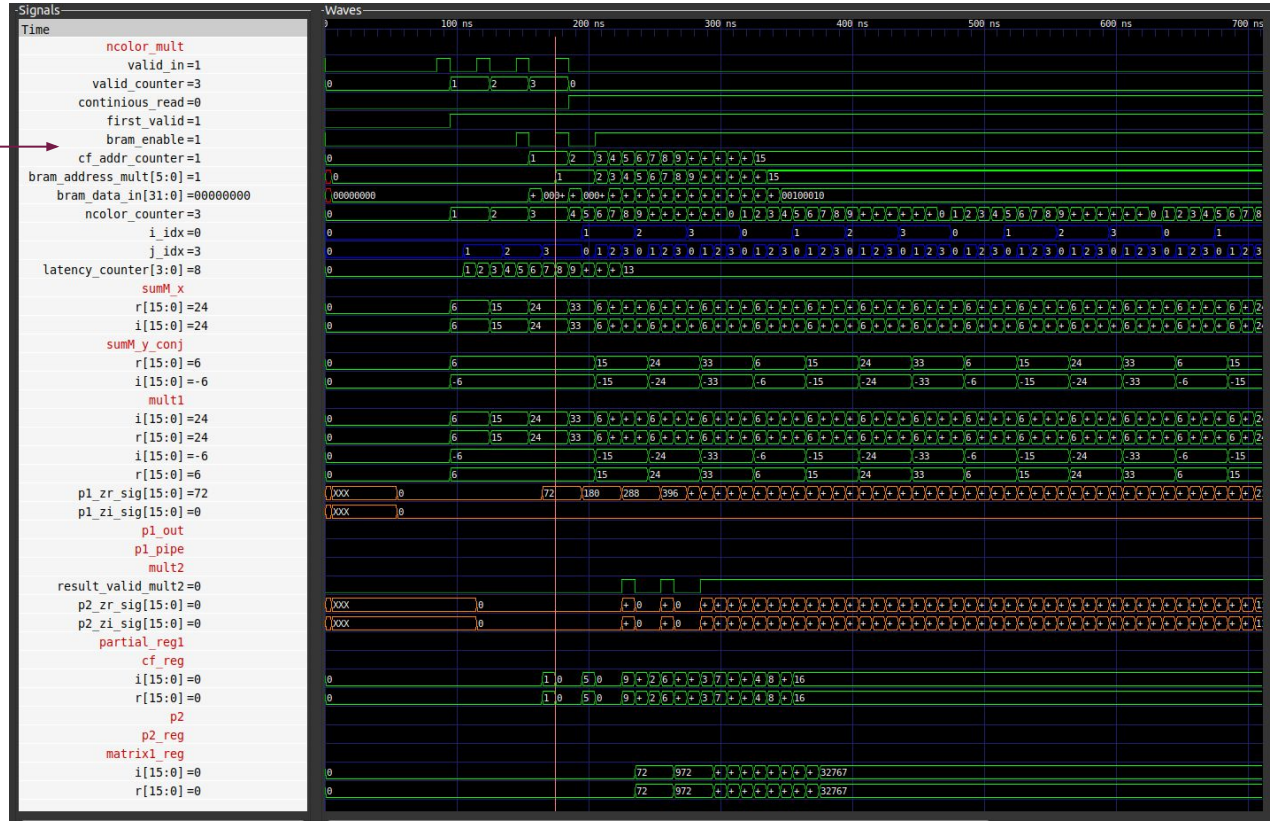
`carlos.vico.villalba@cern.ch`

Backup

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.