Università degli Studi di Perugia

Dipartimento di Fisica e Geologia

Sviluppo del firmware di comunicazione FPGA-FPGA ad uso di esperimenti di nuova generazione

Relatore
Dott. Matteo Duranti

Correlatore
Mirko Mariotti

Laureando
Alessandro Kotchian

N.Matricola
342068

Academic Year
2023-2024

A mia madre,
a Greta,
a tutti i miei amici,
che mi sono stati vicini
nei momenti duri
e nei momenti di gioia

Contents

1
Introduzione

Dal mio interesse verso gli FPGA e il loro fondamentale utilizzo nella fisica moderna è nata questa idea di tesi finalizzata a contribuire al progetto ”BondMachine” attualmente in sviluppo.
Le classiche architetture come quella di Von Neumann, usate dai desktop e laptop,
e quella di Harvard, usate dai microcontrollori, schematizzano l'organizzazione e il funzionamento del hardware di un sistema informatico. I software, come i sistemi operativi o applicazioni, si adattano poi a questa struttura per svolgere operazioni ed eseguire istruzioni. L'obiettivo di questo progetto è la creazione di una nuova architettura che riesce a tenere passo ai nuovi tipi di carichi di lavoro e di ridurre il gap tra hardware e software. Si punta a raggiungere questo con l'utilizzo di vari tools che creano e simulano l'architettura in tutto il suo funzionamento.
Il progetto è sviluppato in diversi linguaggi di programmazione, in particolare Go, Verilog e VHDL, utilizzati nella progettazione della struttura interna degli FPGA, ma soprattutto in ambito della comunicazione tra vari dispositivi.
Il mio ruolo nel progetto è stato creare un modo per far interfacciare un cluster di FPGA, permettendo di scomporre un processo in più parti cosi da aumentare la potenza di calcolo.

2

Teoria delle FPGA

Le componenti che normalmente sono usate per il calcolo sono: Central Processing Unit (CPU), il processore standard che si trova all'interno di un qualsiasi calcolatore e che esegue in maniera sequenziale le istruzioni assembly che compongono i suoi programmi; la Graphics Processing Unit (GPU), nata per elaborazione 3D ma ora ampiamente utilizzata per il calcolo matriciale.
L'ultimo sviluppo è quello di utilizzare per il calcolo le FPGA.

Gli Field Programmable Gate Array (FPGA) sono dispositivi logici programmabili nati nella seconda metà degli anni '80 dall'azienda Harris Semiconductor. La loro evoluzione, grazie alla competizione iniziale tra le diverse aziende come IBM, Motorola e Toshiba, ha portato all'aumento delle capacità computazionali di un fattore 10000 e la performance di un fattore 100, ma questo non ha portato ad un aumento del costo o del consumo, anzi, sono diminuiti anche loro di un fattore 1000 [14].

Gli FPGA vengono utilizzati anche per l'estrema versatilità, dovuta al fatto che l'hardware è riconfigurabile, a differenza di un Application Specific Integrated Circuit (ASIC), la cui struttura, per fare un dato lavoro, deve essere progettata specificamente. Questa riconfigurabilità è, ad esempio, particolarmente utile qualora in fase di progettazione si sia effettuato un errore di disegno: con un ASIC bisognerebbe ripartire dalla fase di disegno e, soprattutto, effettuare una nuova produzione, mentre con un FPGA basterebbe correggere il codice e caricare nel dispositivo il nuovo firmware. In figura 2.1 sono mostrati i principali utilizzi degli FPGA.

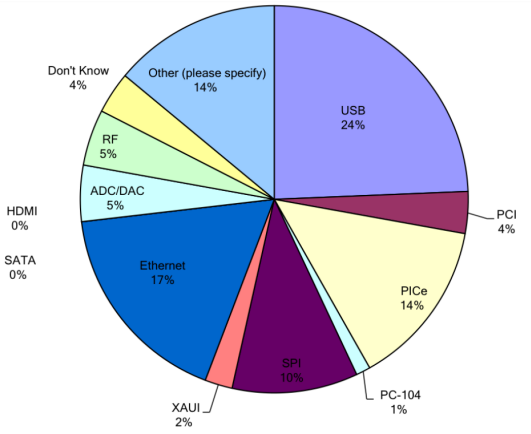

Figure 2.1: Utilizzo degli FPGA nei vari progetti

2.1 Struttura FPGA

La struttura di un FPGA è composta da una matrice di Configurable Logic Blocks (CLB) collegati tra di loro con connessioni programmabili, come in figura 2.2. La matrice realizza le funzione logiche e queste ultime vengono trasmesse attraverso linee di interconnessione lunghe o brevi.
La differenza tra le linee sta proprio nel limite massimo entro i quali due CLB possono essere connessi: quelle brevi sono usate solo per due CLB vicini, questo consente di avere bassa latenza e basso ritardo del segnale, mentre quelle lunghe vengono usate per due CLB distanti al massimo ”6 CLB”, per minimizzare il ritardo e non renderlo eccessivo.
Le celle logiche, che compongono i CLB, eseguono operazioni booleane memorizzandole al loro interno. Sono composte principalmente da una o più Look-up Table (LUT), le quali permettono di mappare funzioni che prendono degli input e restituiscono uno o più output, e da un registro, precisamente un Flip-Flop di tipo D (con set e clr asincroni). Ai margini dei CLB ci sono gli Input Output Block che hanno il compito di far interfacciare l'FPGA con i dispositivi collegati a esso e, per velocizzare la comunicazione, tutti i CLB sono collegati ad una RAM che agisce da buffer.
Tutta questa architettura implica che un FPGA non ha un set di istruzioni di base e la memoria ha una gerarchia variabile permettendo così di eccellere in tutti i tipi di parallelismo.

2.2 Programmazione

I principali software di programmazione sono prodotti dalla stessa casa madre che produce gli FPGA, le più rinomate sono AMD/Xilinx ed Intel/Altera che usano differenti software ed entrambi usano linguaggi di tipo Hardware Description Language (HDL) come Verilog e VHDL.

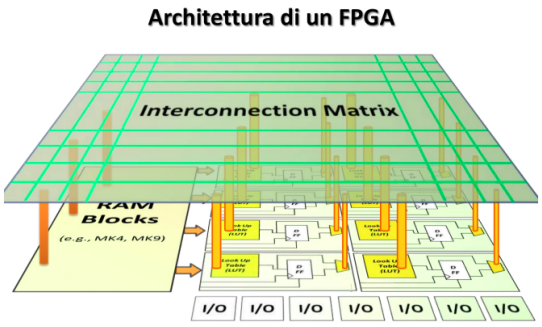

Figure 2.2: Un esempio di matrice che compone un FPGA

Mentre nei linguaggi di programmazione ad alto livello come C, C++ e Java, le istruzioni sono eseguite sequenzialmente da una struttura come una CPU, gli HDL definiscono blocchi di hardware, e le istruzioni non vengono eseguite in modo sequenziale ma in parallelo e senza run-time. Il processo di sintesi, fatto dal software, trasforma una descrizione HDL in una netlist (componenti elettronici fisici e loro collegamenti) e, attenendosi ad un template preciso deciso dalla casa madre, vengono analizzati e riconosciuti dai sintetizzatori che danno luogo alle porte logiche corrispondenti. Il sintetizzatore trasforma il testo in uno schema a blocchi e in fase di Mapping, usa le celle logiche degli FPGA per mappare le funzionalità descritte sull'hardware.
Oltre a questo il software prende gli elementi di memoria di alto livello, come ad esempio registri e contatori, e li riconduce ad un Flip-Flop. Le equazioni descritte dal programma, vengono ottimizzate e mappate sulle LUT delle celle logiche.
Tutto questo lavoro di controllo, lettura, ottimizzazione e creazione, seguita da un'ulteriore ottimizzazione porta ad avere tempi di compilazione molto elevati che, a seconda della struttura del programma, possono durare dai minuti alle ore.

2.3 Verilog

L'FPGA che ho utilizzato nel mio lavoro è stata costruita dall'azienda Xilinx e per generare il firmware si usa come software editor Vivado, che si basa sul linguaggio Verilog.
La sintassi di Verilog è molto simile al linguaggio C e le parole chiave per il controllo del flusso del programma sono `if` e `while`.
Un progetto Verilog consiste in un insieme di moduli legati da una certa gerarchia. Ogni modulo può contenere funzioni di altri moduli così da poter facilitare la progettazione, inoltre è presente un insieme di input e output (I/O) descritti da un file *.xdc* che permette al software di conoscere i pin o I/O fisici

della board su cui è montato un FPGA. Come esempio è riportata qui sotto una parte del file *.xdc*:

```
set_property -dict { PACKAGE_PIN W5   IOSTANDARD LVCMOS33 } [get_ports clk]
create_clock -add -name sys_clk_pin -period 10.00
-waveform {0 5} [get_ports clk]
```

Nell'esempio, il software leggendo il file *.xdc* riesce a capire le caratteristiche dell'FPGA come il clock, la frequenza di clock e il tipo di forma d'onda.
In generale un programma è strutturato come segue:

```
module nome_programma(
//output variabili_uscenti,
//input variabili_entranti)
begin

always @ (//condizione_del_while
) begin
//cosa_fa_il_programma;
end
endmodule
```

Le variabili si classificano come fisiche ,ad esempio `reg` e `wire`, oppure costanti, vettori o interi.
Concentriamoci sulle variabili fisiche:

- `wire`: sono collegamenti utilizzati per connettere componenti. Questi possono essere da 1bit oppure più bit. Un esempio `wire [0:7] x`, crea un collegamento denominato "x" di dimensione 8 bit;
- `reg`: dall'inglese "register", sono dati che vengono immagazzinati in una parte della memoria del dispositivo. Un esempio `reg [0:31] v`, crea un registro da 32 bit denominato "v".

Una volta completato il codice, si può creare il circuito fisico partendo dal processo di implementazione che prevede varie fasi:

1. sintesi: il codice scritto viene tradotto in una rappresentazione di componenti hardware di basso livello, creando le porte logiche (AND, OR, ecc...), flip-flop o strutture per gestione dati, che vengono interconnesse a seconda del comportamento descritto dal codice;
2. mappatura al Dispositivo di Destinazione: la rappresentazione viene mappata sulla specifica architettura del dispositivo di destinazione, tenendo conto delle restrizioni del dispositivo stesso;

3. ottimizzazione: il circuito viene ottimizzato per ridurre il numero di risorse, minimizzare il ritardo e per bilanciare la distribuzione delle risorse in modo da evitare problemi di performance;
4. ulteriore controllo errori (Timing Analysis): viene fatto un controllo sui tempi con cui i vari segnali vengono modificati dal programma per verificare che rispettino i vincoli del dispositivo;
5. generazione Bitstream: se il codice passa il controllo finale, l'implementazione viene salvata in un file *bitstream*, che può essere caricato, sotto forma di firmware, nell'FPGA ed è questo che effettivamente va a modificare l'hardware.

Come già detto in precedenza, il tempo che il software impiega per compiere tutte le fasi precedentemente descritte può variare in base alla complessità del programma e può passare da minuti ad ore.
Una fase fondamentale del processo di implementazione è creare dei file detti *testbench* che permettono di simulare la risposta del modulo HDL in sviluppo. Questi file, scritti nello stesso linguaggio del modulo, creano un ambiente virtuale e attraverso dei segnali detti stimoli verificano se il modulo si comporta coerentemente con le attese. Anche i *testbench* hanno una struttura definita:

1. stimoli: vengono definiti tutti gli I/O necessari per il programma;
2. controllo dell'output: il testbench deve controllare se il programma risponde correttamente;
3. controllo dei cicli: il testbench controlla come si comporta il programma ad ogni ciclo di clock.

3
BondMachine

Con la richiesta sempre maggiore di potenza di calcolo, i sistemi moderni si scontrano con i limiti della tipologia di carichi processabili, in quanto la GPU è in grado di processare efficacemente soltanto alcuni specifici task, mentre la CPU, per quanto sia multi-purpose, risulta comunque lenta se comparata ad un acceleratore hardware. La nascita di nuovi tipi di carichi di lavoro (come il Machine Learning) ha dimostrato come questo tipo di architettura non sia più sufficiente.
Il progetto BondMachine ha come obiettivo quello di sviluppare un nuovo tipo di architettura per calcolatori da utilizzare su FPGA. Tramite un framework si può sia generare queste architetture che il software che sarà eseguito su di esse. L'obiettivo di tutto ciò è quello di creare dei sistemi hardware/software che abbiano migliori prestazioni e migliore efficienza energetica rispetto a una soluzione standard.
BondMachine[11] è un framework che può generare architettura, in questo senso è diverso da CPU e GPU che hanno delle architetture ben codificate e statiche. La BM può essere schematizzata come un insieme di Connecting Processors (CPs) e di Shared Objects (SOs) che rendono il sistema flessibile.
I CP implementano le istruzioni necessarie per costruire le funzionalità dell'intero sistema e giocano il ruolo di elaboratori. Ognuno di essi è costruito per compiere un singolo compito con un piccolo **instruction set**.
Gli **OPCODE** sono le istruzioni che un determinato processore può eseguire, anche i processori della BM utilizzano questo concetto. Tuttavia, non tutte le istruzioni sono attivate all'interno di un CP. Questa è la principale forma di configurazione dei CP. L'insieme delle operazioni eseguibili costituisce l'**instruction set** del CP. In definitiva in una BM con più CP questi potrebbero avere un diverso **instruction set**.

Infatti, ogni singolo core generato all'interno della BM, come si può notare in figura 3.1, viene progettato per contenere un **instruction set** necessario a svolgere un determinato compito e può essere connesso ad altri core attraverso speciali registri I/O, integrando anche dispositivi esterni.
Ad ogni CP è presente anche una RAM utile come buffer nel processo di calcolo.

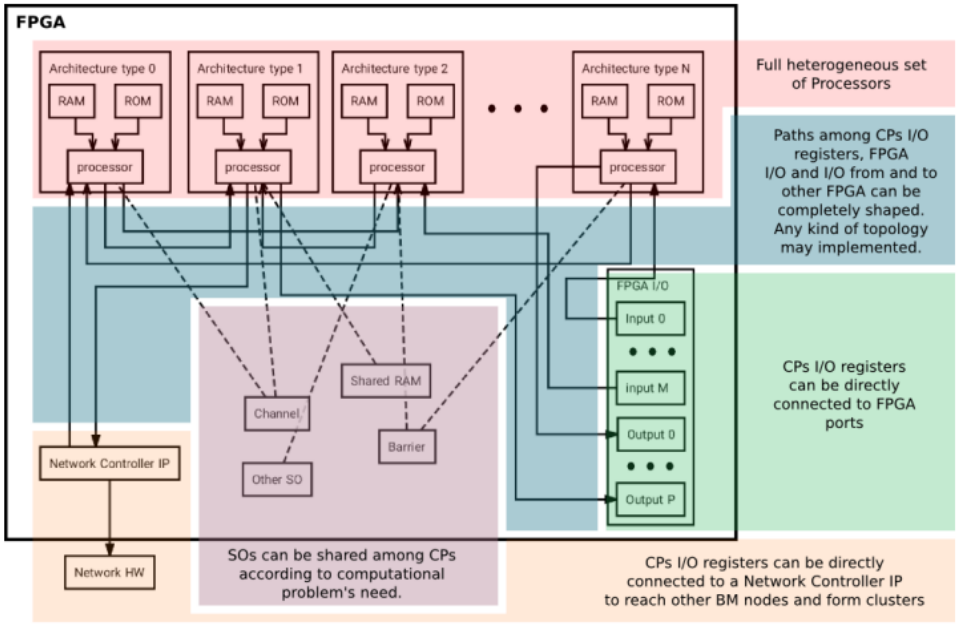

Figure 3.1: Il design fatto da BondMachine di un FPGA descritto da un diagramma a blocchi

Diversi CP possono essere collegati con oggetti condivisi come gli SO. Gli SO sono oggetti che espandono le capacita elaborative migliorando comunicazione e sincronizzazione tra i CP. Alcuni di questi sono:

- channels: permettono la comunicazione tra due o più CPs;
- memorie Condivise: semplicemente una RAM condivisa da più CPs. Hanno l'accesso a dati in entrata e in uscita e di poterli modificare. La loro dimensione dipende dalla task che le CPs connesse devono fare;
- barriere: sincronizzano i vari processori in modo da dare il risultato completo solo una volta che tutti hanno terminato il lavoro. Quando un processore raggiunge una Barriera, esso viene posto in uno stato di inattività fintanto che tutti gli altri processori raggiungono la stessa Barriera;
- generatore di Numeri Pseudo-casuali: utilizzato per generare numeri utili ai CPs connessi ad esso.

Alcuni SO utilizzano **OPCODE** per compiere certe task come nella comunicazione attraverso canali e barriere. Sono stati sviluppati, principalmente in linguaggio Go, dei toolkit per poter creare la struttura, modificarla facilmente e testare la sua funzionalità. I toolkit messi a disposizione sono in grado di generare una BM su più dispositivi differenti, consentendo l'esecuzione simultanea sugli stessi di un file binario.
Ogni FPGA deve contenere un Connection Component (CC) cosi da poter creare una rete di comunicazione. Per questo è stato realizzato il protocollo di rete chiamato "Etherbond" per la comunicazione di più dispositivi attraverso chip di rete come wireless o wired. I tool principali, scritti in Go, sono 3: "Processor builder", che gestisce la configurazione dei parametri dei CPs, "BM Builder", che gestisce le interconnessioni tra le CPs e gli SOs, e "Simulazione ed emulazione", in grado di testare la BM senza un FPGA.
Il tool "Processor builder" crea e gestisce i CPs ed utilizzando la sintassi *procbuilder -registers *regNumbers* –opcodes *opc1, opc2, ...** il tool genera un CP con dimensioni dei registri, il numero di registri, la quantità di memoria da allocare, il numero di registri di I/O e gli **OPCODE** necessari. Ad esempio: "procbuilder -registers 3" crea un CP da 8 registri a 3-bit. Ogni modifica fatta dall'utente può essere salvata in un file formato JSON che registra i diversi CPs e può essere letto anche dalle altre applicazioni. Il programma scritto in assembly viene tradotto in linguaggio macchina per poi essere eseguito nel CP generato. In fine il tool genera il codice HDL per essere implementato nell'FPGA.
Il tool "Bondmachine builder" è utilizzato per l'interconnessione dei CP generati dal "Processor builder" ed i vari SO. Il comportamento è analogo al "procbuilder", ma si gestisce l'interconnessione dei componenti anziché il singolo CP. Il tool permette di aggiungere o rimuovere CPs, creare interconnesioni tra registri I/O e CPs, la definizione di SO e la loro condivisione e generazione di codici Verilog per essere caricati nell'FPGA.
Per emulare l'architettura è stato costruito "simbox": questo permette di testare il comportamento su una workstation Linux in assenza di dispositivi fisici. Viene offerta anche la possibilità di eseguire parte di codice sulla workstation e parte di codice su FPGA. L'emulazione è possibile eseguirla anche con "Processor builder" e "Bondmachine builder" allo stesso modo descritto precedentemente.
Per semplificare il processo di costruzione di BM, sono stati sviluppati dei pacchetti software a più alto livello. Il compilatore "BondGO", che si occupa di generare la BM ottimale, ed il relativo codice assembly, a partire da un file di input scritto in GO. Grazie alla semplicita del linguaggio, l'utente ha solo bisogno di scrivere la task che deve eseguire la BM, farla compilare da "BondGO" per poi ottenere un codice da eseguire o da caricare. L'utilizzo di questo linguaggio è dovuto alla sua capacita di integrare nativamente l'analisi di qualunque codice sorgente e di utilizzare strutture come "GOroutine" e "Channels" permettendo cosi una corrispondenza uno ad uno con CP e "Channels". Inoltre alcuni pacchetti di base possono essere usati per costruire la struttura necessaria in assembly.
Riportando un lavoro svolto dal gruppo di ricerca dell'Università degli Studi di Perugia, si è generato il

codice register transfer level (RTL), che descrivere il funzionamento del circuito, di un FPGA Xilinx (Xilinx KC705 evaluation card) per valutare la performance di una BM. L'architettura dell' FPGA è composta da 256 CPs, 128 canali, con un clock di 200MHz e un consumo di 6.13W su cui è stato fatto un benchmark con CPU come i7-2600K a 3.4GHz e un Xeon E52430 v2 a 2.5GHz.
Come si può notare in figura 3.2, e come ci si aspetta, la BM sfrutta le caratteristiche del FPGA: il tempo di operazioni delle CPU aumenta linearmente con il numero di CP emulati e ciò è dovuto alla loro natura sequenziale, invece il tempo di operazione dell'FPGA rimane costante visto il parallelismo intrinseco del sistema. Si aspetta tuttavia che questo rimanga vero finché tutta l'FPGA non sia completamente riempita.

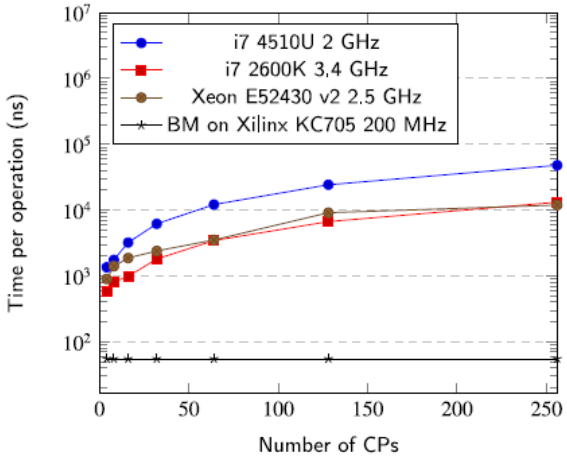

Figure 3.2: Tempo di operazione necessaria alle differenti architetture

Grazie alla flessibilità dell'architettura, siamo in grado di costruire dinamicamente l'hardware più adatto all'esecuzione di un determinato software ogni volta che ne abbiamo bisogno, ottimizzando le risorse a disposizione del sistema[13] come ad esempio dispositivi del Internet of Things (IoT), Cyber Physical System e High Performance Computing.

3.1 Machine Learning

Il modello di processori eterogenei interconnessi, su cui si basa il design di BondMachine, è chiaramente il framework perfetto per eseguire carichi di lavoro moderni come quelli associati al Machine Leaning (ML) e Grafici computazionali. Per sfruttare questa opportunità, il gruppo, ha sviluppato diversi strumenti di livello superiore. Tutti condividono la capacità di generare BondMachine partendo da fonti diverse, consentendo la creazione di grafici o reti neurali sotto forma di codice HDL.
L'applicazione "Neuralbond" permette di creare reti neurali sulle BM.
Proprio come i neuroni di un cervello, i CPs vengono generati dall'applicazione usando le diverse librerie del linguaggio Go.
Molto spesso gli FPGA vengono messi a svolgere il ruolo di acceleratori dell'hardware, cioè posti a fianco a computer o dispositivi che hanno bisogno di potenza di calcolo. Il gruppo di Perugia si è mosso per creare un interfaccia tra FPGA e computer con sistemi operativi come Linux. L'Interfaccia, sviluppata con linguaggio C, deve tenere conto anche del modo in cui sono connessi i due dispositivi, come ad esempio un PCI express bus o attraverso un cavo ethernet.
Nel caso in cui l'FPGA sia connessa su scheda PCIe non esiste una comunicazione diretta tra l'FPGA e il processore. È quindi necessario un driver PCIe per farli interfacciare tra loro.
L'obiettivo finale del gruppo è quello di portare il progetto anche nell'ambito della ricerca in fisica.

4 Progetto

Algoritmi sempre più complicati e enormi quantità di dati da leggere ed elaborare richiedono una sempre maggiore potenza di calcolo. Il singolo calcolo su FPGA non basta, per la facilità con cui si può occupare un intero dispositivo, e per questo si opta per sistemi multi-FPGA che, comunicando tra loro, si suddividono il compito risolvendolo velocemente. Il mio ruolo nel progetto di Bondmachine è stato sviluppare un modulo di comunicazione tra FPGA o diversi componenti dello stesso FPGA che svolgono funzioni differenti nel calcolo complessivo. La comunicazione tra FPGA differenti era già stata fatta dal gruppo di ricerca come riporta la tesi [13] dove i dispositivi erano stati collegati attraverso uno switch di rete e i segnali venivano trasmessi tramite quest'ultimo.
Tuttavia l'uso di switch o server per mettere in rete i dati è molto meno performante rispetto ad una connesione fisica tra i dispositivi. Per questo molti cluster sono interconnessi tra di loro attraverso dei cavi o bus seriali.
In quest'ottica si colloca il mio lavoro di tesi:

1. creazione moduli Verilog per la comunicazione filo-filo fra 2 differenti board
2. creazione di funzioni Verilog, `Receiver` e `Sender` per la comunicazione

4.1 Comunicazione diretta FPGA-FPGA

Nel mio setup sperimentale sono state utilizzate due Evaluation Board Basys 3 [4] con un FPGA Artix-7. Questa board presenta 33280 celle logiche, ognuna dotata di 6 input per ogni LUT e 8 FF,

una RAM di 1800 Kbits, un clock interno di 450MHz e, come mostrato in figura, presenta differenti dispositivi di I/O:

- 16 Switch
- 16 Led
- VGA port
- Display 7-seg per 4 cifre
- Usb port

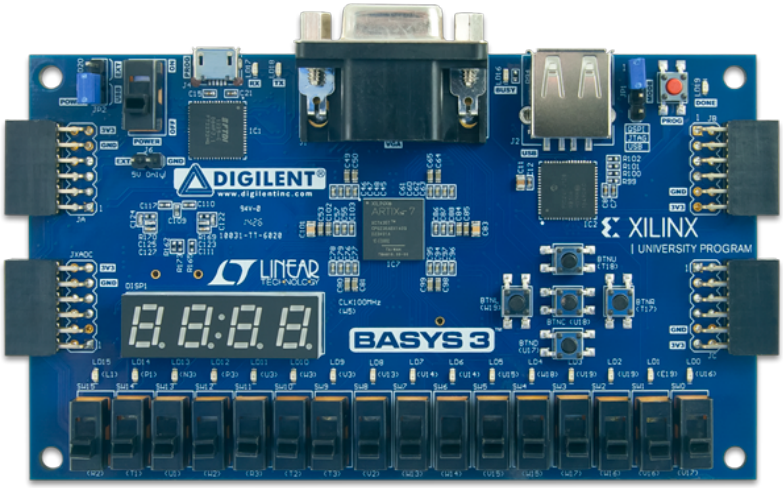

Figure 4.1: Basys 3 Artix-7 FPGA Trainer Board

Come mostrato in figura 4.2 le due board vengono connesse tramite cavi utilizzando i pin nominati `JB` per il `Sender` e `JXADC` per il `Receiver`. In seguito vengono riportati i diversi pin e i loro ruoli:

$$Sender: \begin{cases} JB0 = \text{clock signal} \\ JB1 = \text{message signal} \\ JB2 = \text{check signal} \\ JB3 = \text{reset signal} \end{cases} \qquad Receiver: \begin{cases} JXADC0 = \text{clock signal} \\ JXADC1 = \text{message signal} \\ JXADC2 = \text{data error signal} \\ JXADC3 = \text{reset signal} \end{cases}$$

4.1.1 Sender

L'FPGA che manda il segnale usa 3 differenti moduli Verilog: *counter*, *bond tx* e *parity*. Gli ultimi due moduli, *bond tx* e *parity*, vengono usati dal primo modulo attraverso le funzioni `btx` e `pt`.

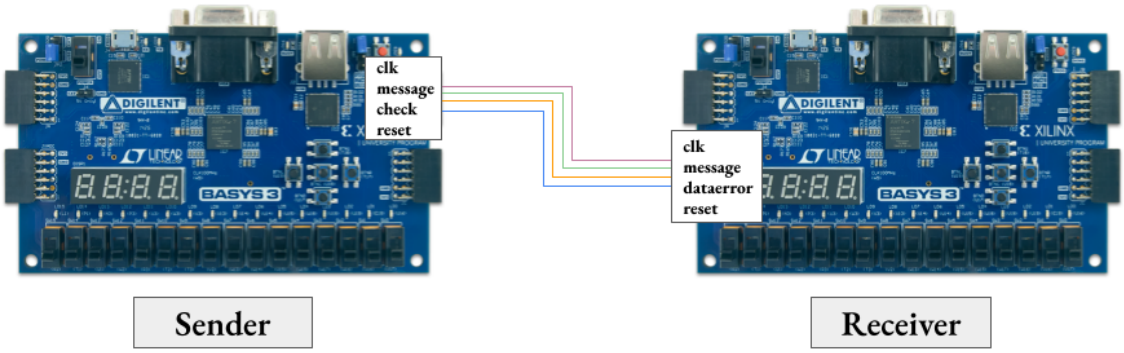

Figure 4.2: Setup sperimentale

La funzione `btx` ha 4 dati in input e 2 dati in output ed il suo compito è di mandare il messaggio ogni volta che ho un segnale che conferma la possibilità di inviare i dati, e di riportare alla board il suo stato di attività. Il messaggio e il clock, che sono i dati in output della funzione, vengono associati ai corrispettivi pin della board `JB1` e `JB0`.

```
1    bond_tx btx(.clk(clk), .message(par), .data_enable(trig),.busy(led), .tx_clk
     (tx_clk), .tx_out(tx_out));
2
3    parity pt (.a(mem), .b(par));
```

La funzione *parity* controlla se ci sono errori nel mandare il messaggio all'altra board. Questo controllo viene fatto aggiungendo 1 bit, detto bit di parità, al messaggio per rendere pari/dispari il numero di bit con valore 1.
In questo caso vengono confrontati il messaggio da inviare e il messaggio nel buffer per vedere se ci sono errori. Nella prima parte del codice viene creato un contatore che ogni volta che il ventiduesimo bit è uguale ad 1, viene inviato il segnale e viene mostrato anche il messaggio attraverso l'accensione dei led:

```
        if(check==0) begin
            counter<= counter+1;
            old <= counter[22];

            if ((old != counter[22]) && (counter[22] == 1)) begin
                trig <= 1;
            end
            else begin
                trig <= 0;
            end

            JB0<=tx_clk;
            JB1<=tx_out;
            check<=JB2;

            for (i=0 ; i<9 ; i=i+1) begin
                led[i] <= par[i];
            end
```

Come si può notare dalla linea di codice 14, la board controlla sempre che il segnale in `JB2` di errore rimanga effettivamente sempre spento. In caso di errore della board stessa o segnale di errore che arriva dall'altra board, la variabile `check` viene modificata e al il ciclo di clock successivo la board si ferma, si resetta e manda un segnale di reset anche all'altra board.

```
       if(check!=0) begin
            check=0;
            counter<=26'b0;
            old<=0;
            mem<=8'b0;
            JB3<=1;
          end
        else begin
            led<=4'b10;
        end
```

Una volta che l'intero setup è stato resettato, al ciclo successivo, il sistema torna a funzionare e il messaggio viene inviato nuovamente.

4.1.2 Receiver

L' FPGA che riceve il segnale usa gli stessi 2 moduli *bondtx* e *parity* mentre il modulo *Receiver* descrive il comportamento della board.

```
if(reset==0) begin
            old_data_ready <= data_ready;
            reset<=JXADC3;
            if (data_ready == 1'b1 && old_data_ready == 1'b0) begin
                rom<=message;
                for (o=0; o<=7; o=o+1) begin
                    led[o]<=rom[o];
                end
           end
       end
```

La board controlla ad ogni ciclo di clock se il segnale di reset è alto o basso, in quest'ultimo caso, memorizza il messaggio ricevuto nella variabile `rom` che poi verrà mostrato attraverso i led.
In caso di reset, al ciclo successivo non viene più verificato l'*if*, quindi ogni variabile sarà resettata e verrà poi inviato il segnale di conferma del reset. In caso di qualsiasi errore la board manderà un segnale attraverso `JXADC2` e questo verrà letto dall'altra board.
Il messaggio usato era composto da 8 bit, a cui sono stati fatti corrispondere i primi otto led della board per poter verificare il funzionamento del sistema. Le due macchine lavorano a velocità molto elevate, per questo, affinché il messaggio sia leggibile tramite i led, è necessario diminuire la loro frequenza di accensione aumentando i bit di `counter` assegnati nel modulo `Sender`.
È fondamentale osservare che nel caso generico in cui le board messe in comunicazione sono diverse, è necessario rallentare la frequenza di clock affinché sia compatibile con entrambe, questo perché la board più lenta deve comunque fare tutte le operazioni dettate, senza perdere informazione.

4.2 Creazione di funzioni/moduli Verilog Receiver e Sender

Sulla falsa riga con cui è stata sviluppata la comunicazione tra i due FPGA, i due moduli Verilog `Receiver` e `Sender`, che poi verranno sfruttati come funzioni, hanno una funzionalità analoga a quanto descritto precedentemente, ma qui saranno sfruttati per far comunicare 2 componenti dello stesso FPGA che si occupano di compiti differenti. Questi due moduli possono anche essere usati nella comunicazione con più FPGA. Rispetto a quanto descritto nella sezione precedente 4.1, i due moduli non usano più `bond_tx` e `parity` quindi non è più presente un controllo del messaggio ma, in un primo approccio, questo viene inviato direttamente.

4.2.1 Receiver function

L'obiettivo della funzione è quello di ricevere il messaggio diviso in singoli bit e, una volta completata la ricezione, passare il messaggio alla macchina. Utilizza come variabili:

- *clk*: ritmo con cui lavora la funzione
- *busy*: comunica che un messaggio è in ricevimento
- *valid*: comunica che la ricezione del messaggio è avvenuta correttamente ed è stata salvata in memoria
- *error*: comunica che la funzione ha avuto qualche errore
- *delay*: valore che rallenta il ritmo con cui la comunicazione avviene
- *message*: spazio in memoria in cui viene salvato il messaggio
- *clk wire*: filo attraverso il quale viene inviato la frequenza con cui si riceve il messaggio
- *text wire*: filo attraverso il quale viene trasmesso il messaggio
- *speed max*: velocità massima del clock

La funzione passa a differenti stati attraverso la variabile `cases` e per ogni valore di questa, agisce in modo differente: Nello stato di *idle*, la funzione aspetta finché non ho un segnale dal `clk_wire`, dopodiché salva in memoria la frequenza a cui dovrà lavorare, comunica alla board che sta per arrivare un messaggio e passa al prossimo stato.

```
            idle: begin
                if(clk_wire!=0) begin
                    speed<=delay;
                    cases<=wait_delay;
                    j<=delay;
                    busy<=1;
                    valid<=0;
                    k<=speed_max;
                    error<=0;
                end
            end
```

Nello stato *wait delay* la board avvia contemporaneamente 2 contatori: `j` parte dal valore di `speed` e va a 0, questo determina il tempo che il componente deve aspettare prima di iniziare a ricevere il messaggio, mentre `k` tiene conto del tempo massimo di attesa prima di segnalare un errore.
Questo viene fatto per evitare eventi transienti che non descrivono realmente un messaggio in arrivo.

```
        wait_delay: begin
                if(k==0)begin
                    error<=1;
                    cases<=idle;
                end
                else begin
                    k<=k-1;
                end
                if(j!=0)begin
                    if(clk_wire==0) begin
                        j<=speed;
                        end
                    else begin
                        j<=j-1;
                        end
                    end
                else begin
                    cases<=read_bit;
                end
            end
```

```
        read_bit: begin
               if(i==8) begin
                   cases<=resetting;
                   text[i]<=text_wire;
               end
               else begin
                   i<=i+1;
                   cases<=wait_delay;
                   text[i]<=text_wire;
               end
           end
```

In *read bit*, il messaggio inviato attraverso `text_wire` viene letto e viene salvato nel i-esimo bit di `text` per poi ritornare nello stato *wait delay* per l'arrivo dell'altro bit. Una volta che il messaggio viene completamente ricevuto, la funzione passa allo stato *resetting* in cui tutte le variabili vengono resettate, il segnale di `valid` viene inviato alla board, il messaggio ricevuto viene salvato nella memoria della board e la funzione passa infine allo stato *idle* in attesa del prossimo messaggio.

```
           resetting: begin
               busy<=0;
               valid<=1;
               cases<=idle;
               message<=text;
               i<=0;
           end
```

Nel caso di errore nel processo di comunicazione tra le due board, ogni volta che un segnale di `error` viene inviato, alla board, il clock di lavoro viene rallentato aumentando il delay in differenti modi (e.g. logaritmico, esponenziale, ecc...) per poi ritornare allo stato *idle*.

4.2.2 Sender function

L'obiettivo della funzione è quello di prendere il messaggio dalla macchina, dividerlo in bit singoli e inviarli attraverso un canale di comunicazione al ritmo deciso dalla board.
Le variabili su cui si basa il funzionamento di questo modulo sono:

- *clk*: ritmo con cui lavora la funzione
- *valid*: segnale che avvia l'invio del messaggio
- *message*: messaggio salvato in memoria
- *busy*: segnale inviato alla macchina nel caso in cui il componente stia inviando il messaggio
- *clk wire*: filo attraverso il quale viene inviato il ritmo di invio del messaggio
- *text wire*: filo attraverso il quale viene trasmesso il messaggio
- *speed clock*: velocità con cui manda il messaggio

Come in precedenza, il valore assegnato alla variabile `cases` definisce il comportamento della funzione. Nello stato *idle*, la funzione aspetta la conferma dalla macchina per inviare il messaggio per poi passare allo stato *init*.

```
        idle: begin
            if(valid) begin
                cases<=init;
            end
        end
        init: begin
            text<=message;
            speed<=speed_clock;
            busy<=1;
            cases<=transfer_up;
            j<=speed_clock;
        end

```

Nello stato di *init*, la funzione memorizza il messaggio, la velocita di clock, comunica alla macchina che sta inviando il messaggio e passa allo stato di *transfer up*. In questo stato, `clk_wire` viene messo ad 1 e viene inviato il primo bit del messaggio, passando allo stato di *transfer down*.

```
        transfer_up: begin
            clk_wire<=1;
            text_wire<=message[i];
            if(j==0) begin
                j<=speed;
                i<=i+1;
                cases<=transfer_down;
             end
            else begin
                j<=j-1;
            end
        end

```

In *transfer down* viene spento il segnale di `clk_wire` passando allo stato di *transfer up* oppure se il messaggio è stato completamente inviato passa allo stato finale.

```
        transfer_down: begin
        clk_wire<=0;
            if(j==0) begin
             j<=speed;
                 if(i==9) begin
                         cases<=resetting;
                 end
                 else begin
                         cases<=transfer_up;
                     end
             end
             else begin
                     j<=j-1;
                 end
    end

```

L'ultimo stato, nominato *resetting*, manda a zero tutte le variabili e comunica alla macchina che non è più occupato ritornando, al clock successivo, nello stato *idle* in attesa di un nuovo messaggio da inviare. Anche qui, come le due funzioni nella precedente sottosezione, comunicano tra di loro mandandosi un messaggio composto da 8 bit.

Per visualizzare il funzionamento delle macchine, alla board corrispondente il modulo `Receiver`, sono stati associati i primi otto led hai bit del messaggio. Le due funzioni, nella loro semplicità, possono essere richiamate più volte in un sistema di più FPGA. Facendo così un FPGA potrà fare da ponte di comunicazione tra altre due senza occupare altri dispositivi I/O.

5 FPGA e ricerca scientifica

Negli ultimi anni si fa un uso sempre più estensivo degli FPGA in vari campi della fisica e non solo, in quanto permette agevolmente di distinguere efficientemente eventi di interesse da quelli di fondo durante la presa dati, ma fondamentale è anche l'aumento della velocità di calcolo per poter eseguire algoritmi sempre più complessi e che elaborano una quantità di dati sempre maggiore.
Nei maggiori centri di ricerca come CERN, Ego e NSSTC, tra i più famosi, è sempre più importante l'avere dati precisi e in numero maggiore, al fine di avere una migliore valenza statistica.
Tra i diversi strumenti e metodi usati per aumentare la precisione durante la fase di acquisizione dei dati, da anni gli FPGA sono risultati fondamentali.

5.1 LHCb

Nel 2018 l'esperimento LHCb è stato aggiornato ad un sistema a lettura "triggerless" ad un rate di collisione di 40 MHz. La farm di computer a cui arrivano in prima battuta i dati è stata portata ad un livello superiore passando da un flusso di 500Gb/s fino a 40Tb/s di dati, questo è fondamentale poiché viene usato come sistema di pre-analisi dei dati. Come riporta l'articolo [2], il gruppo ha studiato diversi sistemi per affrontare la mole di dati partendo dall'algoritmo di identificazione delle particelle del Ring Imaging Cherenkov (RICH) detector. La versatilità degli FPGA fornisce la possibilità di usarli sia per aumentare potenza di calcolo sia per filtrare gli eventi. Il sistema di trigger era composto da 3 livelli : il primo in hardware, costituito da FPGA, e gli altri 2 in software. Il prototipo di acceleratore per la farm di dati è un doppio server Intel Xeon-FPGA: Il primo socket contiene una CPU Intel Xeon E5-2680 v2 mentre il secondo ospita un Stratix V GX A7 FPGA.

La connessione tra CPU e FPGA avviene a banda larga e bassa latenza con un bus seriale (QuickPath Interconnect) QPI e una memoria cache condivisa. La CPU scrive il dato in memoria fornendo il relativo puntatore al FPGA il quale può prendere l'indirizzo e modificare il dato. Per verificare le funzionalità e le prestazioni del sistema sono stati fatti diversi test standard come il ordinamento (sorting) di dati oppure il calcolo dei Frattali di Mandelbrot passando poi con il test per la ricostruzione della traccia con i dati forniti dal RICH. Quando una CPU deve ordinare un array di n elementi il tempo impiegato scala come $n * log(n)$, ma con un buon sistema di FPGA si può arrivare ad avere un tempo di calcolo dipendente solo dalla frequenza del clock in buona parte dei test effettuati. Il sistema di sorting con FPGA inizia a perdere efficenza a grandi n perchè il tempo di calcolo del FPGA scala come n^2. Come si può notare in figura la velocità con cui i due sistemi (CPU e CPU+FPGA) riescono ad ordinare gli array dipende dal numero e dalla lunghezza degli array stessi.[5.1]

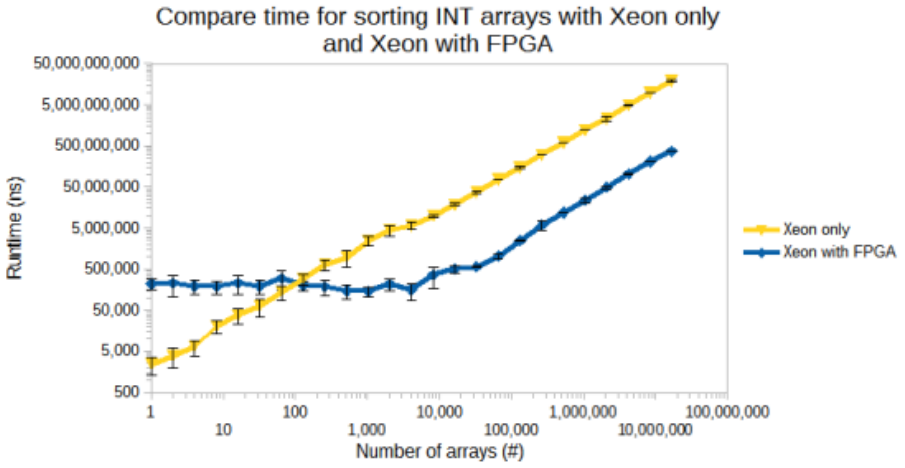

Figure 5.1: Confronto del tempo speso per ordinare array interi tra CPU e CPU+FPGA

Per un numero di array basso la sola CPU ha un netto vantaggio su CPU+FPGA ma all'aumentare di questo, il runtime del sistema con FPGA rimane più basso di quello della sola CPU. Precedentemente la parte software del trigger del RICH era troppo lenta per essere attivata in ogni collisione protone-protone. Questo ha portato allo sviluppo di un algoritmo implementato su un Xeon-FPGA per la ricostruzione dell'angolo Cherenkov. Tutte le cariche che viaggiano ad una velocità maggiore della luce nel mezzo emettono una radiazione Cherenkov distribuita a cono con angolo di apertura che dipende dalla velocità. I test ha mostrato che solo il 50 percento dei registri della Stratix V sono utilizzati dall'algoritmo. La pipeline lavora a 200MHz e riesce a calcolare ogni singolo fotone entro 5ns.

Come mostrato in figura [5.2] per un numero ridotto di fotoni la velocità di calcolo della sola CPU è migliore, questo è dovuto alla maggiore latenza del sistema CPU+FPGA.

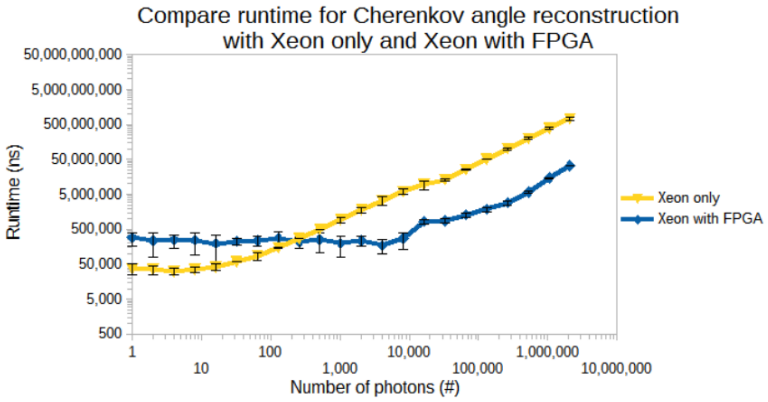

Figure 5.2: Confronto del tempo speso per ricostruire l'angolo di Cherenkov tra CPU e CPU+FPGA

Come per il test di sorting, il tempo di calcolo sui dati del detector mostrato in figura [5.2], è caratterizzato da un iniziale vantaggio della CPU che poi viene perso all'aumentare del numero di fotoni. Per un numero elevato di fotoni il confronto tra i due tempi di calcolo mostra un guadagno che va da un fattore 20 ad un fattore 35 a favore del sistema CPU+FPGA. Inoltre a livello teorico, aumentando la larghezza di banda, il vantaggio potrebbe arrivare fino ad un fattore 64.

5.2 Simulazione interazione di N-corpi

In astrofisica, già dal 2008, si cercava un modo per simulare il moto di N-corpi interagenti gravitazionalmente utilizzando diverse approssimazioni. Un gruppo di ricerca dell'università di Heidelberg, [12] è riuscito a creare un sistema nominato MPRACE-1 il quale utilizzava algoritmi basati sull'approssimazione "smoothed particle hydrodinamic" (SPH) per lo studio di modelli di evoluzione galattica o formazione stellare. In questo studio, viene presentato l'upgrade MPRACE-2 il quale implementa l'utilizzo di GPU insieme al vecchio sistema di FPGA. Gli FPGA vengono usati per implementare diversi algoritmi per accelerare il calcolo associato alla simulazione, sfruttandoli come coprocessori e associandogli una o più pipelines e una memoria esterna. Il sistema lavora a 62.5MHz ed è capace di memorizzare fino a 256 mila particelle. Per il nuovo cluster MPRACE-2 si è pensato di usare GPU NVIDIA cosi da poter utilizzare le librerie CUDA e gli stessi algoritmi di comunicazione per gli FPGA.

Dal confronto tra il tempo di calcolo dell'algoritmo SPH con vari sistemi in figura [5.3], il sistema con FPGA è fino a 6-11 volte più veloce della CPU mentre il sistema con GPU fino a 17 volte più veloce.

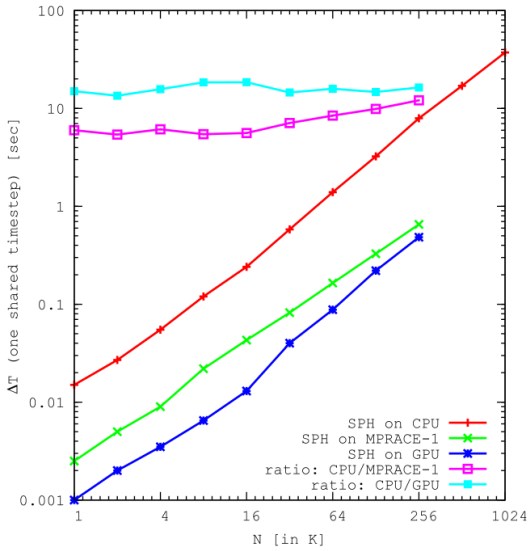

Figure 5.3: Tempo e accelerazione di differenti sistemi

5.3 Virgo

Una soluzione innovativa per diminuire costi computazionali degli algoritmi utilizzati attualmente, è l'implementazione di algoritmi di machine learning (ML) negli FPGA. Un altro campo di utilizzo è la rivelazione di onde gravitazionali. Infatti con l'aumento delle capacità dei detector e lo sviluppo di interferometri di nuova generazione come Einstein Telescope (ET) [6] o Cosmic Explorer [5] saranno sempre maggiori le rivelazioni di onde gravitazionali associate a sistemi binari di stelle di neutroni. Maggior sensibilità implica una maggiore complessità computazionale legata all'aumento della durata dei segnali che passa da alcuni secondi, con gli attuali detector, ad ore con i detector di nuova generazione. Uno degli obiettivi dei futuri interferometri è riuscire a rivelare segnali di onde gravitazionali già durante la fase di spiraleggiamento per permettere ai telescopi di osservare l'evento per estrarre informazioni dalla controparte elettromagnetica. Tuttavia questi segnali hanno frequenze basse e sono molto deboli per questo attualmente sono oscurati dai rumori di fondo. Gli algoritmi di ML possono essere computati GPU o CPU ma il rapporto tra l'efficienza e il costo necessario per operare i sistemi è sempre minore con il passare del tempo. Gli FPGA, usati in questo lavoro, mostrano un promettente rapporto efficienza-costo e soprattutto una vita media più lunga rispetto a CPU e GPU.

Per allenare gli algoritmi di ML sono stati usati i dati della collaborazione LIGO-Virgo-KAGRA di tutti i periodi di osservazione. L'obiettivo di questi algoritmi non è solo estrarre segnali di onde gravitazionali dal rumore di fondo il prima possibile, ma anche di minimizzare il numero di falsi positivi. Nel lavoro presentato [1] da Ana Martins et al. viene messo a confronto l'attuale algoritmo `FindCNN` con quello creato dal gruppo `GWaveNet`. Come mostra la figura [5.4] le prestazioni di `GWaveNet` sono migliori rispetto a `FindCNN`: necessita di meno parametri, minor potenza computazionale ma soprattutto mostra un maggiore numero di "floating point operation" (FLOPs) con una riduzione degli errori in fase di training del modello. (FLOPs indica il numero di operazioni in virgola mobile eseguite in un secondo dal processore)

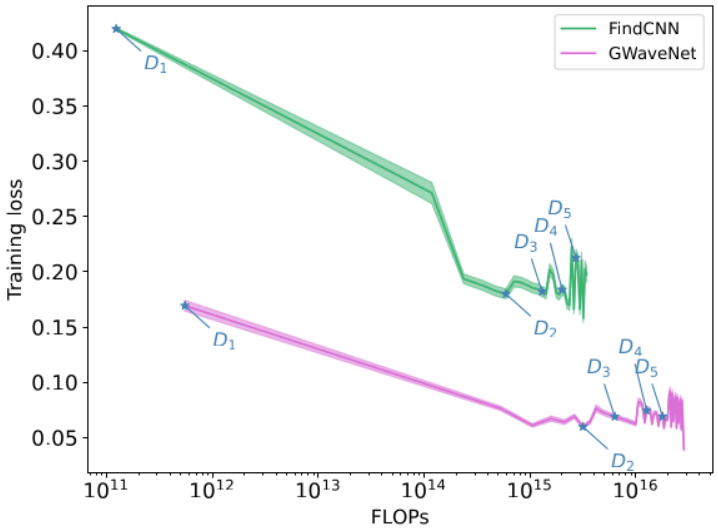

Figure 5.4: rapporto tra "Training loss" e FLOPs fatti dai due algoritmi

L'algoritmo `FindCNN` è stato fatto girare su GPU, CPU e FPGA, precisamente AMD Kria KV260 Vision AI Starter Kit con il software Vitis AI. L'algoritmo `GWaveNet` non è stato fatto girare sul FPGA questo per la scarsa memoria del dispositivo ma, come il gruppo sottolinea, l'FPGA usata è di basso livello mentre sono presenti nel mercato FPGA con maggiore potenza e lo studio viene fatto per creare nuovi algoritmi sempre più efficaci. Osservando la tabella dei risultati in figura [5.5], si nota che l'algoritmo `GWaveNet` non è stato fatto girare sul FPGA a causa della scarsa memoria del dispositivo, ma l'articolo sottolinea che altri FPGA presenti nel mercato potrebbero farlo. Inoltre gli algoritmi non sono ottimizzati per gli FPGA infatti il tempo di calcolo per eseguire `FindCNN2D` e `FindCNN2DModified` risulta maggiore rispetto a CPU e GPU. L'articolo sottolinea inoltre come l'uso di FPGA sia l'opzione più sostenibile in termini di costi e di ambiente.

Model	Time (ms)			Energy consumption (mJ)			Economic cost ($/year)		
	CPU	GPU	FPGA	CPU	GPU	FPGA	CPU	GPU	FPGA
FindCNN	27.53 ± 3.69	21.75 ± 1.89	-	107.24 ± 4.26	617.12 ± 37.11	-	28.83	210.01	-
FindCNN2D	29.58 ± 0.10	20.90 ± 1.56	84.54 ± 0.10	177.93 ± 1.60	606.74 ± 32.06	468.59 ± 2.40	44.52	214.87	41.03
FindCNN-2DModified	29.54 ± 0.11	24.87 ± 1.65	84.28 ± 0.10	178.35 ± 1.34	694.59 ± 28.46	471.51 ± 2.96	44.69	206.72	41.41
GWaveNet	154.75 ± 0.38	33.60 ± 2.76	-	583.66 ± 5.56	983.83 ± 60.52	-	27.92	216.72	-
GWaveNet2D	85.28 ± 2.87	29.41 ± 1.77	-	568.92 ± 4.95	858.11 ± 36.83	-	49.38	215.96	-
GWaveNet-2DModified	65.54 ± 3.67	26.02 ± 3.25	-	240.76 ± 5.85	752.49 ± 57.90	-	27.19	214.05	-

Figure 5.5: tempo medio (ms) e consumo energetico (mJ) per predire un singolo sample e il costo per usare ogni modello per anno

5.4 Medicina

Con lo sviluppo della medicina genomica che entra a far parte dei trattamenti clinici base, ci si trova di fronte ad una sfida di computazione sempre più elevata anche in ambito medico. In particolare, in oncologia è fondamentale caratterizzare il tumore al fine di attuare il miglior trattamento personalizzato per il paziente. Per fare questo è fondamentale il sequenziamento dei geni delle cellule tumorali che richiede grandi potenze di calcolo. Come riporta l'articolo *"Precision Medicine and FPGA Technology"* [8] l'uso di questi strumenti ha portato all'aumento del numero di mutazioni catalogate grazie al sequenziamento di più di 100000 genomi di tumori. Questo sequenziamento porta ad una mole di dati a cui i sistemi informatici devono fare fronte, infatti se a tutte le persone americane affette da tumore fosse sequenziato il genoma ogni 2 settimane, non avremmo Terabytes ma Exabytes di dati, impossibili da sostenere.[5.6]

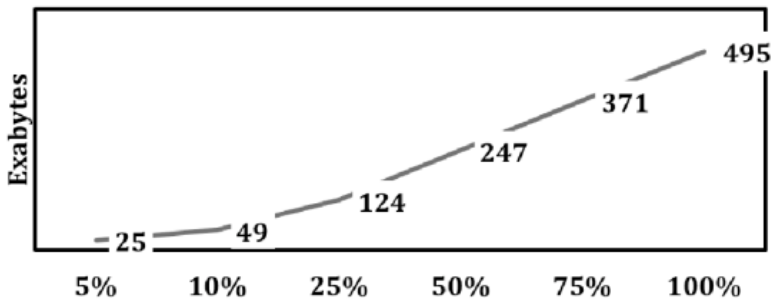

Figure 5.6: Percentuale di persone affette di tumore in USA ai quali viene sequenziato il genoma ogni 2 settimane

Con la ricerca e lo sviluppo di questi dispositivi, insieme alla nascita di nuovi algoritmi di intelligenza artificiale, si punta ad un aumento della velocità di calcolo e della precisione degli algoritmi bioinformatici. Tutto questo risulterà essere un fondamentale strumento che aiuterà i medici a fornire cure accurate per i pazienti in tempi sempre più brevi. Un eccellente esempio di questo sistema è fornito dalla piattaforma Illumina DRAGEN BIO-IT, basata su un processore integrato con una

board FPGA con una serie di algoritmi già inseriti, che permette una veloce identificazione di genomi, sequenze RNA e tumori. La comunicazione tra server online e la board permette, ad esempio, il sequenziamento di tutto il genoma umano in soli 20 minuti invece di molte ore. [9]

6 Conclusion

Dall'importanza che i Field Programmable Gate Array (FPGA) hanno in ambito scientifico, nasce l'interesse verso lo sviluppo di nuovi sistemi di calcolo basati su di essi. In particolare il progetto Bondmachine ha come obiettivo quello di sviluppare un framework capace di creare strutture dove sia hardware che software sono costruiti in modo da sfruttare a pieno le capacità del sistema anche in termini di efficienza energetica.
L'idea è quella di creare una macchina che possa dinamicamente dividere il sistema di FPGA in più processori interconnessi tra loro, così da ottimizzare i processi e i lavori svolti dalla macchina stessa. All'interno di questo progetto è fondamentale la comunicazione tra diversi FPGA o tra diversi componenti di uno stesso FPGA. In questo ambito si colloca questo progetto di tesi, creando i moduli Verilog necessari all'invio e alla ricezione di informazioni.
Una prima parte è legata alla creazione di un rete di comunicazione tra due FPGA: i bitstream generati dai due moduli `Sender` e `Receiver`, sottoforma di firmware, sono stati caricati su due evaluation board Basys 3 messe in comunicazione attraverso cavi collegati ai pin delle due board.
La seconda parte del lavoro è stata volta a sviluppare due funzioni in Verilog il cui compito è lo stesso dei moduli sviluppati in precedenza, ma che permettono di far comunicare tra loro diverse componenti di una stessa board. Le stesse funzioni `Sender` e `Receiver`, sono scritte in maniera da poter essere utilizzate da qualsiasi FPGA per gestire la comunicazione tra due componenti dello stesso FPGA o tra FPGA diversi.

Bibliografia

[1] Q. Meijer M. van der Sluys C. Van Den Broeck G. Baltus A. Martins, M. Lopez and S. Caudill. *"Improving Early Detection of Gravitational Waves from Binary Neutron Stars Using CNNs and FPGAs". Cornell University*, 2024.

[2] J. Machen C. Färber, R. Schwemmer and N. Neufeld. *"Particle Identification on an FPGA Accelerated Compute Platform for the LHCb Upgrade". Ieee transaction on nuclear science, Vol. 64, no. 7 July*, 2017.

[3] M. Danelutto. *"Note sull' utilizzo di Verilog per la prima parte del corso di Architettura degli Elaboratori"*. `http://didawiki.cli.di.unipi.it/lib/exe/fetch.php/informatica/ae/verilog2.pdf`.

[4] Digilent. Basys 3 reference manual. `https://digilent.com/reference/programmable-logic/basys-3/reference-manual`.

[5] D. Reitze et al. *"Cosmic Explorer: The U.S. Contribution to Gravitational-Wave Astronomy beyond LIGO ". Bull. Am. Astron. Soc., vol. 51, no. 7*, 2019.

[6] M. Punturo et al. *"The Einstein Telescope: A third-generation grav- itational wave observatory". Class. Quant. Grav., vol. 27*, 2010.

[7] I. Goodfellow, Y. Bengio, and A. Courville. *"Deep learning". MIT press*, 2016.

[8] B. Hill, J. Smith, G. Srinivasa, K. Sonmez, A. Sirasao, A. Gupta, and M. Mukherjee. *"Precision Medicine and FPGA Technology ". IEEE Explorer*.

[9] Illumina. *Illumina DRAGEN Bio-IT Platform v3.5*. `https://support.illumina.com/content/dam/illumina-support/documents/documentation/software_documentation/dragen-bio-it/dragen-bio-it-platform-v3.5-user-guide-1000000111887-00.pdf`.

[10] A. Kluge. *"FPGA-application"*. `https://indico.ictp.it/event/a11204/session/27/contribution/17/material/0/0.pdf`.

[11] Mirko Mariotti, Daniel Magalotti, Daniele Spiga, and Loriano Storchi. The bondmachine, a moldable computer architecture. *Parallel Computing*, 109:102873, 2022. ISSN 0167-8191. doi: https://doi.org/10.1016/j.parco.2021.102873. URL https://www.sciencedirect.com/science/article/pii/S0167819121001150.

[12] G. Marcus A. Kugel G. Lienhart I. Berentzen R. Männer R. Klessen R. Banerjee R. Spurzem, P. Berczik. *"Accelerating astrophysical particle simulations with programmable hardware (FPGA and GPU)"*. 12 May 2009.

[13] G. Surace. *"Sviluppo di una libreria di interfaccia verso acceleratori basati su FPGA"*. https://github.com/BondMachineHQ/TesiSurace.git.

[14] S. Trimberger. *"Three Ages of FPGAs: A Retrospective on the First Thirty Years of FPGA Technology"*. *Fellow IEEE*, 2015.

[15] B. Vaughn. *"FPGA Architecture for the Challenge"*. https://www.eecg.toronto.edu/~vaughn/challenge/fpga_arch.html.