POLITECNICO DI TORINO

Laurea Magistrale in INGEGNERIA INFORMATICA

Tesi Magistrale

Studio e Sviluppo di un Algoritmo di Routing per FPGA Radiation-Hardened ottimizzato per GPGPU

Relatori

Prof. Luca STERPONE

Dott. Corrado DE SIO

Ing. Andrea PORTALURI

Candidato

Andrea SARACINO

Luglio 2024

Sommario

I Field Programmable Gate Arrays (FPGAs) sono circuiti integrati riprogrammabili un numero indefinito di volte dopo la produzione. Questa caratteristica permette di adattare il comportamento del dispositivo per applicazioni specifiche e di rispondere rapidamente a nuove esigenze.
All'interno del flusso di implementazione di un nuovo design su FPGA, il routing è una fase cruciale che può determinare sia la fattibilità di un'implementazione su logica programmabile, sia il rispetto dei vincoli temporali dei segnali interni dell'FPGA. Il processo di routing riceve come input le specifiche delle connessioni tra i vari componenti interni del circuito, e ha lo scopo di determinare i percorsi ottimali per collegare tali componenti, cercando di garantire l'efficienza e l'affidabilità del sistema. Questo processo può essere molto dispendioso in termini di tempo e risorse, rappresentando spesso un collo di bottiglia nella toolchain per FPGA.
In questa tesi presentiamo il primo router personalizzato progettato specificamente per le architetture FPGA di NanoXplore resistenti alle radiazioni. Le FPGA di riferimento utilizzate sono NG-Medium e NG-Ultra. Il router, accelerato tramite tecnologia GPU NVIDIA, rappresenta le sue risorse fisiche su un grafo di risorse di routing. Su tale grafo sono applicate due tecniche di parallelizzazione su GPU, ognuna delle quali opera su un diverso livello. La tecnica che opera a basso livello parallelizza l'esplorazione delle risorse per la costruzione del singolo percorso ottimale sul grafo di risorse di routing. La seconda tecnica, che opera a livello più alto, suddivide i percorsi da trovare in set indipendenti dal punto di vista delle risorse, e si occupa di generare diversi percorsi contemporaneamente in modo concorrente su GPU.
Infine si è analizzato il router sia dal punto di vista delle performance, sia della memoria consumata durante l'esecuzione. Riguardo le performance, è stato proposto un confronto tra tre varianti di router: una completamente sequenziale su CPU basata sull'algoritmo di ricerca in ampiezza, una che implementa solo la tecnica di parallelismo a basso livello su GPU, generando però i percorsi in modo sequenziale, e una che implementa entrambe le tecniche di parallelismo ad alto e basso livello. Il confronto ha rivelato un incremento di performance di circa 100 volte rispetto alla versione sequenziale, e di come la tecnica ad alto livello ne affini ulteriormente

le performance.
Per quanto riguarda il consumo di memoria, è emerso che esso aumenta linearmente rispetto al numero di percorsi generati contemporaneamente, dovuto all'allocazione di strutture dati necessarie per supportare più ricerche concorrenti sul grafo.

Ringraziamenti

Un sentito ringraziamento al prof. Luca Sterpone, ai corelatori Corrado De Sio e Andrea Portaluri, e all'intero laboratorio per avermi accompagnato durante la realizzazione di questa tesi. Ringrazio familiari, amici e chi mi ha sostenuto e supportato durante l'intero percorso di studi. Un grazie speciale a mio nonno Mario, a cui dedico la tesi.

Andrea Saracino

Indice

Elenco delle tabelle

Elenco delle figure

Capitolo 1

Introduzione

1.1 La Tesi

La crescente complessità dei progetti basati su Field Programmable Gate Array (FPGA) richiede soluzioni innovative per ottimizzare i processi di sviluppo. Uno dei passaggi più critici e dispendiosi in termini di tempo e risorse è il routing, ovvero la fase in cui vengono determinati i percorsi delle connessioni interne all'FPGA per garantire che tutte le interconnessioni siano realizzate nel rispetto dei vincoli temporali e di design. Questo passaggio spesso rappresenta un collo di bottiglia nella toolchain di sviluppo per FPGA.
La tesi si propone di affrontare questa sfida attraverso lo sviluppo di un algoritmo di routing parallelo, specificamente adattato ai chip europei resistenti alle radiazioni di NanoXplore. Per migliorare significativamente le prestazioni, l'algoritmo sfrutta la potenza di calcolo delle GPU NVIDIA, permettendo una parallelizzazione efficace e ottimizzata.

1.2 Attività svolte

Questa sezione fornirà una breve panoramica sulle attività svolte durante la tesi e che hanno portato al suo conseguimento. Saranno esposte sia le attività di studio sia le attività pratiche, quali la messa a punto degli algoritmi e l'analisi dei risultati. Si presenterà quindi nella sua totalità il percorso seguito.
Non avendo alcuna nozione pregressa sull'uso delle GPU, la tesi è iniziata con lo studio di tali dispositivi e di come poterli impiegare sul problema di interesse. A tale scopo, si sono inizialmente studiate nozioni di carattere generale sul funzionamento delle GPU, come architettura, potenzialità e limitazioni. Si è poi deciso di adottare dispositivi della famiglia NVIDIA e proseguire con lo studio di cuda programming, per una questione di disponibilità di GPU di tale azienda.

Successivamente, la tesi si è concentrata sull'approfondimento delle FPGA, con un'attenzione particolare alla toolchain. È stato studiato il processo di routing, ricercando lo stato dell'arte e focalizzandosi su tecniche e implementazioni che sfruttano la programmazione concorrente. Una volta acquisito un solido background in materia, è stato implementato un prototipo di router su GPU. Inizialmente, questo prototipo non utilizzava le informazioni architetturali specifiche delle FPGA di NanoXplore, ma veniva testato su grafi generati casualmente.
Infine, l'attenzione si è spostata sull'architettura delle FPGA NG-Ultra e NG-Medium, che condividono la stessa struttura di base, con NG-Medium caratterizzata da un numero più limitato di risorse. È stato fornito un database rappresentante le risorse fisiche di NG-Medium, sul quale è stato implementato l'algoritmo di routing precedentemente testato su piccoli grafi randomici.
Le funzionalità del router sono state poi ulteriormente sviluppate e ottimizzate utilizzando il profiler NVIDIA per migliorarne le prestazioni. È stata quindi condotta una valutazione delle performance ottenute e un'analisi del consumo di memoria RAM e VRAM. Questo processo di analisi ha permesso di valutare efficacia e criticità dell'algoritmo sviluppato e di identificare eventuali aree di miglioramento per futuri lavori.

1.3 Organizzazione

Questo documento è organizzato in modo tale da fornire inizialmente una conoscenza di base riguardante le FPGA e le GPU. Avere questo background è fondamentale per comprendere cosa sia il processo di routing che ci prefiggiamo di implementare e i motivi per cui sia cruciale velocizzare tale processo. Si procede poi con nozioni architetturali specifiche riguardanti le FPGA target di NanoXplore, e con l'algoritmo di routing sviluppato su tali architetture. Si conclude infine con un analisi delle performance e con considerazioni su possibili sviluppi futuri. I capitoli trattano i seguenti temi:
Il primo capitolo è introduttivo ai temi trattati e alle attività svolte durante la tesi.
Il secondo capitolo fornisce una panoramica su cosa sia un FPGA, includendo un approfondimento sulla sua architettura e sulla toolchain utilizzata per la loro programmazione.
Il terzo capitolo introduce invece le GPU, descrivendone l'architettura e spiegando come possano essere utilizzate per il routing delle FPGA, includendo le limitazioni e lo stato dell'arte in questo campo.
Nel quarto capitolo si discute dell'architettura delle FPGA NG-Ultra e NG-Medium, includendo la descrizione di come le risorse fisiche di tali architetture siano rappresentate in un database a nostra disposizione.
Il quinto capitolo tratta la scrittura di un algoritmo di placement basilare e i motivi

di tale implementazione, mentre il sesto capitolo, fulcro della tesi, si focalizza sulla scrittura di un algoritmo di routing personalizzato basato su GPU. Del router sono illustrate nel dettaglio le strutture dati e le tecniche di programmazione concorrente usate.
Infine, il settimo capitolo presenta i risultati ottenuti in termini di performance mostrando un confronto con un approccio sequenziale, e analizzando inoltre il consumo di memoria all'aumentare del parallelismo. La tesi si conclude con l'ottavo capitolo che propone una riflessione su possibili sviluppi futuri e possibili migliorie apportabili.

Capitolo 2

Field Programmable Gate Arrays

2.1 Cos'è un FPGA

Gli array di porte programmabili sul campo o FPGA sono dispositivi a semiconduttore con la funzionalità di poter essere riprogrammati dall'utente dopo la produzione per implementare circuiti e applicazioni con logiche custom. Fanno parte della categoria dei Programmable Logic Devices (PLDs). In altre parole, le FPGA sono circuiti integrati (IC) personalizzabili un numero indefinito di volte dopo la produzione. Ciò conferisce a questi dispositivi un'elevata flessibilità, rendendoli adatti ad un'ampia varietà di applicazioni. Il termine "Field Programmable" si riferisce alla capacità di questi chip di poter essere riprogrammati dall'utente dopo la produzione per svolgere specifiche funzioni logiche, mentre il termine "Gate Arrays" si riferisce invece alla struttura dell'architettura hardware, formata da una grande quantità di porte logiche (gate) distribuite su un array bidimensionale di CLB (configurable logic blocks) [1].
Le FPGA sono tipicamente utilizzate come parte di un sistema più complesso che include altri componenti hardware e software. Vengono spesso integrate ad esempio a sistemi con microcontrollori (MCU) o microprocessori (CPU), e può essere utilizzata per accelerare funzioni specifiche, mentre il microcontrollore o il processore esegue operazioni e logica di alto livello. Possono anche essere impiegate per l'interfacciamento con periferiche esterne, come sensori, memorie e display, per gestire protocolli di comunicazione complessi e fornire elaborazioni in tempo reale. Anche sistemi di tipo embedded possono far uso di FPGA. Un uso comune per i sistemi embedded è la gestione dell'elaborazione di alcuni segnali provenienti da sensori per prendere decisioni in tempo reale in base agli algoritmi programmati.
Nonostante non siano generalmente dispositivi standalone, ci sono alcune situazioni

Figura 2.1: FPGA Virtex™ UltraScale+™

in cui un FPGA può operare in modo relativamente autonomo. Alcune schede di sviluppo FPGA sono equipaggiate con tutti i componenti necessari per operare autonomamente, come alimentatori, clock generator, memorie RAM e flash, e interfacce I/O. Tali schede possono eseguire operazioni preprogrammate senza necessitare di un microcontrollore o di un processore esterno. In alcuni progetti specifici, un FPGA può essere configurata per eseguire un compito dedicato in modo indipendente. Tuttavia, nella maggior parte delle applicazioni pratiche, le FPGA sono utilizzate in combinazione con altri componenti hardware e software per formare sistemi complessi e multifunzionali.

2.2 Caratteristiche

A seconda delle specifiche esigenze del progetto, è importante valutare se un FPGA sia il dispositivo più adatto o meno per soddisfare tali esigenze. Le FPGA

presentano una serie di caratteristiche peculiari, tra cui:

- **Flessibilità e Riprogrammabilità:**
  Non esiste limite al numero di volte che è possibile riprogrammare un FPGA. Ciò le rende ideali per ambienti di sviluppo e prototipazione rapida. Questa flessibilità consente agli ingegneri di aggiornare sul campo le funzionalità del chip per correggere bug, ottimizzare le prestazioni o aggiungere nuove funzioni senza dover sostituire l'hardware.

- **Prestazioni Elevate rispetto un'implementazione software:**
  Rispetto a soluzioni software eseguite su CPU o GPU, le FPGA offrono una soluzione hardware a problemi di calcolo anche molto complessi. Ciò le rende, per molte applicazioni, vantaggiose dal punto di vista prestazionale.

- **Tempo di Commercializzazione Ridotto:**
  La possibilità di iterare rapidamente sui design grazie alla riprogrammabilità delle FPGA riduce il tempo necessario per portare un prodotto sul mercato. Questo è particolarmente utile nelle fasi iniziali dello sviluppo di nuovi dispositivi.

- **Personalizzazione Elevata:**
  Le FPGA possono essere configurate per eseguire operazioni molto specifiche che potrebbero non essere ottimizzate su CPU o GPU standard. Questa personalizzazione permette di adattare il dispositivo alle esatte necessità del progetto.

- **Supporto per Applicazioni Critiche:**
  Le FPGA sono utilizzate in applicazioni che richiedono elevata affidabilità e prestazioni in tempo reale, come nei settori aerospaziale, della difesa, delle telecomunicazioni e delle reti neurali.

- **Costo:**
  Le FPGA possono essere significativamente più costose rispetto ad altre soluzioni come i microcontrollori o le CPU, soprattutto per volumi elevati di produzione. Inoltre, rispetto ai dispositivi ASIC, presentano costi di produzione marcatamente più elevati per il singolo componente, nonostante un ASIC abbia un costo iniziale di progettazione nettamente più elevato. Ciò rende gli ASIC più economici se si intende mandare in produzione di massa il chip, mentre le FPGA risultano più economiche se si intende produrre pochi pezzi.

- **Consumo Energetico:**
  Le FPGA tendono a consumare più energia rispetto agli ASIC. Questo avviene perchè un ASIC ha un'architettura specializzata ad adempiere ad un certo compito, mentre le FPGA hanno un'architettura di risorse generiche

programmabili. Questo può essere un fattore limitante in applicazioni dove il consumo energetico è critico.

- **Prestazioni, Dimensioni e Integrabilità rispetto agli ASIC:**
  Anche riguardo le prestazioni e le dimensioni del chip, le FPGA risultano essere una soluzione meno ottimizzata rispetto alla progettazione ad hoc di un ASIC. Questo avviene, come per il consumo energetico, perché gli ASIC presentano un'architettura specializzata ed ottimizzata al task specifico. Perciò se l'interesse è rivolto in modo particolare alle prestazioni, alle dimensioni ridotte del chip, o al basso consumo energetico, gli ASIC forniscono un'alternativa migliore. Tuttavia la scelta ricade sulle FPGA se si è interessati ad un'alta flessibilità del chip o ad una produzione non di massa.

2.3 Architettura

Le FPGA seguono lo schema architetturale Logic Cell Array (LCA), ossia presentano un'architettura composta da una griglia di celle logiche interconnesse programmabili. Le celle logiche sono interconnesse tra loro per mezzo di una rete di connessioni, anch'esse programmabili. Questa rete di interconnessioni è essenziale per definire il flusso di segnali tra le diverse celle logiche, consentendo la realizzazione di circuiti complessi. I tre elementi principali di un FPGA sono:

- Blocchi logici configurabili (CLB)
- Interconnessioni programmabili
- Blocchi di Input/output (IOB)

I blocchi logici configurabili (CLB) sono i componenti fondamentali di un FPGA. Disposti in una configurazione a matrice, ognuno dei CLB contiene una piccola quantità di logica, tra cui LUT (Look-Up Table), flip-flop, multiplexer e logica aritmetica. In particolare, le LUT sono quei componenti che possono essere configurati per implementare una tabella di verità definita dall'utente, e che quindi permettono all'intera CLB di poter implementare una funzione logica digitale personalizzata. Una LUT, piuttosto che essere un circuito che calcola gli output a partire dagli input, è una memoria che fa una mappatura tra input e output. Per ogni combinazione di bit in input, la LUT restituisce l'output corrispondente salvato in memoria. Questo le consente di essere più efficiente nel fornire il segnale in uscita rispetto a una soluzione che calcoli tale segnale in tempo reale. I flip-flop sono invece componenti che permettono di memorizzare uno stato binario, consentendo al CLB di eseguire operazioni sequenziali che necessitano di uno stato.
Questi CLB, disposti fisicamente in una configurazione a matrice, si interconnettono

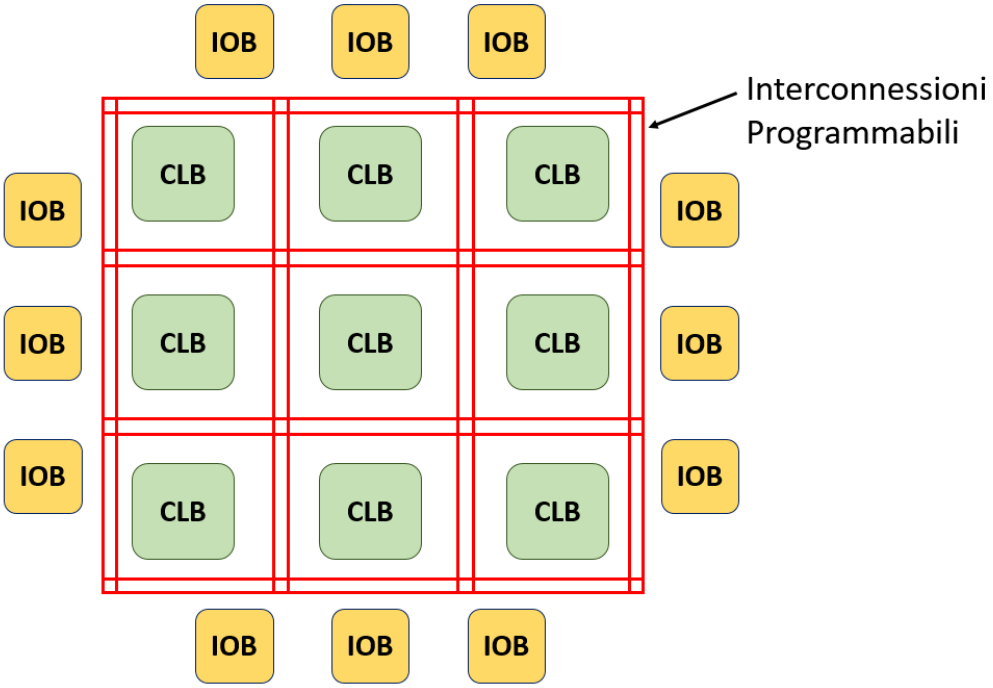

Figura 2.2: Architettura interna di una FPGA generica.

tra loro tramite un complesso sistema di connessioni, anch'esse programmabili. Questo sistema offre ai progettisti la flessibilità di configurare i CLB per creare tabelle di verità personalizzate, i cui output possono essere reindirizzati a piacere in input ad altri CLB. In questo modo, i progettisti possono creare una rete di CLB in grado di implementare design di circuiti complessi secondo le loro specifiche esigenze. Le interconnessioni non solo possono connettere CLB tra loro, ma interconnettono anche le CLB con i blocchi di I/O, permettendo la comunicazione con i dispositivi esterni.
Gli FPGA dispongono di numerosi blocchi I/O (IOB) che permettono al dispositivo di comunicare con altri componenti del sistema, come microprocessori, memorie, e periferiche esterne. Gli IOB gestiscono il trasferimento di segnali verso l'interno e l'esterno del circuito integrato e sono in grado di supportare vari standard di comunicazione. Le risorse di I/O possono anche tradurre segnali analogici o digitali in valori digitali compatibili con l'FPGA, consentendo internamente l'elaborazione di questi segnali.
Molte FPGA includono blocchi di memoria RAM configurabile, che possono essere utilizzati per l'archiviazione temporanea dei dati durante l'elaborazione. Quando la RAM è disponibile sotto forma di blocco logico, prende il nome di Block RAM (BRAM), ma esistono anche FPGA che offrono soluzioni con RAM disposta

internamente in modo distribuito, la Distributed RAM, utile tipicamente per applicazioni che necessitano di piccole memorie molto vicine ai blocchi di logica.
Infine, alcune FPGA possono contare su blocchi specializzati in aggiunta ai blocchi standard di cui abbiamo appena discusso. Degli esempi di blocchi specializzati possono essere moltiplicatori hardware, blocchi DSP (Digital Signal Processing), microprocessori, e transceiver ad alta velocità per migliorare le prestazioni in applicazioni specifiche. Sempre più spesso è possibile trovare tali blocchi tra le risorse disponibili a causa della frequenza con la quale queste funzioni sono richieste [2].

2.4 Computer-Aided Design tools per FPGA

Come menzionato in precedenza, le FPGA sono circuiti integrati programmabili. Per programmare un FPGA ed ottenere l'implementazione del circuito desiderato, viene utilizzato un insieme di strumenti software specifici noti come strumenti di progettazione assistita da computer (Computer-Aided Design tools, CAD tools). I tool CAD sono fondamentali per implementare un circuito custom su FPGA, e la loro qualità influisce fortemente sul risultato finale. Questi strumenti determinano non solo la funzionalità e la correttezza del circuito, ma impattano anche sulla velocità operativa, sul consumo energetico e sull'efficienza complessiva del design. Inoltre, l'uso efficace dei tool CAD può migliorare la densità del circuito, ridurre il tempo di sviluppo e ottimizzare l'utilizzo delle risorse dell'FPGA.
Il processo imbastito dagli strumenti CAD per FPGA prende in ingresso la descrizione del circuito digitale da implementare, insieme alle sue caratteristiche e ai suoi vincoli. Questo input viene poi tradotto in un formato che rappresenta la mappatura delle risorse logiche definite nell'input sulle risorse fisiche disponibili sull'FPGA target. L'output risultante, che rappresenta tale mappatura, è un file di configurazione in formato binario chiamato bitstream, il quale codifica in binario la configurazione di ciascuna delle risorse dell'FPGA.
Data la complessità di questo processo di mappatura delle risorse, esso viene suddiviso in diversi passaggi distinti all'interno del flusso di lavoro degli strumenti CAD [3]. Questi passaggi comprendono fasi di sintesi, ottimizzazioni, piazzamento e instradamento, ciascuna delle quali svolge un ruolo cruciale nel garantire che il design finale soddisfi i requisiti funzionali e di prestazione specificati inizialmente. Ogni fase del flusso di lavoro è progettata per affrontare specifici aspetti della traduzione e ottimizzazione del design, assicurando un'implementazione efficiente ed efficace sulla piattaforma FPGA scelta.
Vediamo ora in modo più dettagliato le fasi del processo di programmazione di un FPGA:

- **Design Entry**

- **Design Synthesis**
  - Sintesi RTL
  - Sintesi Logica
- **Design Implementation**
  - Mapping
  - Placement
  - Routing
- **Bitstream Generation**

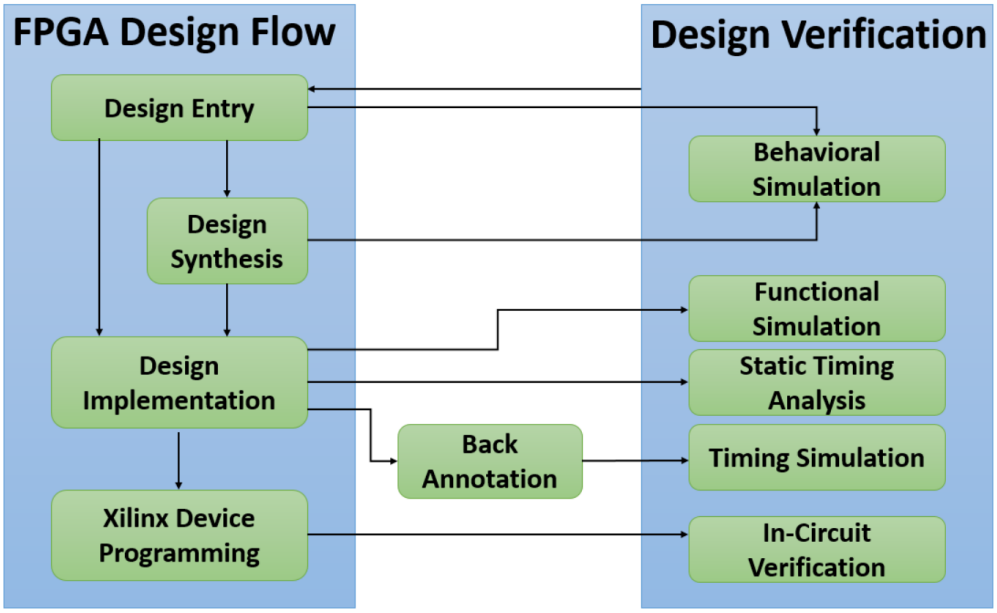

Figura 2.3: Esempio degli steps di design flow per Xilinx ISE: design entry, design synthesis, design implementation e Xilinx device programming. La design verification avviene durante i vari steps del design flow [4].

Prima di descrivere questi steps nel dettaglio, è importante precisare che per diversi produttori questa catalogazione subisce leggere variazioni. Alcuni step potrebbero essere accorpati assieme o separati in più parti. Nonostante le variazioni, il flow generale delle trasformazioni ai dati rimane per lo più questo.

2.4.1 Design Entry

La fase di Design Entry è una fase che comprende diversi steps. Tuttavia i progettisti si occupano principalmente di descrivere l'hardware del circuito da implementare. Ciò può essere fatto con due approcci differenti.
Un approccio è quello di descrivere l'hardware tramite appositi linguaggi di descrizione hardware (HDL). VHDL o Verilog sono i linguaggi HDL più comunemente utilizzati. Usare linguaggi HDL è preferibile per un approccio algoritmico e per design complessi, scalabili e facilmente verificabili tramite simulazioni.
L'approccio alternativo è invece quello di usare strumenti di progettazione schematica. Questi strumenti permettono ai progettisti di disegnare graficamente il circuito, connettendo simboli che rappresentano componenti logici. E' un approccio di solito utilizzato e adatto per design semplici, dove la visualizzazione delle connessioni fisiche è cruciale, o per design più intuitivi da gestire graficamente. E' anche possibile approcciare la fase di descrizione dell'hardware combinando gli approcci, ed usare la progettazione schematica come complemento alla descrizione HDL per visualizzare ad esempio parti critiche del circuito.

2.4.2 Design Synthesis

La fase di Design Synthesis del processo di progettazione FPGA è la fase in cui la descrizione ad alto livello di un circuito digitale, ottenuta dalla precedente fase di Design Entry, viene trasformata in una rappresentazione a basso livello chiamata netlist. La netlist rappresenta il circuito come una rete logica di elementi interconnessi pronti ad essere implementati sull'FPGA.
Questo processo di Design Synthesis combina sia la sintesi RTL (Register-Transfer Level) sia la sintesi logica, ed è responsabile di assicurare che il design funzioni correttamente sull'hardware target, rispettando i vincoli di prestazione e consumo energetico. In particolare, la sintesi RTL si occupa di tradurre la descrizione hardware ad alto livello in un formato intermedio, la netlist RTL, che rappresenta l'hardware sotto forma di blocchi logici di basso livello, come ad esempio moltiplicatori, sommatori, multiplexers e macchine a stato [3]. La sintesi logica, invece, è il processo che trasforma la netlist RTL in una netlist ottimizzata per essere poi mappata su blocchi logici specifici alla tecnologia dell'FPGA target. In questo passo, vengono applicate ottimizzazioni per adattare la rappresentazione generica ottenuta dalla sintesi RTL all'architettura specifica dell'FPGA, predisponendo così la netlist ad una mappatura efficiente sulle risorse fisiche disponibili. Il risultato finale della fase di Design Synthesis è una netlist ottimizzata all'architettura specifica che è pronta ad essere processata nelle fasi successive del processo di progettazione FPGA.

Dopo aver completato la Design Synthesis, la fase successiva è quella di Design Implementation. Questa fase ha come input la netlist prodotta dalla Design Synthesis, e comprende di solito il processo di mapping, il placing, e il routing.

2.4.3 Mapping

Il technology mapping, spesso integrato direttamente nella fase di sintesi logica, è quel processo in cui gli elementi della netlist vengono mappati su specifici elementi logici disponibili nell'architettura dell'FPGA target. In altre parole, le funzioni logiche del design vengono mappate sulle LUT e sulle risorse logiche dell'FPGA.
Il mapping deve essere ottimizzato per ridurre l'uso delle risorse e migliorare le prestazioni del circuito. Uno degli obiettivi principali del technology mapping è infatti quello di ottimizzare l'uso di risorse per ridurre i tempi di propagazione e migliorare le prestazioni temporali.
In pratica, il technology mapping in un FPGA traduce una netlist generica in una configurazione ottimizzata di LUT, flip-flop e altre risorse specifiche dell'FPGA, bilanciando l'uso delle risorse, migliorando le prestazioni temporali e rispettando i vincoli di progetto.

2.4.4 Placement

Alla fine del technology mapping si ottiene un network di blocchi logici che sono pronti per essere posizionati sulle risorse fisiche dell'FPGA. La fase di placement, che segue il technology mapping, ha come obiettivo principale il determinare la posizione fisica ottimale dei blocchi logici e delle funzioni logiche all'interno del chip, tenendo conto di vincoli di performance, area e potenza.
Il placement cerca di posizionare i blocchi logici in modo tale che la successiva fase di routing possa ridurre al minimo la distanza delle interconnessioni necessarie, minimizzando la lunghezza totale del cablaggio delle interconnessioni usate. Questo è un aspetto importante poiché una minor lunghezza totale del cablaggio significa minori latenze e consumi energetici più bassi. Un altro aspetto altrettanto importante per il placement è garantire che i segnali possano propagarsi tra i blocchi logici entro i vincoli temporali specificati, essenziale per rispettare le frequenze di clock richieste e per evitare malfunzionamenti del circuito o la non implementabilità del design. In altre parole, il placement deve saper posizionare i blocchi logici su risorse fisiche in modo ottimale, tenendo conto delle informazioni spaziali e delle specifiche tecniche dell'FPGA target.
La qualità del placement ha un impatto diretto sulle fasi successive del design flow, in particolare sul routing. Un buon placement facilita un routing più semplice ed efficiente, riducendo la possibilità di congestione e migliorando le prestazioni

temporali del circuito. Inoltre, un placement ottimale può ridurre il consumo energetico e migliorare l'affidabilità del chip.

2.4.5 Routing

Una volta completato il placement, la posizione fisica assegnata ai blocchi logici è nota. Tuttavia, ancora da generare sono i percorsi fisici che i segnali devono intraprendere all'interno dell'FPGA per interconnettere i blocchi tra loro. Il routing è quella fase del design flow delle FPGA che si occupa del processo di generare l'instradamento dei segnali e dei dati attraverso la rete di interconnessioni all'interno del chip FPGA. Questa rete di interconnessioni collega le risorse tra loro e permette loro di comunicare secondo la configurazione specificata dal progettista.
Il router è il software che si occupa del processo di routing nelle FPGA per garantire che i segnali possano fluire correttamente attraverso il circuito. Per ogni collegamento logico tra due risorse, il router si occupa di determinare come propagare al meglio tale connessione attraverso le risorse ancora non occupate dalla trasmissione di altri segnali. Il routing risulta una delle parti essenziali del processo di progettazione delle FPGA e può avere un impatto significativo sulle prestazioni e sulla funzionalità dell'intero circuito.
Diversi fattori possono influenzare il router, come la disponibilità di risorse fisiche di instradamento, la lunghezza delle connessioni, le latenze intrinseche delle connessioni, e le restrizioni temporali. Metriche chiave per valutare la qualità del processo di routing riguardano sia le performance temporali, sia metriche come la wirelength totale utilizzata e il delay del critical path [3]. La configurazione finale ottenuta dal router dovrebbe utilizzare efficientemente le risorse disponibili dell'FPGA per evitare problemi come la congestione dei segnali, minimizzando al contempo la wirelength totale usata e il delay del critical path. Tali metriche sono fondamentali in quanto da essi dipendono limitazioni importanti, come ad esempio la frequenza massima di clock implementabile nel circuito.
Router che falliscono nel rispettare queste metriche potrebbero generare configurazioni di instradamento che non sono implementabili e possono essere affetti da diversi problemi, tra cui:

- **Ritardi di instradamento:** Se i percorsi disponibili sono congestionati o troppo lunghi, i segnali possono impiegare più tempo del previsto per raggiungere la loro destinazione. Questo può compromettere le prestazioni del circuito FPGA, specialmente in applicazioni ad alta velocità o a bassa latenza.

- **Utilizzo inefficiente delle risorse:** Se il routing non è ottimizzato, si potrebbero utilizzare più risorse di instradamento di quanto sia necessario, lasciando meno risorse disponibili per altre parti del design e rischiando una situazione di congestione dei segnali.

- **Congestione:** Un instradamento inefficace può portare alla congestione dei segnali, condizione in cui troppi segnali competono per le stesse risorse fisiche causando ritardi e malfunzionamenti nel circuito o perfino l'impossibilità di implementare il design.
- **Difficoltà nel soddisfare i vincoli di temporizzazione:** Il progettista deve garantire che i segnali raggiungano le loro destinazioni entro determinati limiti di tempo per evitare malfunzionamenti del circuito. Se il routing non è efficiente, potrebbe essere difficile soddisfare questi vincoli di temporizzazione.

In pratica, il processo di routing è fondamentale per garantire che il design dell'FPGA funzioni correttamente e in modo efficiente, ottimizzando l'uso delle risorse e rispettando i vincoli temporali e prestazionali del progetto.

2.4.6 Bitstream Generation

La generazione del bitstream è l'ultima fase dell'implementazione di un design su FPGA. Qui il design viene tradotto in un file binario, noto come bitstream, che può essere caricato nell'FPGA per configurarla. Il bitstream, essenzialmente, rappresenta in binario una mappa digitale del circuito, dove ogni componente logico è collocato e connesso in modo ottimale. Ogni dettaglio architetturale, dall'allocazione delle LUT ai percorsi di instradamento, viene codificato all'interno di questo file di configurazione.

2.5 Applicazioni

Le FPGA possono essere impiegate come soluzione per qualsiasi problema computabile [5], e grazie alla loro flessibilità e capacità di essere riprogrammate sono utilizzate in una vasta gamma di applicazioni. I campi in cui sono impiegate spaziano dai data center, all'ingegneria aerospaziale, all'automotive, al settore della difesa, all'intelligenza artificiale, all'IoT (Internet of Things) e ad una moltitudine di altri settori industriali[6].
Le FPGA sono anche utilizzate per accelerare via hardware delle funzioni che altrimenti sarebbero implementate via software. Grazie a questo, oggi sono spesso usate in situazioni che richiedono una risposta in tempo reale a bassa latenza o per gestire task computazionalmente pesanti. Esempi di questo tipo possono essere l'elaborazione dei segnali, l'impiego in sistemi di comunicazione per elaborare protocolli complessi, automazioni industriali e il settore delle reti neurali e machine learning. Inoltre, le FPGA trovano applicazione in ambiti critici come i sistemi di sicurezza e monitoraggio, dove è essenziale una risposta rapida e affidabile. Nei data center, le FPGA sono utilizzate per l'accelerazione delle operazioni di rete e

di storage, migliorando l'efficienza e la velocità dei servizi cloud.
Le FPGA possono anche rivelarsi uno strumento ausiliario al testing di circuiti integrati ASIC (Application-Specific Integrated Circuits). La funzione di riprogrammabilità delle FPGA rende infatti questi dispositivi particolarmente flessibili e adatti alla prototipazione di ASIC, consentendo di testarne il design prima della produzione di massa. Prima di fabbricare un ASIC, è essenziale verificare che il suo design funzioni correttamente. Le FPGA permettono ai progettisti di testare e validare il comportamento del circuito in un ambiente hardware reale, piuttosto che solo attraverso simulazioni software. Questo consente ai progettisti di identificare e correggere eventuali errori nel design, riducendo così il rischio di costosi rifacimenti nella fase di produzione. Le FPGA permettono iterazioni rapide del design, offrendo la possibilità di apportare modifiche e miglioramenti in tempo reale. L'uso di FPGA per la prototipazione consente anche di verificare l'integrazione del sistema in un ambiente, testando come l'ASIC interagirà con altri componenti hardware e software.

Capitolo 3

Graphics Processing Unit

L'idea alla base della tesi è quella di usare le GPU per velocizzare il processo di routing, ma perché usare la GPU? Qual è il vantaggio rispetto all'uso delle CPU? Per rispondere a queste domande, è necessario capire come è fatta l'architettura delle GPU ed esplorare quali siano le differenze con le CPU, e perché tale architettura e modello di esecuzione potrebbero sposarsi bene con il problema del routing delle FPGA.

3.1 Architettura generale di una GPU moderna

Le GPU (Graphics Processing Unit) sono dispositivi per la computazione di dati progettate per risolvere in modo concorrente problemi con alto grado di parallelizzazione. Inizialmente nate con lo scopo di essere impiegate nell'ambito grafico, delle immagini, e dei video, oggi sono anche utilizzate per risolvere problemi di natura concorrente in ambiti lontani da quello multimediale. L'uso delle GPU per risolvere problemi di natura variegata e non prettamente multimediale prende il nome di GPGPU (General-purpose computing on graphics processing units). Nel paradigma GPGPU, la GPU è vista come un coprocessore ausiliario da affiancare alla CPU, dotato di migliaia di core che processano dati in parallelo. CPU e GPU non lavorano in mutua esclusione, ma devono cooperare e lavorare assieme per ottenere performance ottimali. In figura 3.1 è raffigurato uno schema che mostra in modo semplificato un'architettura generica per una GPU. Questo paragrafo proseguirà con un approfondimento di tale architettura.
I blocchi logici più importanti all'interno di una GPU sono:

- un array di streaming multiprocessors (SMs)
- un sistema di memorie

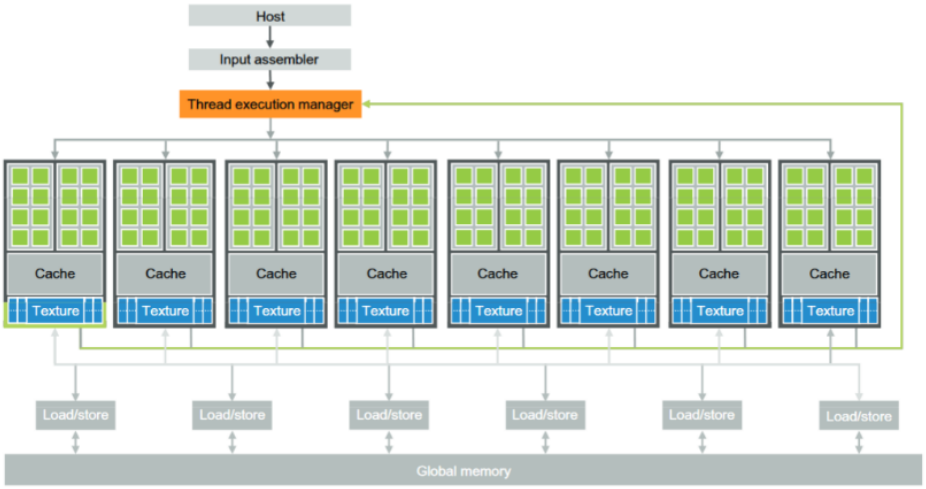

Figura 3.1: Architettura Nvidia

- un sistema di bus interno che mette in contatto gli elementi della GPU

3.1.1 Streaming Multiprocessors

Gli Streaming Multiprocessors (SMs) sono le unità fondamentali di una GPU al cui interno avviene la computazione, e sono responsabili della gestione dell'esecuzione parallela dei thread. Ogni GPU ha un certo numero di SMs, che a loro volta sono composti da un insieme di core, generalmente noti come Streaming Processors (SPs) o CUDA cores nelle architetture Nvidia. Ogni CUDA core è responsabile dell'esecuzione di un singolo thread o flusso di istruzioni alla volta. I thread sono raggruppati ed eseguiti in batch di 32, chiamati warp, che rappesentano l'entità basilare da schedulare. L'unità di schedulazione di una GPU gestisce la schedulazione dei warp, e non dei thread. I thread che compongono un warp sono infatti schedulati ed eseguiti contemporaneamente all'interno di un SM. In tutte le architettture Nvidia, la dimensione dei warp resta una costante di 32 thread. I chip appartenenti all'architettura Nvidia Pascal, ad esempio, possono contare su un massimo di 60 SMs su cui schedulare i warp, con ciascuno dei SMs composto da un totale massimo di 128 CUDA cores.
Il modello di esecuzione di un warp è Single Instruction Multiple Data (SIMD). Questo vuol dire che i thread all'interno di un warp condividono lo stesso instruction pointer e hanno la limitazione di accedere alle stesse istruzioni, ma possono essere eseguiti su diverse unità di esecuzione all'interno dell'SM e quindi possono eseguire tali istruzioni su dati diversi. Inoltre, ogni thread del warp eseguirà

la stessa istruzione in modo sincronizzato con gli altri thread dello stesso warp. Possiamo quindi immaginare il warp come un batch di thread sincronizzati sulla stessa operazione, ognuno dei quali che però lavora su un dato diverso, come per esempio (tipicamente) su un pixel diverso. Infine, è importante notare che i warp sono un blocco di esecuzione parallela totalmente trasparente al programmatore. Quel che tipicamente avviene quando un programmatore decide di eseguire un insieme di thread su un ambiente Nvidia è definire una griglia divisa in blocchi multidimensionali (1D, 2D, 3D) di thread di dimensioni arbitrarie. A runtime, ogni blocco della griglia sarà gestito da uno o più warp da 32 thread che saranno trattati come unità base su cui poter eseguire operazioni di scheduling.

3.1.2 Memorie

All'interno delle GPU abbiamo diversi tipi di memorie, che si distinguono per dimensione, velocità, e utilizzo.
La Global Memory è la memoria più grande, a cui si fa spesso riferimento tramite la sigla VRAM (Video Random Access Memory). È una memoria accessibile sia in modo diretto da tutti i thread del dispositivo, sia indirettamente dalla CPU tramite operazioni di trasferimento dati da VRAM a RAM. All'interno della GPU è la memoria più lenta in quanto non è on-chip, ma è un HW separato rispetto al chip in cui avviene la computazione. È generalmente usata come storage di grosse moli di dati, come immagini in fase di processing. Dal punto di vista software, si cerca di scrivere algoritmi per fare meno accessi possibile alla global memory e di sfruttare il più possibile gli altri tipi di memoria presenti sul dispositivo, qualora sia possibile.
Il tipo di memoria più veloce in assoluto sulla GPU sono i registri, piccole memorie collocate fisicamente all'interno dei core della GPU. Generalmente, per ogni thread è possibile trovarne un massimo di 32 o 64 da 32 bit. Sono usate come storage per le variabili locali che un thread usa durante l'esecuzione del suo flusso di istruzioni.
Come performance e collocazione fisica, tra i registri e memoria globale possiamo trovare altri tipi di memorie. In particolare abbiamo le cache, suddivise su vari livelli, e le shared memory.
Le shared memories sono piccole memorie condivise tra gruppi di threads appartenenti allo stesso blocco di computazione. Differentemente dalla VRAM e similmente ai registri, sono collocate on-chip, cosa che le rende molto veloci. Tuttavia non hanno una visibilità globale, ma solo locale al proprio gruppo di thread. Ogni thread può accedere solamente alla shared memory che gli compete, e non può accedere alle shared memory di altri blocchi di threads. Sono generalmente usate da chi scrive software come mezzo di data sharing tra threads dello stesso blocco computazionale, oppure come storage intermedio per dati collocati in memoria globale a cui si accede con alta frequenza, per ridurre il numero di accessi in

memoria globale. In altre parole, sono usate come strumento di comunicazione tra thread, oppure per realizzare pattern di accesso ottimizzato ai dati.
Le cache invece, a differenza delle shared memories, sono gestite dall'hardware stesso con meccanismi del tutto trasparenti al software. Il loro scopo è unicamente quello di ridurre il tempo di accesso medio ai dati che sono collocati sulla Global Memory. Questo viene realizzato interamente in HW, che copia nelle cache i dati a cui si fa accesso frequentemente secondo il principio di località spaziale e temporale. Nel momento in cui si richiede la lettura di un dato, se è presente in cache si accede a questa invece che alla Global Memory. Inoltre, a differenza delle altre memorie, sono suddivise su più livelli (tipicamente L1, L2 e L3), con ogni livello avente prestazioni e capienza differenti.

3.2 GPU o CPU?

Delineata l'architettura generale di una GPU, illustreremo in questa sezione le differenze principali tra queste e le CPU (Central Processing Unit), e i casi in cui in un applicazione sia opportuno impiegare l'una o l'altra per la computazione di un risultato.
Nelle applicazioni che hanno natura intrinsecamente parallela, l'uso delle GPU può portare ad un incremento in performance sostanziali, che può arrivare potenzialmente ad uno speedup di fino a 10-40 volte rispetto ad un approccio classico su CPU. Il motivo di tale boost in performance rispetto alle CPU va ricercato nell'architettura e nella diversa filosofia di design tra CPU e GPU. Le CPU sono progettate avendo in mente la massima efficienza su un flow sequenziale di istruzioni ed hanno un numero di core molto limitato, generalmente dai 4 ai 16. Le GPU, invece, sono progettate avendo in mente la massima efficienza nell'eseguire diversi flow in parallelo, e si prefiggono come obiettivo l'eseguire un numero elevato di threads nello stesso istante, potendo contare su migliaia di core pronti ad eseguire altreattanti thread contemporaneamente.
Le CPU dedicano molta più chip area per tecniche che velocizzano il singolo thread, tra le quali, unità di branch prediction sofisticate, e cache di grandi dimensioni rispetto a quelle presenti su GPU. Le GPU, d'altro canto, dedicano molta più chip area per supportare la concorrenza in modo intensivo ed efficiente, sacrificando l'efficienza nell'esecuzione sequenziale. La concorrenza è ottimizzarla con diverse tecniche, tra cui una gestione efficiente e leggera del thread switch, che su GPU viene gestita con uno scheduler implementato in hardware, che permette in un clock cycle di fare context switch. Su CPU, invece, operazioni come il context switch di un thread sono molto meno efficienti in quanto eseguite in SW e in più clock cycles. In altre parole, nel caso si voglia eseguire un'applicazione con una percentuale di execution time parallelizzabile molto bassa, si avranno performance molto migliori

su CPU, dove la priorità sta nell'ottimizzare le latenze e le performance del singolo thread. Le GPU, d'altro canto, avranno performance molto superiori se impiegate su applicazioni con un'alta percentuale di execution time parallelizzabile.
Eseguire un'applicazione scarsamente parallelizzabile su GPU porterà a scarsi risultati, perché non si sfrutta appieno la potenza di calcolo del dispositivo, in quanto molti core resteranno inattivi, e i pochi che svolgeranno lavoro attivo sono meno performanti rispetto ai core di una CPU. E viceversa, eseguire su CPU un'applicazione altamente parallelizzabile produrrà scarsi risultati, perché i pochi core disponibili, seppur più performanti ai core della GPU se presi singolarmente, dovranno eseguire in modo sequenziale la mole di lavoro che viene svolta in modo parallelo su GPU.

3.3 Limiti delle GPU

Nonostante le GPU siano dei dispositivi che permettono di raggiungere performance superiori per quanto riguarda applicazioni altamente parallelizzabili, sono dispositivi che vanno incontro a diverse sfide e limitazioni. Di seguito verranno elencati vincoli sulle prestazioni, limitazioni architetturali e problematiche di carattere generale che potrebbero rendere inappropriato l'utilizzo di una GPU per un'applicazione. Tali vincoli rappresentano le principali problematiche da tenere presente quando, all'inizio di un progetto, si valuta la possibilità di impiegare una GPU.

3.3.1 Vincoli algoritmici

In primis, non tutti gli algoritmi beneficiano dell'uso di una GPU. Come già illustrato nel paragrafo 3.2, le GPU non sono adatte ad eseguire algoritmi con basso grado di parallelizzazione. Queste sono infatti progettate per gestire in modo efficiente applicazioni che ne sfruttano appieno le risorse computazionali in parallelo. Tanto più un algoritmo sfrutta appieno l'architettura multicore della GPU, maggiore sarà l'incremento in performance. Nella pratica, non solo il problema si deve prestare bene ad essere parallelizzato, ma l'algoritmo che lo risolve deve essere scritto in modo adeguato affinché sia sfruttata appieno la potenzialità di parallelizzazione del problema che risolve.
Si può comprendere meglio il motivo per cui non tutti gli algoritmi si prestano bene ad essere accelerati tramite GPU grazie alla Legge di Amdahl. La Legge di Amdahl è un concetto fondamentale nel campo della computazione parallela che dimostra come lo speedup massimo complessivo che è possibile ottenere sia limitato dalla frazione temporale dell'algoritmo a cui non è possibile applicare lo speedup. Nel nostro caso, lo speedup rappresenta l'incremento di velocità ottenuto tramite la parallelizzazione dell'algoritmo. Mentre la frazione temporale su cui non è possibile applicare lo speedup è quella porzione temporale in cui l'algoritmo esegue

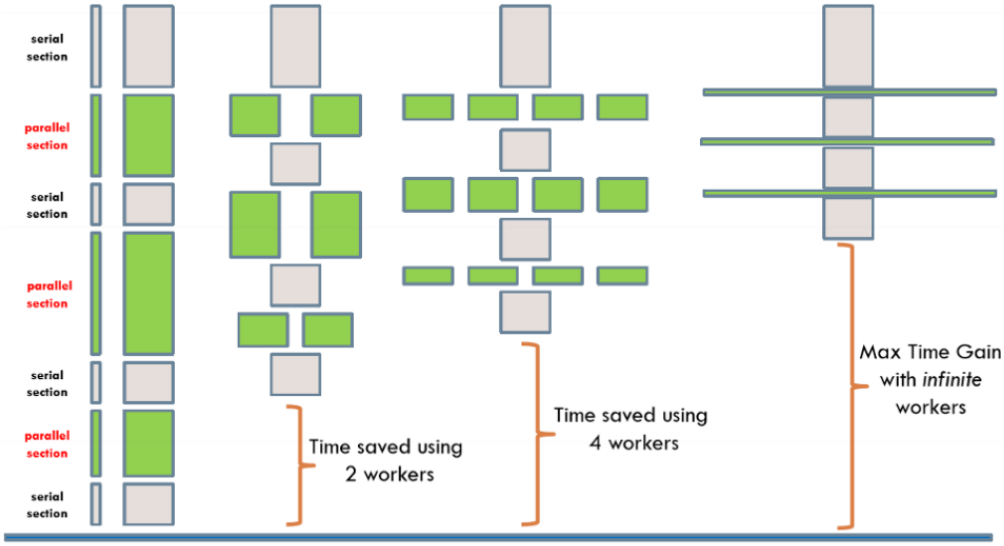

Figura 3.2: In verde, le sezioni parallelizzabili dell'algoritmo. All'aumentare del parallelismo, si nota come lo speedup complessivo massimo sia limitato dal tempo di esecuzione delle sezioni non parallelizzabili.

operazioni sequenziali che presentano dipendenze, e che quindi non si prestano ad essere parallelizzate. In termini matematici, la legge di Amdahl afferma che:

$$S = \frac{1}{(1-P) + \frac{P}{N}}$$

Dove P è la percentuale di execution time del programma a cui si è apportato uno speedup S.
Se costruissimo la GPU perfetta che permettesse di effettuare uno speedup infinito, e la usassimo per accelerare un programma del quale è possibile parallelizzare solo il 50% del tempo di esecuzione del codice, secondo la legge di Amdahl otterremo uno speedup complessivo pari ad un fattore 2. Da ciò si evince che più è piccola la percentuale di execution time che possiamo accelerare, meno impattante sarà lo speedup apportato a tale porzione sul tempo totale di esecuzione del programma. Nella pratica, questo comporta che stimare quanto sia grande la percentuale P sia la prima cosa da considerare quando si decide se usare una GPU o non usarla per accelerare un algoritmo. Nel caso non fosse possibile parallelizzare gran parte dell'algoritmo, è opportuno valutare se la soluzione su CPU sia più efficiente di una soluzione su GPU, anche tenendo conto delle altre limitazione che le GPU presentano.
Un altro limite relativo all'algoritmo utilizzato è il problema delle divergenze

all'interno dei warp. I warp di una GPU implementano il paradigma SIMD, il che vuol dire che i thread al suo interno hanno la limitazione di dover eseguire la stessa istruzione contemporaneamente in modo sincronizzato. Qualora diversi thread all'interno di un warp presentassero divergenze tra i loro flow, dovute per esempio a branch presi e non presi, loop con numero di iterazioni differenti, o divergenze nel flow di qualsiasi tipo, l'unità di scheduling del warp sarà costretta a separare in gruppi diversi i thread del warp, e ad eseguire questi gruppi in modo sequenziale. Ciò comporta un più basso utilizzo delle risorse della GPU in quanto si hanno in esecuzione meno thread contemporaneamente. Viene però ovviamente mantenuta l'esecuzione parallela tra thread dello stesso gruppo, poiché eseguono le stesse istruzioni in modo sincronizzato.
Algoritmi che non prestano attenzione al problema delle divergenze possono andare incontro a penalità di performance significative. In alcuni casi, se le divergenze sono eccessive o persistenti, l'esecuzione su GPU potrebbe essere meno efficiente rispetto all'esecuzione su CPU. Tuttavia, in tal caso, è bene precisare che nella maggior parte dei casi è possibile riscrivere l'algoritmo in modo tale da ridurre in modo significativo il numero di divergenze.
In generale, scrivere algoritmi che si preoccupano di ridurre al minimo il numero di warp che presentano divergenze risulta cruciale per ottenere un elevato utilizzo delle risorse della GPU e quindi alte performance.

3.3.2 Overhead di trasferimento dati

Il secondo problema da considerare è quello dell'overhead di trasferimento dati. Quando viene usata la GPU, all'inizio della computazione, i dati su cui si eseguono le operazioni sono trasferiti in copia dalla RAM alla memoria globale della GPU e, a fine computazione, dalla memoria globale della GPU alla RAM. Questa copia inserisce un overhead temporale, dovuto ai tempi di copia e alle latenze dei bus, che in certi casi non può essere trascurato. In base agli algoritmi implementati, può anche essere necessario trasferire dati da CPU a GPU e viceversa più volte all'interno del flow del programma, cosa che si ripercuote negativamente sulle performance.
L'overhead temporale dei trasferimenti dati si può ritenere trascurabile qualora occupi una piccola percentuale del tempo totale di esecuzione. La percentuale di tempo perso in overhead di trasferimento dipende principalmente dalla mole di dati su cui si effettua la computazione e sulla frequenza dei trasferimenti tra CPU e GPU (e viceversa).

- **La mole di dati:** Qualora per esempio usassimo le GPU per fare una computazione parallela su pochi dati, nonostante la fase di computazione effettiva sia più veloce nella versione con GPU, il tempo risparmiato sui

calcoli sarebbe meno del tempo perso per trasferire i dati, rendendo quindi la soluzione con GPU più lenta rispetto alla versione sequenziale su CPU. Figura 3.3 mostra graficamente questo problema.

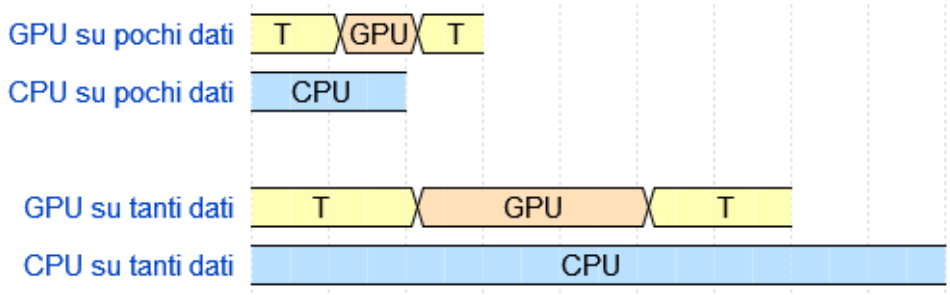

Figura 3.3: Su pochi dati le latenze di trasferimento non sono trascurabili e si ottengono performance peggiori nonostante la computazione sia più veloce su GPU. Su tanti dati, nonostante le latenze possano essere anche superiori rispetto al caso precedente a causa del trasferimento di più dati, le prestazioni della GPU sulla computazione dei dati compensano il tempo perso nella latenza dei trasferimenti.

- **La frequenza dei trasferimenti:**
  L'overhead diventa significativo per applicazioni che usano con frequenza il trasferimento dati, senza che utilizzano completamente le capacità di processing della GPU. Diventa cruciale per le performance scrivere algoritmi che minimizzano il movimento dati, massimizzano lo sfruttamento della data locality, e tengano i dati vicino laddove vengono processati.

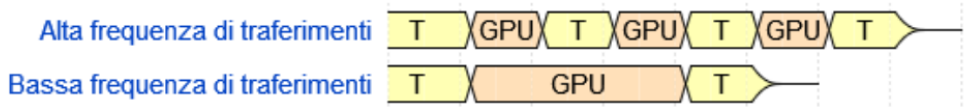

Figura 3.4: Il primo algoritmo esegue N trasferimenti, il secondo ne minimizza il numero. Minimizzare il numero dei trasferimenti riduce il tempo speso nelle latenze di trasferimento.

Nella pratica, un'applicazione che teoricamente dovrebbe avere performance migliori su GPU potrebbe essere più lenta rispetto alla versione CPU se non si preoccupa di considerare gli overhead dei trasferimenti dati.

3.3.3 Limitazioni di memoria

Quando si utilizza una GPU, ci sono diverse limitazioni legate alla memoria che è importante considerare durante lo sviluppo di un'applicazione. Per quanto riguarda la memoria globale, le GPU, rispetto alle CPU, hanno tipicamente molta meno memoria disponibile. Questo perché le CPU hanno accesso alla RAM, che è tipicamente più capiente della memoria globale delle GPU (VRAM), memoria su cui i thread della scheda grafica fanno operazioni di lettura e scrittura. Questo può diventare un problema qualora si lavori con GPU su grandi dataset o strutture dati complesse e durante la computazione si finisse la memoria disponibile. In applicazioni che fanno un pesante uso di dati, come il deep learning o simulazioni scientifiche, se non gestito in modo adeguato, far sì che i dati stiano tutti in memoria VRAM e che non si raggiunga il limite di memoria occupato può diventare facilmente fortemente limitante, oppure un collo di bottiglia di performance. Ottimizzare l'uso della memoria e impiegare tecniche che arginano queste limitazioni diventa quindi un altro degli aspetti cruciali nell'uso delle GPU, ancor più che su un corrispettivo algoritmo implementato invece su CPU.

3.3.4 Limitazioni varie

Infine, rispetto ad una soluzione su CPU, possiamo avere altri aspetti di cui dobbiamo tenere conto. In base al tipo di impiego e al tipo di esigenze specifiche, specialmente in ambienti in cui l'efficienza energetica è importante, il consumo energetico extra dovuto alla potenza assorbita dalla GPU può diventare un altro elemento decisivo che può influenzare la scelta di usare una GPU o meno.
Ancora un altro aspetto da considerare è l'eterogeneità delle architetture delle varie GPU, ognuna con limiti differenti per quanto riguarda le risorse disponibili. Scrivere un algoritmo su GPU, e scriverlo sfruttando in modo efficiente le risorse su ciascun hardware diverso può diventare un processo non banale. Non solo GPU appartenenti ad aziende diverse (AMD, NVIDIA e INTEL le principali) hanno architetture diverse che richiedono framework specifici che supportano l'esecuzione su tale eterogeneità di device, ma persino GPU prodotte dalla stessa azienda possono presentare tra loro sostanziali differenze di architettura che necessitano l'intervento del programmatore nel proprio codice affinché le risorse siano utilizzate in modo efficiente in ciascun device usato.

3.4 Analisi preliminare sull'impiego di GPU per il routing

Finora abbiamo delineato con una panoramica generale sia il problema da affrontare (quello del routing), sia il funzionamento delle GPU, con i relativi benefici e le

relative problematiche che questi device si portano dietro.
Il primo step da effettuare prima di implementare il router custom per le architetture di FPGA NanoXplore sarà quello di valutare se sia adatto o meno trattare il routing con un approccio su GPU. Dobbiamo quindi saper rispondere, in linea generale, se i vincoli descritti nel paragrafo precedente siano limitanti a tal punto da abbandonare l'idea dell'uso della GPU ancor prima di iniziare. Successivamente, una volta analizzato che i vincoli non siano un problema bloccante per il routing su GPU, è opportuno effettuare una stima iniziale grossolana di quelli che possono essere i benefici che ne trarremo nel usare una GPU. Ed infine, è opportuno visionare altri studi già presenti in letteratura riguardo l'uso delle GPU per implementare un router.
Analizziamo ora i principali vincoli presentati nel paragrafo precedente, tenendo a mente che il nostro obiettivo sia quello di scrivere un algoritmo di routing indirizzato principalmente verso le performance:

- **Natura parallelizzabile dell'algoritmo di routing:**
  Il vincolo più stringente per l'adozione delle GPU è trovare un algoritmo di routing che abbia un alto grado di parallelizzazione. Come già esposto, il routing è il processo di generazione delle route dei segnali attraverso la rete di interconnessioni interne al chip FPGA. Il suo scopo è quello di generare i percorsi che connettono fisicamente tra loro le varie risorse fisiche, e di farle comunicare tra loro come indicato dal progettista.
  Il router generalmente ha come input principale una unrouted netlist, ossia una descrizione delle risorse fisiche e di come si interconnettono logicamente tra loro. Tuttavia, è compito del router generare i path fisici che realizzano queste interconnessioni. L'output del router è una routed netlist, che ha le stesse informazioni della unrouted netlist ma con l'aggiunta delle informazioni di instradamento dei segnali.
  In altre parole, il router, a partire da un insieme di N connessioni logiche tra componenti fisici, deve generare i corrispondenti percorsi fisici.
  Già ad una prima analisi è evidente come ci siano opportunità per la parallelizzazione. La più evidente è la parallelizzazione della generazione degli N percorsi fisici. Nei paragrafi seguenti vedremo come ci siano anche altre possibilità di parallelizzazione a più basso livello. Il routing si presta quindi potenzialmente bene ad essere implementato tramite GPU, risultando ad una prima analisi parallelizzabile.

- **Trasferimenti dati e Quantità di memoria GPU:**
  Sia il numero di trasferimenti dati, sia la quantità di memoria usata dalla GPU sono dettagli di cui tener conto e che possono penalizzare pesantemente le performance se non gestite adeguatamente. Tuttavia, in entrambi i casi è possibile scrivere l'algoritmo in modo tale da mitigarne gli effetti negativi

e rendere marginale la perdita in performance. Non costituiscono quindi un problema.

- **Consumi energetici, eterogeneità degli HW:**
  I consumi energetici e l'eterogeneità dell'HW possono diventare, in certi contesti, aspetti importanti da considerare. Tuttavia, nel nostro caso non sono stati considerati come aspetti rilevanti.

Dopo una prima analisi con lo scopo di capire se fosse opportuno o meno usare il paradigma di programmazione concorrente per implementare un algoritmo di routing, possiamo provare a fare una stima grossolana sul potenziale incremento in performance rispetto ad una soluzione sequenziale su CPU. La fase di routing

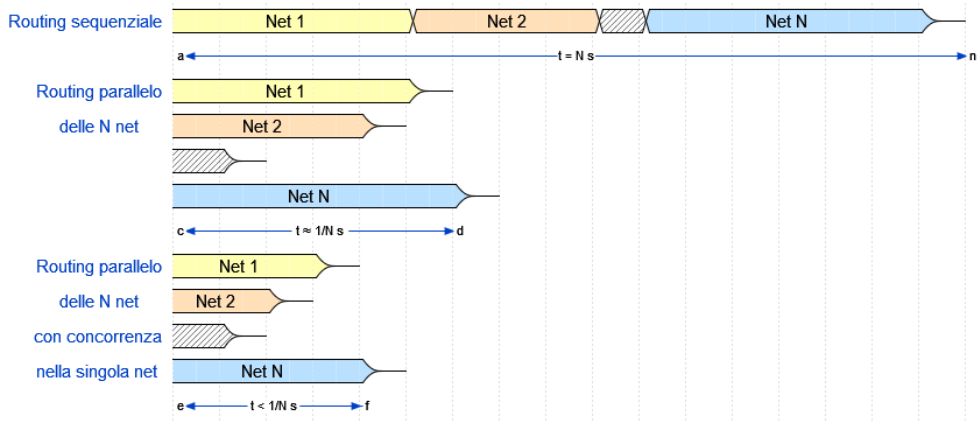

Figura 3.5: Confronto tra tempistiche teoriche per un router sequenziale, un router parallelo, ed un router parallelo che accelera anche la ricerca del path per la singola net

consiste nella ricerca del miglior percorso per N segnali diversi. La soluzione concorrente ideale sarebbe quella di far processare in parallelo la ricerca delle N route, garantendo uno speedup massimo di un fattore N rispetto ad un algoritmo che calcola le N route sequenzialmente. Nella pratica, raggiungere tale speedup non è possibile, sia a causa di ritardi dovuti a sincronizzazioni negli accessi alle strutture dati, sia a vincoli algoritmici che esploreremo nei paragrafi seguenti. Tuttavia, in aggiunta al calcolo concorrente delle N route, è possibile migliorare ulteriormente le performance usando la computazione concorrente per ridurre le tempistiche di ricerca della singola route. In figura 3.5 sono mostrate graficamente le tempistiche di computazione di N route con i tre approcci appena descritti.
Da questa analisi preliminare, si ritiene dunque che l'introduzione della computazione concorrente su GPU possa migliorare in modo sostanziale le performance della fase di routing nei tool CAD delle FPGA.

3.5 Routing e parallel computing: Stato dell'arte

Dopo una prima analisi generale sulla possibilità dell'uso di una GPU per il routing, approfondiremo ora i lavori presenti in letteratura. Presteremo particolare attenzione ai router che fanno uso della programmazione concorrente, in particolare ai router che sfruttano un algoritmo basato su GPU. L'impiego della computazione multithreading per migliorare le performance non è qualcosa di nuovo nell'ambito dei tool CAD e degli algoritmi di routing, con implementazioni che comprendono sia soluzioni basate su CPU, sia soluzioni basate su GPU.
Tra gli algoritmi di routing, di particolare rilevanza è il Path Finder Algorithm. La maggior parte dei router, anche quelli parallelizzati, sono varianti di questo algoritmo sequenziale [7]. Varianti di questo algoritmo sono anche state impiegate da due grandi produttori di FPGA, Xilinx e Altera, nei loro tool ufficiali [8], e diversi router usano questo algoritmo come base su cui proporre miglioramenti. Vediamone una breve panoramica:

3.5.1 Path Finder Algorithm

Il Path Finder Algorithm è un algoritmo di routing iterativo su CPU che rappresenta su un grafo, noto come il routing resources graph G(V,E), le risorse disponibili per il routing. Questo algoritmo fa uso di due meccanismi che gli autori chiamano "signal router" e "global router" per generare i path e gestire la congestione dei segnali (ossia il problema per cui più segnali si contendono le stesse risorse di routing).
Il signal router si occupa di trovare il path dei segnali basandosi su una ricerca in ampiezza (Breadth First Search, BFS) sul routing resources graph. La ricerca in ampiezza sceglie come path migliore il cammino con minor delay del segnale, e con minor congestione. Il global router si occupa invece di modificare dinamicamente ad ogni iterazione i costi di congestione dei nodi del grafo di risorse. In pratica, nella prima iterazione del Path Finder Algorithm i segnali vengono routati nel modo migliore possibile, senza tener conto delle congestioni. Ottenuti i path, il global router calcola il grado di congestione di ciascun nodo del grafo. Nelle iterazioni successive, il signal router rigenera i path dei segnali secondo le nuove informazioni di congestione dei nodi del grafo, e il global router aggiorna nuovamente le informazioni sulla congestione dei nodi. Le iterazioni si arrestano quando le N route degli N segnali non presentano risorse condivise, ossia quando si trova una configurazione senza congestione.

3.5.2 Router concorrenti su CPU

Diversi sono i router in letteratura che sfruttano tecniche di programmazione concorrente per massimizzare le performance. Un trend generale negli algoritmi di routing concorrenti è quello di combinare l'uso di tecniche di parallelizzazione fine-grained con tecniche di parallelizzazione coarse-grained nello stesso algoritmo. Di solito, il parallelismo fine-grained si riferisce a tecniche di parallelizzazione di basso livello, in particolare nel migliorare le performance nella ricerca della singola net. Mentre le tecniche note come coarse-grained si riferiscono di solito a tecniche di parallelizzazione che comprendono globalmente il processo di routing, ad esempio tecniche che migliorino le performance nella costruzione di più route. Vediamo ora alcuni router che fanno uso di tecniche di parallelismo su CPU, tra cui anche varianti del Path Finder Algorithm.

- **Accelerating FPGA Routing Through Parallelization and Engineering Enhancements [8]:**
  Questo è un router che sfrutta un parallelismo su 4 processi CPU. E' una variante del Path Finder Algorithm che propone di suddividere i segnali in 4 set differenti. Conoscendo le informazioni spaziali e fisiche riguardanti la collocazione delle risorse sul FPGA, la suddivisione dei segnali nei set avviene per segnali geograficamente affini e seguendo criteri per minimizzare la comunicazione tra processi. Ciascuno dei 4 set sarà poi associato ad un core differente, che lavorerà sul proprio set e che avrà anche il compito di sincronizzarsi con gli altri core tramite protocollo MPI, laddove ce ne sia necessità.

- **Serial-Equivalent Static and Dynamic Parallel Routing for FPGAs [9]:**
  Router che può essere accelerato da un numero arbitrario di processi CPU. Il suo obiettivo non è solamente quello di creare uno scheduler per le nets per apportare uno speedup, ma si prefigge anche l'obiettivo di produrre un output equivalente ad un router seriale eseguito sullo stesso set ordinato di nets. Lo scheduler è dependency-aware, ossia è in grado di riconoscere le dipendenze tra net, e il routing avviene per stage. Le net che possono essere processate in parallelo apparterranno allo stesso stage, mentre quelle che presentano dipendenze saranno schedulate in stage sequenziali diversi, in modo da preservare il flow delle dipendenze. Il routing delle net è eseguito quindi serialmente per le net che presentano dipendenze, mentre le net indipendenti possono essere processate in parallelo nello stesso stage.

- **Deterministic Multi-Core Parallel Routing for FPGAs [10]:**
  Router che combina una tecnica fine-grained con una coarse-grained, tecniche che usano rispettivamente un approccio multithreading e interprocess. Nella tecnica coarse-grained, in modo simile al primo router citato, si suddividono

le net in vari set da assegnare a diversi processi che comunicano tra loro tramite protocollo MPI. La tecnica fine-grained invece, che lavora assieme a quella coarse-grained, accelera ulteriormente le performance migliorando le prestazioni della costruzione di una singola route grazie all'ausilio del multithreading (posix threads). In particolare, l'esplorazione del routing resources graph è accelerata distribuendo il carico di computazione tra N threads.

3.5.3 Router concorrenti su GPU

Nell'ultima decina di anni, il general-purpose computing on GPU (GPGPU) è diventato sempre più popolare anche nell'ambito dei tool CAD per FPGA. E' possibile infatti trovare in letteratura diversi paper che implementano tecniche con GPU di programmazione concorrente su svariati step del flow dei tool CAD per FPGA [11][12][13].
Tuttavia, secondo le nostre conoscenze attuali, nonostante ci siano degli articoli scientifici affini algoritmicamente nel risolvere il problema del routing [14], pare ci sia una carenza di articoli che riguardano nello specifico l'implementazione di router su GPU. Tra i pochi studi disponibili, il più rilevante è quello presentato da Minghua Shen, Guojie Luo, e Nong Xiao, chiamato Corolla, e le sue varianti, sul quale ora ne seguirà una panoramica generale.

- **Corolla: GPU-Accelerated FPGA Routing Based on Subgraph Dynamic Expansion [15]:**
  Il flow generale dell'algoritmo rimane invariato rispetto a quello del Path Finder Algorithm, ossia anche su Corolla la congestione è ridotta in modo incrementale smantellando e ricostruendo le route ad ogni iterazione finché non viene trovata una configurazione senza congestione. Corolla implementa un algoritmo GPU-friendly di scoperta del cammino minimo sfruttando l'idea della riduzione della dimensione del problema limitando la ricerca in sottografi più piccoli. Il router Corolla sfrutta anche una tecnica di parallelismo fine-grained su GPU che combina parallelismo statico con quello dinamico. Infine, viene implementata anche una tecnica di parallelismo coarse-grained per parallelizzare la creazione di più net contemporaneamente. Questa tecnica presta particolare attenzione a mantenere il risultato finale equivalente ad un router che processa le net in modo sequenziale. Ciò è possibile raccogliendo nell'ordine sequenziale originale le net su cui fare routing in un set, per poi instradare in parallelo queste net finchè non se ne raccoglie una che presenti dipendenza con le net appena raccolte. Mantenendo l'ordine delle dipendenze ma processando in modo concorrente le net che non presentano dipendenze, è possibile ottenere performance migliori. Le dipendenze sono rilevate quando si raccolgono le net controllando se il sottografo della net appena raccolta si

sovrappone ad almeno uno dei sottografi delle net già raccolte e in fase di processamento.

Capitolo 4

NanoXplore e architettura delle FPGA NG-Medium e NG-Ultra

4.1 NanoXplore

Figura 4.1: Logo di NanoXplore.

Fondata nel 2010 con due stabilimenti in Francia a Sèvres e Montpellier, NanoXplore è un'azienda francese privata di semiconduttori fabless. Pioniera nell'ambito, è stata la prima azienda europea a sviluppare completamente chip FPGA resistenti alle radiazioni (rad-hard), compresa l'intera toolchain software necessaria [16].
La commercializzazione di dispositivi rad-hard ha permesso a Nanoxplore di guadagnare rapidamente popolarità nel mercato europeo, offrendo soluzioni avanzate che mitigano gli effetti delle radiazioni grazie a tecnologie come la ridondanza intrinseca delle risorse, la robustezza aumentata delle celle di memoria e un numero maggiore

di transistor nelle celle logiche. Questi miglioramenti assicurano una quasi completa tolleranza ai guasti indotti dalle radiazioni, anche se a un leggero aumento dei costi [17].
All'avanguardia nel design di circuiti integrati e nello sviluppo di FPGA, l'azienda è specializzata in prodotti destinati ad applicazioni ad alte prestazioni e mission-critical. Nanoxplore, con la sua forte presenza internazionale e più di 100 ingegneri di ricerca e sviluppo, e grazie alla sua catena di approvvigionamento europeo indipendente [18], si è affermata tra i leader nel mercato Spazio e Difesa, settori in cui la sicurezza e l'affidabilità sono di fondamentale importanza.
NanoXplore opera a livello globale e si pone tra i principali attori europei nell'ambito della progettazione e sviluppo di tecnologia SoC FPGA, collaborando con importanti aziende del settore spaziale come Thales Alenia Space, CNES ed ESA, formando una collaborazione importante per l'indipendenza dell'Europa in ambito spaziale della difesa [18]. Inoltre, oltre a proporre prodotti destinati all'utilizzo in campo spaziale e della difesa, l'azienda si impegna anche nella creazione di prodotti più commerciali [19].

4.2 FPGA Rad-Hard: NG-Ultra e NG-Medium

Nel suo catalogo di prodotti, NanoXplore presenta soluzioni progettate per essere safety-critical, fornendo componenti all'avanguardia adatti per essere impiegati in contesti come quello spaziale. Tali prodotti sono accompagnati da una suite software completa sviluppata internamente nota col nome di Impulse.
Per quanto riguarda il campo delle FPGA, per i suoi dispositivi NanoXplore può vantare un processo produttivo di celle SRAM che spaziano dai 65 nm fino ai 28 nm, con soluzioni sia embedded che non [17]. A partire dal 2014, NanoXplore si è focalizzata nello sviluppo di una famiglia di FPGA resistenti alle radiazioni per applicazioni spaziali e avioniche.
Attualmente, l'azienda è impegnata nella commercializzazione di diverse famiglie di componenti, tra cui le FPGA NG-MEDIUM e NG-ULTRA.

4.2.1 NG-Medium

NG-Medium di NanoXplore è una linea di FPGA progettata specificamente per applicazioni spaziali. Questa FPGA è infatti conforme agli standard spaziali internazionali, il che la rende idonea per una vasta gamma di missioni spaziali, sia commerciali che scientifiche.
Una delle sue caratteristiche principali che la rendono conforme all'uso nello spazio, è quella di essere resistente alle radiazioni. La resistenza alle radiazioni è una caratteristica essenziale per l'elettronica utilizzata nei satelliti e nelle missioni spaziali dove l'ambiente risulta estremamente ostile. In praticolare, nonostante le

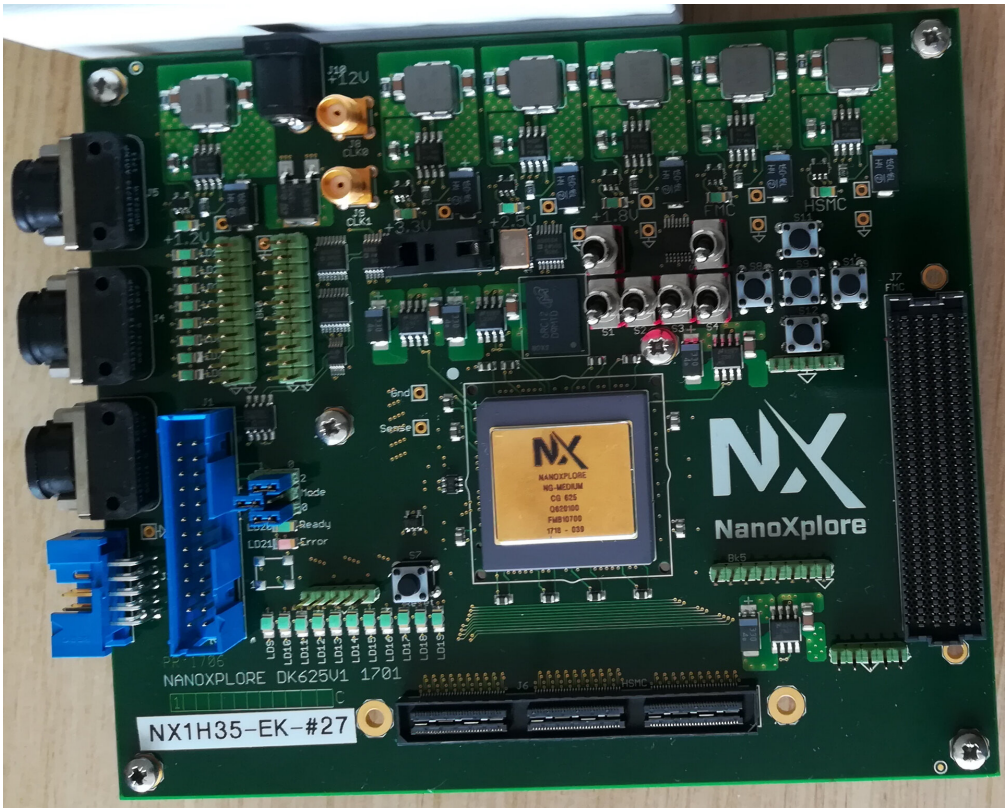

Figura 4.2: Development kit equipaggiato con FPGA NanoXplore NG-Medium.

FPGA siano dispositivi particolarmente sensibili alle radiazioni [20], NG-Medium è progettata specificamente per essere conforme con gli standard di resistenza ai Single Event Upsets (SEUs), ai Single Event Transient (SET) e alle Total Ionizing Dose (TID), effetti che sono comuni in ambienti spaziali. Recenti studi hanno analizzato l'impatto dei TID e SEU su una grande varietà di architetture avanzate, evidenziando l'importanza di progettazioni robuste testate con tecniche di fault injection in termini di resistenza alle radiazioni per mantenere l'affidabilità operativa in ambienti spaziali estremi [21] [22].
La resistenza alle radiazioni e l'affidabilità operativa in condizioni estreme rendono l'NG-Medium un chip affidabile ed ideale anche per l'uso in missioni spaziali di lunga durata. NG-Medium ha ottenuto la certificazione ESCC QPL per applicazioni spaziali rilasciata dall'Agenzia Spaziale Europea (ESA), rendendo questo dispositivo la prima FPGA europea qualificata per lo spazio [17].
Le FPGA di NanoXplore, incluse le NG-Medium, utilizzano un'architettura basata su SRAM che permette una grande flessibilità nella configurazione dei circuiti interni. Questo tipo di architettura consente di riconfigurare l'FPGA per diversi compiti anche dopo il lancio. La capacità di riconfigurazione post-lancio è un'altra

caratteristica importante, in quanto conferisce la capacità di aggiornare o modificare le funzionalità del satellite senza doverlo sostituire. Le FPGA di NanoXplore, incluse le NG-Medium, utilizzano un'architettura basata su SRAM con tecnologia CMOS a 28 nm.
I chip NG-Medium sono progettati per offrire un equilibrio tra prestazioni, capacità e consumo energetico, ed è ideale per applicazioni che richiedono risorse logiche robuste, memoria e interfacce di input/output (I/O) efficienti [17]. Per quanto riguarda l'efficienza energetica, questi chip sono progettati per operare a basso consumo energetico, rendendoli adatti anche ad operare su satelliti con risorse energetiche limitate, ma potendo comunque gestire applicazioni abbastanza complesse come l'elaborazione di immagini e segnali, controllo di volo, e comunicazioni satellitari.
La linea di FPGA NG-Medium di NanoXplore è particolarmente versatile e trova applicazione in diversi ambiti spaziali. Principalmente, sono dispositivi ampiamente utilizzati nei satelliti per le telecomunicazioni. Qui, gli FPGA NG-Medium svolgono un ruolo cruciale nella gestione della trasmissione e della ricezione dei segnali, migliorando significativamente l'efficienza e l'affidabilità delle comunicazioni in orbita.
Inoltre, questi FPGA sono anche ideali per le missioni di osservazione della Terra. Grazie alla loro capacità di elaborare le immagini catturate dai satelliti, gli NG-Medium contribuiscono a migliorare sia la qualità che la velocità con cui i dati vengono analizzati. Questo è fondamentale per applicazioni come il monitoraggio ambientale, la meteorologia e la gestione delle risorse naturali.
Le FPGA NG-Medium sono anche impiegate per i sistemi di navigazione spaziale. Svolgono un ruolo chiave nel controllo di volo e nella gestione dei sistemi di navigazione dei satelliti, garantendo precisione e affidabilità nelle operazioni.
Infine, un altro ambito in cui queste FPGA trovano impiego riguarda quello delle missioni scientifiche spaziali. Vengono utilizzate in una varietà di esperimenti scientifici, dove è indispensabile avere un hardware non solo affidabile ma anche flessibile come quello di NG-Medium. La possibilità di riconfigurare l'FPGA dopo il lancio consente agli scienziati di adattare l'hardware a nuovi requisiti e scoperte, aumentando l'efficacia delle missioni.
In sintesi, il design della linea di FPGA NG-Medium offre un grande vantaggio in ambito spaziale rispetto ai tradizionali circuiti integrati, in quanto si concentra in modo particolare sulla resistenza alle radiazioni. Questo, combinato alla sua capacità di riconfigurazione sul campo, permette a questi dispositivi di essere particolarmente idonei a missioni spaziali anche di lunga durata, rendendolo un componente chiave per la nuova generazione di satelliti, specialmente in un contesto in cui le missioni spaziali stanno diventando sempre più ambiziose e complesse.

4.2.2 NG-Ultra

NG-Ultra si colloca nella gamma di FPGA di fascia alta di Nanoxplore, rappresentando una delle offerte più potenti della linea di FPGA della società francese. Questi dispositivi sono ottimizzati per applicazioni che richiedono la massima capacità logica e prestazioni più elevate, e può contare su quattro core Arm® Cortex-R52 con frequenze di clock fino 600 MHz, ed è capace di eseguire un ampio array di Sistemi Operativi [19].
Come per i chip NG-Medium, sono particolarmente adatti per essere utilizzati nel settore aerospaziale e della difesa, e laddove sia cruciale garantire affidabilità e tolleranza ai guasti in ambienti estremi. I chip NG-Ultra, essendo anche loro chip per l'ambito spaziale, sono progettati per resistere ai danni causati dalle radiazioni spaziali [18], incluse le Single Event Upsets (SEUs) e le Total Ionizing Dose (TID). Anche questo chip utilizza un'architettura basata su SRAM, che offre una grande flessibilità permettendo di riconfigurare l'FPGA per diverse funzioni anche dopo il lancio, rendendolo estremamente versatile. Come già accennato, NG-Ultra è uno degli FPGA più potenti di NanoXplore, con una capacità di elaborazione superiore rispetto all'NG-Medium ed è adatto per applicazioni che richiedono un'elaborazione intensiva, e laddove ci sia bisogno delle massime prestazioni. NG-Ultra dispone infatti di una maggiore densità di elementi logici rispetto agli altri modelli, consentendo di implementare circuiti più complessi all'interno di un singolo FPGA, e supporta una vasta gamma di interfacce ad alta velocità, che lo rendono ideale per applicazioni che richiedono un'elevata larghezza di banda, come le comunicazioni satellitari avanzate.
Per quanto riguarda le applicazioni in cui questi dispositivi sono comunemente utilizzati, sono pressochè le stesse di NG-Medium. Ossia, sono impiegati nel campo delle telecomunicazioni, elaborazione di segnali complessi, osservazione della terra, elaborazione immagini ad alta risoluzione, sistemi di navigazione e controllo di volo, missioni scientifiche e nell'ambito dell'esplorazione spaziale.
L'NG-Ultra di NanoXplore si pone dunque come l'FPGA di punta progettato per affrontare le sfide delle missioni spaziali moderne e rappresenta un notevole passo avanti nel campo degli FPGA per applicazioni spaziali. Con le sue avanzate capacità di elaborazione, resistenza alle radiazioni e flessibilità architetturale, rappresenta una soluzione ideale per applicazioni spaziali critiche che prestazioni elevate e una grande larghezza di banda.

4.2.3 Confronto tra NG-Medium e NG-Ultra

Abbiamo appena esaminato le due principali soluzioni di FPGA per applicazioni spaziali offerte da NanoXplore: NG-Medium e NG-Ultra. Dopo averle analizzate in dettaglio nei rispettivi paragrafi, procederemo ora con un confronto diretto tra i due chip.

Entrambi i chip sono progettati per garantire un'elevata resistenza alle radiazioni, inclusi SEUs e TID, rendendoli affidabili per missioni spaziali di lunga durata. Le due soluzioni offrono una resistenza alle radiazioni simile, ma NG-Ultra potrebbe presentare potenziali miglioramenti grazie alle sue tecnologie avanzate. La robustezza alle radiazioni è un punto fermo per entrambi i modelli.
Passando alla capacità di elaborazione, NG-Medium bilancia prestazioni e consumo energetico, risultando ideale per applicazioni che richiedono un'elaborazione moderata. D'altra parte, NG-Ultra si distingue per la sua capacità di elaborazione superiore, rendendolo particolarmente adatto per applicazioni ad alta intensità computazionale, come l'elaborazione di segnali complessi e immagini ad alta risoluzione.
Dal punto di vista architetturale, entrambi i chip utilizzano una struttura basata su SRAM, che offre una grande flessibilità nella configurazione dei circuiti. Tuttavia, NG-Ultra vanta una maggiore densità di logica e supporta interfacce avanzate, permettendo configurazioni più complesse e prestazioni migliori rispetto all'NG-Medium.
Quando si considerano le interfacce e le comunicazioni, NG-Medium supporta una gamma di interfacce adatte per le telecomunicazioni satellitari standard e altre applicazioni spaziali. NG-Ultra, invece, supporta interfacce ad alta velocità più avanzate, rendendolo idoneo per applicazioni che necessitano di una maggiore larghezza di banda.
Per quanto riguarda l'efficienza energetica, NG-Medium è progettato per un basso consumo energetico, bilanciando perfettamente prestazioni ed efficienza. NG-Ultra mantiene un'ottima efficienza energetica nonostante le prestazioni superiori, sebbene possa consumare leggermente di più a causa della sua maggiore capacità di elaborazione.
Infine, le applicazioni tipiche in cui questi due chip sono usati sono le medesime, ossia in ambito spaziale e scientifico, con la differenza che NG-Ultra, essendo più performante, si presta meglio ad applicazioni che richiedono una computazione più intensa.
La tabella 4.1 mostra in modo schematico e riassuntivo le principali differenze.

4.3 Architettura delle interconnessioni di NG-Ultra

In questa Sezione verrà esaminata in modo generale l'architettura di NG-Ultra. Comprendere l'architettura di questo dispositivo è fondamentale per sfruttare appieno le sue caratteristiche e massimizzarne le prestazioni di un algoritmo di routing concorrente. Conoscere la posizione fisica delle risorse disponibili e il path

Caratteristiche	NG-Medium	NG-Ultra
Resistenza alle Radiazioni	Alta	Alta
Capacità di Elaborazione	Moderata	Alta
Architttura	SRAM	SRAM, densità più alta
Interfacce e Comunicazioni	Standard	Avanzate, alta velocità
Efficienza Energetica	Alta	Inferiore a NG-Medium
Applicazioni	Spaziali/scientifiche	Spaziali complesse

Tabella 4.1: Confronto tra NG-Medium e NG-Ultra.

che seguiranno i segnali su tali risorse è infatti essenziale per sfruttare tecniche di parallelismo che necessitano delle conoscenze spaziali delle risorse.
Va sottolineato che, sebbene ci concentreremo sulla descrizione dell'architettura di NG-Ultra, le considerazioni che ne derivano sono applicabili anche a NG-Medium, un'altra FPGA di NanoXplore con la medesima struttura architetturale, che vanta però di un numero minore di risorse fisiche. Le due architetture sono infatti molto simili, e pertanto le strategie e le considerazioni che verranno esposte saranno pertinenti per entrambe le soluzioni FPGA.

4.3.1 Lobes e Tube

L'FPGA NG-Ultra è composta da una porzione di area dedicata al SoC, un tessuto di risorse fisiche, e delle aree dedicate all'Input Output (I/O). In figura 4.3 si può vedere, nella parte superiore, l'area dedicata al SoC, mentre nella parte sinistra e destra sono collocate le aree dedicate all'I/O. Il tessuto di risorse fisiche è invece suddiviso in otto aree, denominate Lobes. I Lobes sono a loro volta formati da un alternanza tra Tile, Mesh e CGB.
L'area verticale tra i vari Lobes è denominata Tube. All'interno del Tube, tra ogni coppia di Lobes adiacenti è posizionata una system matrix, per un totale di quattro system matrix (system1, system2, system3 e system4). Le system matrix svolgono la funzione di propagare i segnali low skew all'interno dei Lobes adiacenti.

4.3.2 Segnali low-skew

Con *segnali low-skew* ci si riferisce a tutti quei segnali temporizzati in modo tale che il ritardo di propagazione tra sorgenti diverse sia minimizzato o, idealmente, uniforme. Il termine *skew* si riferisce infatti alla differenza di temporizzazione tra i

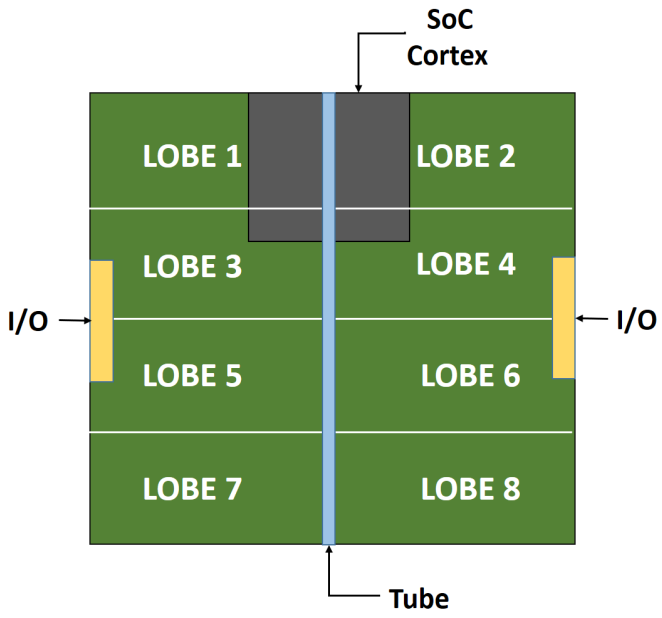

Figura 4.3: Schema ad alto livello di NG-Ultra.

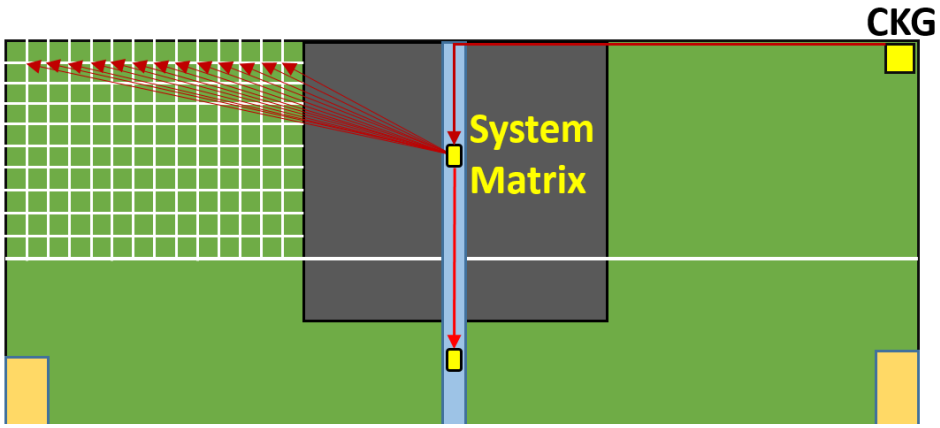

Figura 4.4: Visualizzazione grafica del percorso effettuato da un segnale low-skew di clock, dalla sorgente fino al raggiungimento delle Tile nei Lobes.

segnali in un sistema elettronico. Sono segnali per cui è essenziale garantire che arrivino ai vari elementi logici (come flip-flop o blocchi di logica) in modo sincrono e coerente. Se questi segnali arrivassero in momenti diversi, potrebbero verificarsi problemi come il metastability o il cattivo funzionamento del circuito. In altre parole, i segnali *low-skew* sono progettati per avere ritardi di propagazione uniformi o molto simili su tutta l'architettura, riducendo al minimo la differenza temporale

in cui arrivano nelle varie destinazioni.
Tra i segnali più importanti di low-skew abbiamo i clock, che su NG-Ultra sono generati da sette clock generators CKG1-CKG7. Di questi sette clock generators, quattro sono collocati agli angoli dell'FPGA, due sono collocati sugli angoli superiori del SoC, ed uno è collocato centrato sul bordo inferiore dell'FPGA.
I segnali di clock, insieme ad altri segnali low-skew generati nelle zone periferiche di NG-Ultra, vengono propagati verso le quattro system matrix situate nel Tube. Da qui, i segnali vengono ulteriormente trasmessi ai rispettivi Lobes (fig. 4.4). In particolare, le system matrix del Tube diffondono questi segnali nei network MSI situati all'interno delle Tile. Successivamente, questi network MSI inoltrano i segnali alle system matrix delle Tile stesse. Infine, la system matrix della Tile provvederà ad inoltrare i segnali low-skew a tutte le risorse della Tile che necessitano di tali segnali (fig. 4.5).
Riassumendo il percorso che effettuano i segnali low-skew, dalla sorgente vengono inoltrati nelle system matrix del Tube, per poi arrivare nei vari network MSI delle Tile, nelle system matrix delle Tile, ed infine alle risorse fisiche (dff, ecc...).

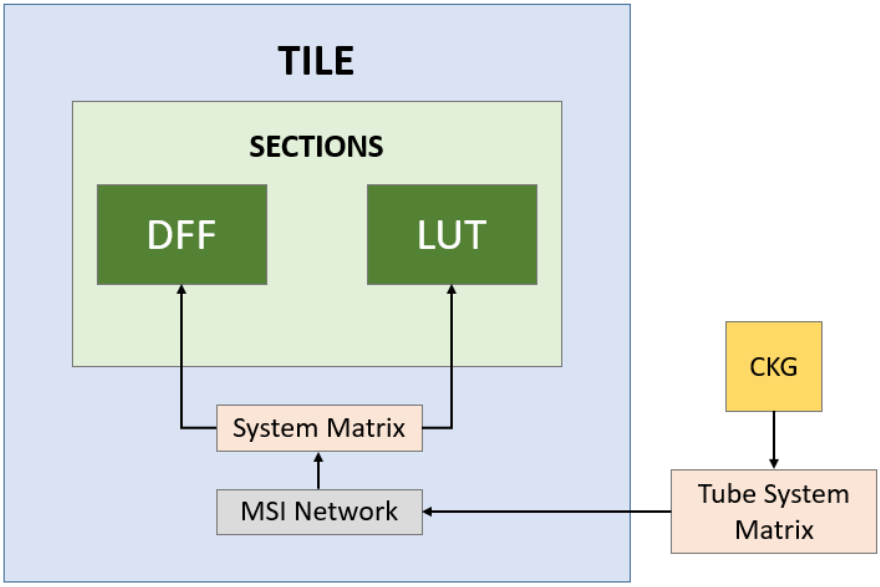

Figura 4.5: Schema semplificato del percorso che effettua un segnale low-skew di clock all'interno dell'FPGA.

4.3.3 Segnali common

Con *segnali common* si intende quei segnali che, pur essendo condivisi tra diversi blocchi logici, non richiedono vincoli di skew così stringenti come accade con la

categoria di segnali low-skew. Mentre segnali come quelli di clock e reset devono essere distribuiti con estrema precisione per garantire che arrivino simultaneamente (o quasi) a tutti i blocchi interessati, i segnali common possono tollerare una maggiore variabilità nei tempi di arrivo. Questo permette una maggiore flessibilità nel loro instradamento e utilizzo, rendendo più semplice la progettazione e l'implementazione del circuito senza compromettere le prestazioni del sistema.
Tra le risorse di routing importanti per l'instradamento dei segnali common troviamo le Mesh. Queste risorse di routing sono presenti nei Lobes assieme alle Tile e ai CGB, e hanno la funzione di formare delle interconnessioni che consentono l'instradamento dei segnali da una Tile (o CGB) all'altra. Analizziamo ora il percorso che effettua un segnale common:

- **Percorso verso le risorse della Tile:**
  Le Mesh permettono ai segnali di raggiungere la Tile destinazione attraverso i network di Command Input presenti nelle Tile stesse. I segnali che provengono dalla Mesh raggiungono la Tile attraverso il network BCI (Bottom Command Input network) se la Mesh è posizionata al di sotto della Tile, o attraverso il network TCI (Top Command Input network) se la Mesh è posizionata al di sopra della Tile. I network TCI e BCI possono quindi essere visti come interfacce di ingresso alle Tile per segnali common provenienti dall'esterno.
  I segnali common provenienti dalle Mesh, dopo essere stati inoltrati al network TCI (o BCI), raggiungeranno i network RE (dal francese réseau d'entrée, network di ingresso). NG-Ultra per ogni Tile ne presenta due, denominate rispettivamente RE1 e RE2. Le reti di ingresso RE hanno il compito di effettuare uno shuffle dei segnali, per poi instradarli verso le reti interne RI (dal francese réseau interne, network interno), di cui NG-Ultra ne presenta quattro per Tile. Infine, le reti interne RI porteranno i segnali di common a destinazione, ossia verso le risorse della Tile. Le risorse di una Tile sono suddivise in aree chiamate Sections, ciascuna composta da un insieme di coppie di righe, rispettivamente formate da LUT e DFF.

- **Percorso interno alla Tile:**
  Per raggiungere un'altra LUT della stessa Tile, il segnale partirà da una DFF, raggiungerà uno dei network interni RI, e verrà reindirizzato alla LUT destinazione. Questo avviene non solo per risorse appartenenti a Section diverse della stessa Tile, ma anche per risorse che appartengono alla stessa Section. Non è quindi architetturalmente possibile un collegamento diretto tra LUT, ma si passa sempre almeno per una RI.
- **Percorso tra Tile:**
  Per raggiungere una risorsa che appartiene ad un'altra Tile, bisogna prima far

arrivare il segnale ad una Mesh.
I segnali common, a partire dalla risorsa interna, devono passare per uno dei quattro network denominati RS (dal francese réseau de sortie, network di uscita). Qui avviene lo shuffle dei segnali, analogamente a ciò che avveniva per i network RE di ingresso. I segnali sono poi instradati dal RS network al TCO network (Top Command Output network) o al BCO network (Bottom Command Output network), in base se si esce dalla parte superiore o inferiore della Tile. TCO e BCO sono analoghi a TCI e BCI, ma per i segnali in uscita anziché in ingresso, e possono essere considerati come interfacce verso l'esterno della Tile. Da TCO/BCO si accede direttamente alle Mesh.
La Mesh provvederà poi ad inoltrare il segnale ad un'altra Mesh avente collegamento diretto al TCI (o BCI) network della Tile destinazione. Da qui il procedimento è il medesimo descritto nel punto *"Percorso verso le risorse della Tile"*, ossia Mesh, TCI/BCI, RE, RI, LUT/DFF.

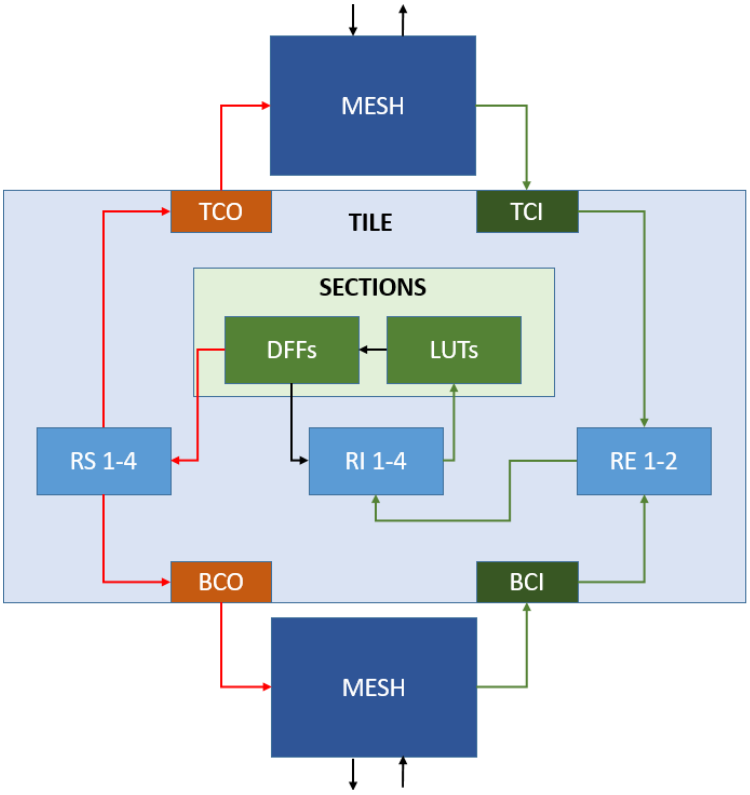

Figura 4.6: Schema semplificato delle interconnessioni interne alla Tile, e di come si interfaccia con le Mesh.

In sintesi, mostriamo in fig. 4.6 con uno schema un'approssimazione della struttra

di una Tile e di come fluiscono i segnali common in ingresso ed in uscita. In rosso è evidenziato il percorso che permette di raggiungere le risorse esterne alla Tile, mentre in verde è evidenziato il percorso che permette di raggiungere le risorse interne alla Tile. Notare come, per connettere due risorse che appartengono alla stessa Tile, non sia possibile connettere direttamente DFF(o LUT) con un'altra DFF(o LUT), ma ci sia bisogno di passare per le network interne RI. Non è invece necessario passare per le Mesh. Questa caratteristica architetturale tornerà utile per sviluppare una forma di parallelismo a granularità grossolana (coarse-grained parallelism) nel router.

4.4 Database rappresentativo delle risorse

In questo capitolo è stato innanzitutto presentata NanoXplore. Successivamente sono stati introdotti i suoi due principali prodotti della linea delle FPGA rad-hard, che si collocano rispettivamente nella fascia media e nella fascia alta tra i prodotti nel mercato, NG-Medium e NG-Ultra. Infine, è stata mostrata una panoramica generale sull'architettura delle interconnessioni di tali FPGA. L'obiettivo di questa sezione è quello di illustrare come tali risorse architetturali siano rappresentate all'interno di questa tesi.
Per poter operare, qualsiasi algoritmo di routing deve avere accesso alle risorse di routing fisiche messe a disposizione dall'FPGA e alle interconnessioni tra loro. Tali risorse e interconnessioni, in questa tesi, sono rappresentate tramite un database relazionale SQL. In particolare, per una questione di dimensioni, è stato scelto di sviluppare il router usando il database che rappresenta l'architettura di NG-Medium rispetto a quello che rappresenta NG-Ultra. Questo perchè le dimensioni del database rappresentativo di NG-Medium ammontano a circa 160 MB, una quantità di Byte di molto inferiore alle dimensioni che occupa il database che rappresenta NG-Ultra, che arriva ad occupare nella sua totalità diversi GB. Nonostante posseggano un numero molto diverso di risorse disponibili, entrambe le FPGA sono fondate sulla medesima architettura. Ciò ha permesso di sviluppare un router per NG-Medium che risulta automaticamente compatibile anche con NG-Ultra, garantendo così una soluzione versatile e adattabile a entrambe le piattaforme. Usare un database più snello permette infatti sia di iterare le modifiche e le fasi dello sviluppo in tempi più brevi, in quanto si accorciano di parecchi secondi i tempi di caricamento in RAM del database, sia di agevolare la fase di debugging.
Il database non è stato fornito da NanoXplore, né è stato estratto in questa tesi esplorando le risorse fisiche dell'FPGA. È stato invece fornito dal Dipartimento di Automatica e Informatica del Politecnico di Torino. In estrema sintesi, in questo progetto si è sviluppato un tool chiamato NXRouting che consente l'esplorazione delle risorse fisiche delle FPGA NanoXplore. Questo strumento ha permesso la

creazione di un database rappresentativo delle risorse fisiche, sul quale possono essere sviluppati algoritmi come quello di routing.
Il database SQL così ottenuto si compone di due tabelle. Una prima tabella rappresenta la totalità delle risorse fisiche di routing disponibili. La seconda tabella è composta da coppie di risorse fisiche e rappresenta invece quali risorse si possono raggiungere direttamente da una certa risorsa.

4.4.1 Tabella Risorse

Figura 4.7: Schema gerarchico del database.

Il database SQL è organizzato per riflettere una struttra gerarchica come quella mostrata in Fig. 4.7. Il plane racchiude in sè tutte le risorse dell'FPGA e rappresenta l'intero chip. A sua volta, il plane è suddiviso in diverse aree chiamate Zone. Importanti esempi di tipi di Zone sono le Tile, le Mesh, i CGB, ma anche risorse come i clock generator CKG. Ogni Zone è composta da diversi Network, che a loro volta sono un insieme di Device. Riguardo le Tile, tra le sue Network ne figurano alcune di cui già si è discusso in precedenza, come i network di ingresso RE, quelli di uscita RS, i network interni RI, i BCO, BCI, TCO e i TCI. Per le Mesh invece ci saranno altri tipi di Network, in base ai dettagli architetturali. Ad esempio, i Network delle Mesh sono S1, S2, S3, S4. I Device che compongono una Network, invece, sono dispositivi di vario tipo caratterizzati da un certo numero di Plug (o pin), suddivisi tra Input Plug e Output Plug.

Zone	Network	Device	Plug	Emitter
Zone A	Network 1	Device 1	Plug 1	Input
Zone A	Network 1	Device 1	Plug 2	Output
Zone B	Network 2	Device 3	Plug 3	Input
Zone B	Network 2	Device 3	Plug 4	Input
Zone B	Network 2	Device 3	Plug 5	Output
Zone B	Network 3	Device 4	Plug 6	Input
Zone B	Network 3	Device 4	Plug 7	Output

Tabella 4.2: Struttura della tabella SQL rappresentante le risorse fisiche di NG-Medium o NG-Ultra.

Tabella 4.2 mostra lo schema della Tabella SQL che riguarda le risorse fisiche. La colonna Emitter indica se il Plug del Device sia utilizzabile per ricevere un segnale in input oppure se sia utilizzabile per inoltrare un segnale in output. Oltre allo schema, nella tabella è anche mostrato un esempio di configurazione delle risorse fisiche. Una tabella SQL basata sul contenuto di Tabella 4.2 rappresenterebbe quattro Device e i loro Plug, distribuiti su tre Network distinti, collocati fisicamente in due Zone, A e B. Questa struttura riflette la complessità e l'organizzazione delle risorse fisiche delle FPGA NG-Medium e NG-Ultra.

4.4.2 Tabella Interconnessioni

Source	Target
Resource 1	Resource 2
Resource 1	Resource 3
Resource 1	Resource 4
Resource 5	Resource 2
Resource 5	Resource 3
Resource 6	Resource 4

Tabella 4.3: Struttura della tabella SQL rappresentante le interconnessioni dirette tra risorse.

Oltre a conoscere quali risorse fisiche sono disponibili, l'altra informazione architetturale essenziale è come queste risorse siano interconnesse tra loro. Questa informazione è rappresentata su una seconda tabella SQL del database, la tabella delle interconnessioni.
La tabella SQL Interconnessioni è composta semplicemente da una lista di coppie di risorse. La coppia è composta da una risorsa sorgente e una risorsa destinazione. Queste coppie rappresentano ovviamente solamente le interconnessioni unidirezionali dirette tra due risorse, e lascierà agli algoritmi il compito di trovare i path tra due risorse connesse indirettamente, ossia raggiungibili passando per altre risorse. Essendo connessioni unidirezionali, la risorsa di partenza sarà sempre un Output Plug, mentre quella destinazione sarà sempre un Input Plug. Non ha senso e non è quindi possibile avere come risorsa di partenza un Input Plug, che può solo ricevere un segnale. Analogamente, non ha senso avere come destinazione un Output Plug, che può solo inoltrare un segnale ad una destinazione. Unico caso particolare è quando un segnale arrivato in uno degli Input Plug sia inoltrato, passando internamente ad un Device, direttamente ad uno degli Output Plug dello stesso Device. In questo caso vi è effettivamente una connessione diretta tra Input

Plug e Output Plug, ma è un caso non presente come record nella tabella, e sarà gestito algoritmicamente durante l'algoritmo di routing.
Tabella 4.3 mostra un esempio di come sia fatta la Tabella SQL Interconnessioni. In questo caso, Resource 1, 5 e 6 faranno sicuramente parte degli Output Plug, e Resource 2, 3 e 4 degli Input Plug.

Capitolo 5

Parser e Placement

5.1 Sostituire il Router della Toolchain ufficiale Impulse

Prima di iniziare a codificare un algoritmo di routing, è essenziale capire come sostituire il router ufficiale del tool Impulse di NanoXplore con un router custom. Figura 5.1 mostra uno schema concettuale generale della toolchain di Impluse,

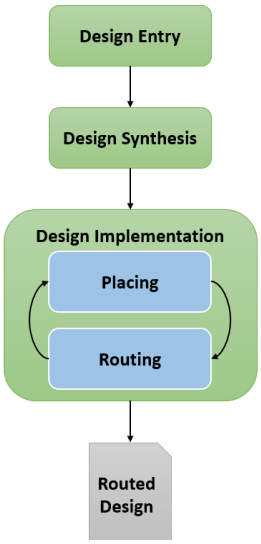

Figura 5.1: Schema della toolchain di Impulse. Placement e Routing non sono eseguiti separatamente ma avvengono assieme.

tool CAD sviluppato da NanoXplore per lo sviluppo sulle proprie FPGA. La cosa importante da notare è come le fasi di Placement e Routing non siano separate,

ma l'una necessita dell'altra per essere completate. Impulse effettua infatti un placement globale preliminare, seguito da una fase di routing globale. Dopo aver finito con il routing globale, Impulse torna sulla fase di placement. Questo approccio che alterna placement e routing rende impossibile per noi sostituire esclusivamente il router con uno custom. Sostituire il router della toolchain di Impulse significa anche sostituire la fase di placement.
Tuttavia, il focus della tesi è rimasto invariato, ossia la scrittura di un algoritmo di routing, a cui si è affiancata anche la stesura di un algoritmo di placement molto basilare, il cui unico scopo è permettere il testing di tale algoritmo di routing su un design sintetizzato. E' importante sottolineare che non si è preteso di scrivere un algoritmo di placement particolarmente efficiente o efficace.
Non sarebbe stata necessaria l'implementazione della fase di placement se, per i design da implementare su FPGA, fosse stato possibile estrarre in modo automatizzato le informazioni di placement necessarie in input al router dalla toolchain Impulse. E' possibile estrarre le informazioni di placement dalla toolchain ufficiale, ma non in modo automatizzato. Lo si può fare navigando la GUI (Graphical User Interface) e appuntando manualmente, per ogni segnale, a quale risorsa fisica sia stato assegnato. E' evidente come questa soluzione non sia sostenibile e scala male se si voglia testare il router su design mediamente complessi.
Nella pratica, si è provato il router su quattro benchmarks appartenenti alla suite di benchmark ITC'99 [23]. Del benchmark più semplice è stato ritenuto accettabile estrarre il placement manualmente dalla GUI e raccogliere tali informazioni su un file apposito. Riguardo invece gli altri tre benchmarks, il placement è stato generato con un algoritmo custom scritto appositamente per non dover estrarre manualmente dalla GUI di Impluse tali informazioni. In entrambi i casi, si è partiti dal file di output post sintesi prodotto dal tool Impulse e lo si è dato in input ad un nostro algoritmo che abbiamo chiamato *Parser e Placement*.

5.2 Parser e Placement

Per i motivi descritti nella sezione 5.1, ossia pricipalmente per poter testare il router su design sintetizzati, una parte della tesi è stata dedicata alla scrittura di un algoritmo di placement basilare che permettesse di assegnare risorse fisiche alle risorse logiche post sintesi.
Come mostrato in Fig. 5.2 e come accennato precedentemente, questo algoritmo riceve in input fino a due file distinti. Il primo è il file post sintesi generato dalla toolchain ufficiale di Impulse di NanoXplore. Questo file racchiude le informazioni riguardanti la descrizione e l'implementazione di un'entità VHDL, con dettagli specifici per essere utilizzata sull'FPGA NG-Medium di NanoXplore. Il secondo

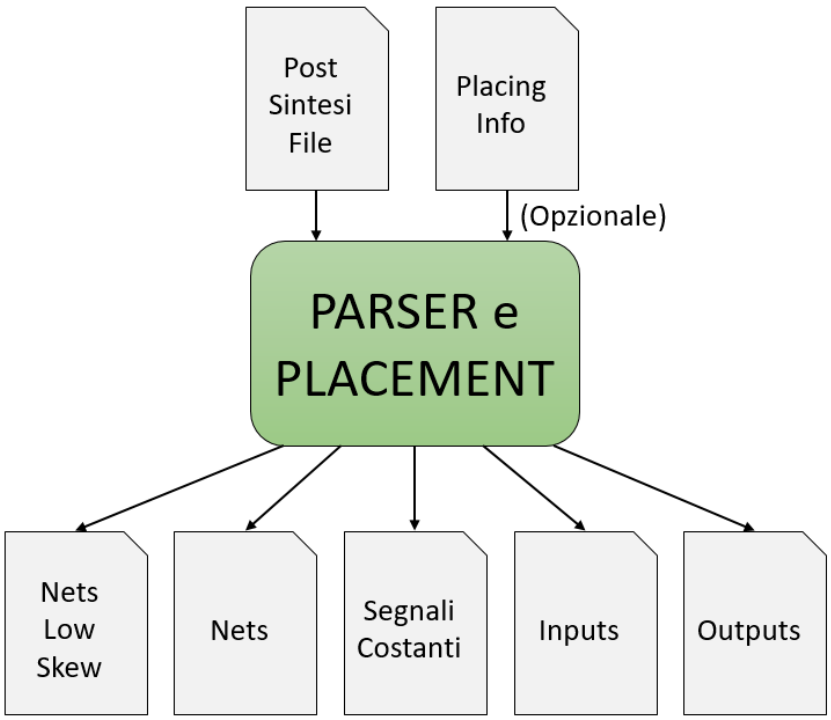

Figura 5.2: Schema dello script custom che si occupa della fase di placement.

è invece un file che si può opzionalmente fornire in input all'algoritmo. Questo file, trascritto a mano, racchiude le informazioni di placement estratte dalla GUI di Impulse. In altre parole, gli input di questo algoritmo di parsing e placement sono l'output della fase di sintesi e technology mapping della toolchain di Impulse (appena prima della fase di design implementation, placement e routing), e, se il design è abbastanza piccolo da permetterlo, anche le informazioni di placement manualmente trascritte. Altrimenti, se tale file non fosse disponibile, l'algoritmo di parsing e placement deve lui stesso occuparsi del placement, ossia di una possibile mappatura delle risorse logiche su quelle fisiche.
L'output di questo processo di parsing e placement consiste invece in un insieme di net suddivise su diversi file sulla base delle loro caratteristiche. Per praticità durante il routing, le net sono suddivise su cinque file distinti. Due di questi rispettivamente contenenti le net dei segnali low skew e le net dei segnali common, e gli altri tre rispettivamente contenenti le risorse fisiche occupate dai segnali costanti, dai segnali di input, e dai segnali di output del circuito che si sta implementando sull'FPGA.

5.2.1 Descrizione del file post sintesi

Il file VHDL post sintesi, che viene ricevuto in input dall'algoritmo di parsing e placement, racchiude informazioni riguardanti la descrizione e l'implementazione di un'entità VHDL specificamente progettata per essere utilizzata su FPGA della NanoXplore.
L'entità è inizialmente descritta dichiarando una serie di segnali che rappresentano gli ingressi e le uscite del design da implementare. Sono poi dichiarati altri segnali, interni all'entità, che interconnettono tra loro le varie risorse logiche. L'architettura è infine descritta elencando le risorse logiche necessarie, indicandone il tipo e, per ogni pin, specificando quale segnale lo occupi. Il segnale che occupa un pin è uno tra i segnali dichiarati in precedenza, ossia può essere uno degli ingressi o delle uscite dell'entità, un segnale interno, o un segnale costante (1 o 0). Esiste anche la possibilità che il pin non sia assegnato ad alcun segnale.
Per quanto riguarda la dichiarazione delle risorse logiche, il tipo di risorsa logica è un'informazione che deriva dalla fase di technology mapping, e indica la primitiva logica specifica dell'FPGA. Questa informazione è fondamentale per la fase di placement, e serve ad indicare su quale tipo di risorsa fisica deve essere implementata tale risorsa logica. Esempi di tipologie di risorse su NG-Medium sono NX_LUT e NX_DFF, che indicano rispettivamente le tecnologie di LUT e DFF presenti tra le risorse disponibili di tale FPGA.
Ricapitolando, le informazioni di interesse per il router presenti in questo file sono:

- I segnali di ingresso e uscita dell'entità
- I segnali costanti
- I segnali interni all'entità
- Le risorse logiche che compongono l'entità

5.2.2 Descrizione del file di placement

Il file di placement è un file di testo opzionale per questo algoritmo, ed è generato navigando la GUI e trascrivendo su file le informazioni di placement delle varie entità logiche. La presenza di questo file permette all'algoritmo di evitare la generazione di un placement custom. L'algoritmo, in questo caso, si occuperà soltanto di generare le net che servono al router partendo dal design sintetizzato ed avente le informazioni di placement.
In altre parole si usa questo file quando, nel sostituire il router ufficiale della toolchain con uno custom, si desidera mantenere le informazioni di placement generate dalla toolchain ufficiale. Questo è vantaggioso e preferibile rispetto a sostituire anche il placement perchè inanzitutto ci si può concentrare esclusivamente

sul router, e in secondo luogo il placement di Impulse sarà eseguito su criteri più avanzati rispetto a quelli di un nostro placement custom ausiliario, permettendo perciò meno problemi nella fase di routing stessa grazie ad un posizionamento più ottimizzato delle risorse fisiche.
La struttura di questo file consiste in tre colonne, che indicano rispettivamente il tipo di risorsa, il nome della risorsa logica, e la posizione fisica che quella risorsa logica occupa all'interno dell'FPGA. Il file è strutturato quindi in modo tale da fornire un mapping tra risorse logiche e risorse fisiche dell'FPGA.

5.2.3 L'algoritmo: il Parsing

L'algoritmo inizia con una fase di parsing, che si occupa della lettura dei due file di input e della costruzione e popolamento di strutture dati adeguate che ne rappresentino il contenuto.
Il parsing inizia con il leggere, se presente, il file di placement. Avviene una singola lettura del file e durante essa viene popolata una mappa che mette in corrispondenza il nome delle risorse logiche con le rispettive posizioni fisiche sull'FPGA. Per un problema di inconsistenze di nomenclatura nel contenuto dei due file, per ogni riga di questo file viene modificato il nome della risorsa logica presa in esame in modo che ci sia un match preciso con le nomenclature presenti nel file post sintesi.
Finito il parsing (opzionale) delle informazioni di placement, si passa alla lettura e parsing del file post sintesi. In questa fase, un oggetto di tipo entità viene creato e i suoi campi popolati man mano che procede la lettura del file. In particolare, l'oggetto entità raccoglie al suo interno tutte le informazioni più rilevanti del file post sintesi, tra cui figurano informazioni sui vari tipi di segnali dell'entità (input, output, segnali interni e costanti) e quali risorse logiche la compongono.
Successivamene alla lettura del file, si dovrà consultare il database delle risorse fisiche (si rimanda al paragrafo 4.4 per i dettagli) per poter capire se i pin delle risorse logiche siano pin di ingresso o di uscita. Questa informazione non è presente nel file post sintesi, ma è fondamentale per costruire correttamente le net, che sono l'output di questo algoritmo. Infatti, come già detto, il file post sintesi è un elenco di risorse logiche che indica per ogni pin a quale segnale sia assegnato. Sappiamo però anche che ogni segnale è generato da una singola sorgente, e può raggiungere più destinazioni. Questo significa che ogni segnale dichiarato in questo file sarà associato ad un solo pin di output e a più pin di input. Saper riconoscere quale sia il pin di output permette di poter ottenere un albero a due livelli la cui radice è il pin di output dove viene generato il segnale, e le cui foglie sono i pin di input raggiunti. Da tale struttura, è possibile ottenere le net che saranno l'input del router.
A questo punto, se sono presenti le informazioni di placement, le strutture dati

posseggono già tutte le informazioni necessarie per poter calcolare l'output. Altrimenti sarà necessario passare per il placement prima di poter generare le net che saranno l'input del router.

5.2.4 L'algoritmo: il Placement

Il placement è una parte facoltativa dell'algoritmo, ed avviene solo nel caso in cui non sia stato fornito il file con le posizioni fisiche delle risorse logiche. Questa fase avviene in modo molto semplice e diretto, senza considerare alcun vincolo particolare durante l'assegnazione delle risorse.
Durante il parsing, al momento in cui è popolata la struttura dati che rappresenta l'entità, si popola anche una struttura dati utile al placement. Tale struttura dati memorizza quante risorse sono usate per ciascun tipo di risorsa. Ad esempio, viene memorizzato tra le varie tipologie il numero di NX_LUT e NX_DFF utilizzate nel file post sintesi per poter implementare il design. Nel caso in cui non si hanno già le informazioni sul placement, questa struttura dati servirà per sapere quante risorse fisiche di ogni tipologia devono essere selezionate dal pool totale di risorse fisiche. Tabella 5.1 mostra un esempio di tale struttura dati.

Tipo	Quantità
LUT	n1
DFF	n2
CY	n3
IOB	n4
IOM	n5

Tabella 5.1: Struttura dati che immagazzina, per ogni tipo, quante risorse sono utilizzate per implementare il design.

Quando poi durante il parsing si legge il database e si carica in memoria RAM la tabella SQL delle risorse fisiche, per ogni tipo verranno selezionate un numero adeguato di risorse e inserite in un pool di risorse pronte ad essere assegnate a quelle logiche. Tabella 5.2 mostra come è strutturato questo pool di risorse.

Tipo	Risorse da assegnare
LUT	Lista LUT fisiche
DFF	Lista DFF fisiche
CY	Lista CY fisiche
IOB	Lista IOB fisiche
IOM	Lista IOM fisiche

Tabella 5.2: Rappresentazione della struttura dati che implementa il pool di risorse fisiche da cui attingere quando si assegna una posizione fisica ad una risorsa logica.

Questa operazione preliminare permette di implementare il placement come una semplice estrazione di risorse da un pool. Il pool è implementato suddividendo le risorse per tipo, e in base al tipo della risorsa logica a cui si sta assegnando una posizione fisica, si estrare dalla parte di pool corrispondente a quel tipo.
Diventa evidente che la complessità e l'efficacia di questo placement si sposta ora principalmente sul criterio in cui è riempito il pool di risorse fisiche da cui si attinge durante l'assegnazione. Il criterio di selezione è stato realizzato semplicemente secondo l'ordine di lettura della tabella SQL. La tabella SQL viene letta e caricata in memoria riga per riga secondo una query SQL, e non appena si legge una risorsa appartenente ad uno dei tipi ricercarti la si aggiunge al pool, fino a raggiungere un numero sufficiente a soddisfare tutte le assegnazioni di risorse logiche.
La tabella SQL viene letta però in modo ordinato rispetto al nome delle Zone e dei Network, grazie ad una operazione SQL di Order By. In questo modo, il pool di risorse viene riempito secondo quest ordine, e di conseguenza l'assegnazione delle risorse avviene secondo il criterio di ordinamento crescente di Zone e Network dei record del database. Questo permette di scegliere risorse fisicamente vicine in quanto probabilmente appartenenti alla stessa Zona e Netowrk, o comunque in Zone e Network adiacenti, selezionando invece Device che possiamo considerare randomici all'interno dei Network.
Questo criterio da una parte permette di assegnare risorse vicine tra loro, comportando di conseguenza path mediamente più corti nella fase di routing. D'altro canto, assegnare troppe risorse spazialmente vicine aggraverà problemi come quello della congestione nella fase di routing. Si è quindi a conoscenza che questo criterio non è un criterio ottimale. Tuttavia, il nostro obiettivo resta quello di concentrarci sul router e sul suo testing, e non sul placement, e accettiamo di avere problemi come un tasso di congestione più alto rispetto ad un algoritmo di placement ben ottimizzato.

5.2.5 L'algoritmo: Generazione delle Net

Superate le fasi di parsing e placement, l'algoritmo passa alla fase finale, ovvero la generazione dell'output. Questa fase può essere vista come una fase di preprocessing antecedente all'operazione del router, producendo un output pronto per essere processato dal router stesso.
L'output generato in questa fase, come mostrato in Fig. 5.2 e discusso in precedenza, consiste in un insieme di net suddivise su cinque file distinti, ciascuno basato su caratteristiche specifiche. I cinque file raccolgono rispettivamente le net riguardanti segnali low-skew, le net riguardanti segnali common, le risorse fisiche (pin/plug dei Device) occupate da segnali costanti, e le risorse occupate dai segnali di input e output del design da implementare.
Per produrre tali output, questa fase inizia preparando una mappatura tra ogni segnale e i pin che utilizza. Ogni entry di tale mappatura si riferisce quindi ad un segnale differente. Tali segnali, sulla base delle loro caratteristiche, dovranno essere catalogati in una tra le cinque categorie che corrispondono ognuna ad uno dei cinque file di output.
Il primo passo per catalogare i segnali sarà quello di riconoscere quali tra i vari segnali siano segnali low-skew. Questo processo avviene identificando quali siano i segnali low-skew in input all'entità, e considerando che ogni altro segnale che li propaga sarà considerato anch'esso low-skew. I segnali di reset e clock sono un esempio di segnali low-skew di input all'entità sicuramente presenti in ogni implementazione. Identificati tali segnali, si procede in modo ricorsivo per determinare quali segnali li propagano catalogando ciascun segnale come low-skew e, nel caso di clock e reset, fermandoci nella ricorsione quando si individuano i pin dei Device che ne impostano clock e reset.
Sulla base della mappatura appena creata tra segnali e pin usati da tali segnali, e sulla base delle informazioni di quale segnale sia low-skew, si cataloga ognuno dei segnali. Analizziamo ora nel dettaglio i tipi diversi di entry nella mappa:

- La entry relativa a un segnale interno si riferisce a N pin, di cui uno è un pin di output e N-1 sono pin di input. Sono create N-1 net aventi come pin sorgente il pin output, e come pin destinazione uno dei pin input. Queste net sono memorizzate nel file di output come coppia di pin, nel formato:
  risorsa1 risorsa2
  Dove risorsa è espressa a sua volta nel formato:
  Zone:Network:Device:Plug:Emitter

- La entry relativa a un segnale catalogato come low-skew sarà trattata analogamente a quelle relative ai segnali interni, con la sola differenza che cadono in due categorie diverse e saranno stampate su due file output separati.

- La entry relativa a un segnale costante mapperà il segnale "0" o "1" con N pin che possono essere di input o output. In questo caso non si può parlare propriamente di creazione di net, perchè non formiamo alcuna coppia di pin. Quel che sarà salvato sul file di output e che è rilevante per il processo di routing sarà, per ciascuno dei pin, una riga che contiene quale pin sia occupato da quale segnale ("0" o "1"). Queste righe indicano al router quei pin che non possono essere usati come risorse di routing e devono essere impostati come occupati durante la fase di routing. Il file di output corrispondente conterrà righe nel seguente formato:
  risorsa1 segnale(1/0)
  Dove risorsa è espressa a sua volta nel formato:
  Zone:Network:Device:Plug:Emitter

- Le entry relative a un segnale di entity output e quelle relative a un segnale di entity input saranno trattate in modo simile ai segnali costanti, ossia saranno create righe che indicano che i relativi pin dovranno essere impostati come occupati durante il routing. L'unica differenza è il formato, che non indica più sia la risorsa che il segnale, ma solo la risorsa. Nel caso ce ne fosse bisogno, il router riconoscerebbe quali siano le risorse occupate da un segnale di entity input e quale da un segnale di entity output grazie alla separazione su due file differenti.

5.3 Problemi e Limitazioni

La parte più problematica di questo algoritmo risiede nel criterio di selezione delle risorse fisiche.
Un algoritmo che ordina le risorse fisiche in ordine alfabetico e assegna alle risorse logiche le prime risorse fisiche della tipologia adatta che incontra presenta sicuramente problemi e limitazioni. Usare questo approccio ha una probabilità molto alta di portare ad un placement subottimale. E' dunque possibile andare incontro ad uno o più dei seguenti problemi:

- **Congestione**
  Assegnare risorse fisiche troppo vicine tra loro può portare a congestione in alcune aree dell'FPGA. Se molte risorse logiche vengono assegnate a una regione specifica, le interconnessioni tra queste risorse possono diventare eccessivamente dense, aumentando inoltre i tempi di propagazione e il consumo energetico.

- **Imbilanciamento nell'uso delle risorse**
  Un algoritmo che usa questo criterio non può bilanciare bene l'utilizzo delle

risorse su tutta l'FPGA in design medi e grandi. Alcune aree possono risultare sovraccariche mentre altre rimangono sotto-utilizzate, causando inefficienze e possibili vincoli di routing. Questo può aumentare il percorso critico del segnale, causando ritardi maggiori e potenzialmente compromettendo le prestazioni del circuito.

- **Vincoli di Timing**
  L'algoritmo non tiene conto dei vincoli di timing critici del design, portando potenzialmente a comunicazioni che non possono rispettare tali timing a causa di un placement non corretto.

- **Problemi di Power Distribution**
  Distribuire le risorse logiche senza considerare il consumo energetico può portare a problemi di distribuzione della potenza. Le aree con molte risorse logiche attive possono soffrire di surriscaldamento o problemi di alimentazione, mentre altre aree rimangono sotto-utilizzate.

- **Scalabilità**
  L'algoritmo non può scalare bene per design complessi. Man mano che il numero di risorse logiche e fisiche aumenta, un approccio così semplice diventerebbe inefficiente, richiedendo modifiche significative per mantenere prestazioni accettabili.

5.3.1 Considerazioni finali

Senza il datasheet, mancano informazioni essenziali come i delay delle connessioni tra risorse e le informazioni spaziali, rendendo impossibile eliminare realmente tali problemi per un placement custom. Attualmente, abbiamo solo informazioni su quali risorse sono disponibili e come sono connesse, e possiamo dedurre la collocazione spaziale delle risorse dalla nomenclatura e consultando la GUI Impulse, ma non sono disponibili informazioni su altri dettagli critici necessari per un placement efficace. Senza questi dati, l'algoritmo non può considerare la latenza delle interconnessioni né la posizione fisica precisa delle risorse, portando a un placement che non può essere adeguatamente ottimizzato per performance, consumo energetico e distribuzione del carico.
Nella pratica, la parte di placement dell'algoritmo non può essere considerato adatto per implementare un placement efficace. Tuttavia, fin dal principio il suo scopo non è mai stato quello di generare un placement ottimale, ma piuttosto quello di fornire una base che permettesse di connettere in modo automatizzato un router custom alla toolchain Impulse qualora non fosse possibile estrarre il placement da Impulse.

Qualora invece si immettessero manualmente le informazioni di placement derivanti dalla toolchain Impluse, questo algoritmo produrrebbe come output una rappresentazione del design adatta ad essere processata da un router custom.

Capitolo 6

Router

Il processo di design e di codifica dell'algoritmo di routing rappresenta l'attività principale della tesi. Si rimanda alla Sezione 2.4.5 per la definizione del routing.
Il router sviluppato è stato progettato per essere compatibile con l'architettura di NG-Medium, in particolare con il suo database che ne rappresenta le risorse fisiche, descritto nella Sezione 4.4.
Il problema principale che questa tesi e che questo router vogliono affrontare è come poter implementare un algoritmo di routing su GPU in modo che sia performante in termini di tempo di esecuzione. Infatti, quando si traduce un file di descrizione hardware HDL in un file di bitstream per design complessi, la compilazione può richiedere ore o persino giorni per completarsi [8]. Questo riduce la produttività e incrementa significativamente i costi di sviluppo e il tempo necessario per rilasciare un design su FPGA sul mercato. Il placement e il routing emergono spesso come le fasi più lente nei tool CAD su FPGA [9][10].
Il nostro obiettivo di sviluppare un router performante su GPU nasce dalla necessità di affrontare questi problemi, migliorando l'efficienza del processo di compilazione. Implementando l'algoritmo di routing su GPU, miriamo a ridurre drasticamente i tempi di esecuzione, aumentando così la produttività e riducendo i costi di sviluppo. Inoltre, un approccio di questo tipo può contribuire a rendere il rilascio di design su FPGA molto più rapido e competitivo sul mercato.
Questo obiettivo richiede non solo l'implementazione di una strategia su GPU che velocizzi l'algoritmo, ma anche un'attenzione particolare alle prestazioni e all'efficienza delle parti dell'algoritmo non implementate su GPU.
In questo capitolo verrà presentata la rappresentazione delle strutture dati utilizzate, spiegando sia il come che le ragioni delle scelte di design. Successivamente sarà esposto come opera l'algoritmo. Infine, verranno illustrate le tecniche di parallelizzazione adottate per ottenere elevate performance.

6.1 Routing Resources Graph

Data una lista di net (i.e. coppie di pin sorgente e pin destinazione, note anche col nome di routes nella terminologia di altre FPGA), il routing consiste nel trovare, per ciascuna net, il path migliore che le connette. Ciascun path, per essere adatto a connettere una net, deve sottostare ai vincoli dettati dal segnale che deve instradare. Tali vincoli possono essere ad esempio relativi ai tempi di propagazione e di consumo potenza. Tra i possibili path che rispettano tali vincoli, il routing deve scegliere il migliore, ossia quello che minimizza una funzione di costo. La funzione di costo può essere relativa a parametri come il ritardo del segnale, la wirelenght usata, il numero di risorse attraversate, o ad una combinazione di queste.
Per costruire e valutare i vari path possibili che connettono una net, l'algoritmo dovrà inevitabilmente attraversare ed esplorare la rete di risorse fisiche e infine decidere come effettuare la connessione, alla ricerca del path migliore secondo i criteri poco fa elencati. A tal proposito, è necessario caricare in memoria le risorse fisiche e rappresentarle in modo da agevolare la scrittura di un algoritmo che esplori la rete di risorse. Non bisogna però dimenticare l'obiettivo iniziale, ossia le performance.
La rappresentazione delle risorse fisiche usata nel database (Sezione 4.4) non è adatta a tale scopo. Questa rappresentazione si presenta come una coppia di tabelle SQL, una contenente la lista di risorse fisiche, e l'altra contenente le interconnessioni dirette tra due risorse fisiche. Questo approccio non facilita lo sviluppo di un algoritmo efficiente per l'esplorazione ricorsiva o iterativa della rete di interconnessioni. Per determinare quali risorse sono direttamente raggiungibili da una data risorsa, sarebbe necessario ricorrere alla memorizzazione in RAM delle tabelle, e all'ordinamento dei record della tabella di interconnessioni. Una volta ordinata, andrebbe fatta una ricerca dicotomica per individuare i record relativi alle interconnessioni che riguardano tale risorsa. L'alternativa di interrogare il database con query mirate è del tutto esclusa, poiché esse rallenterebbero il processo di routing e vanificherebbero ogni ottimizzazione del resto dell'algoritmo.
Scartata tale rappresentazione, questa è stata convertita in un formato più agevole. In letteratura, le risorse necessarie ad un router sono rappresentate con una struttura dati nota come *routing resources graph* [7][24]. Il routing resources graph è un grafo orientato G(V,E) i cui vertici V_i sono risorse e rappresentano o i pin o i collegamenti di un pin di output verso un pin di input, e i cui archi E_i sono le interonnessioni tra vertici. La scelta di cosa rappresentare con i vertici V_i è libera, e nel nostro caso i vertici V_i rappresenteranno i pin dei Device, mentre con E_i rappresenteremo le interconnessioni dirette tra un pin e un altro. D'ora in avanti i termini *vertici*, *nodi* e *pin* saranno usati in modo interscambiabile.
È tuttavia possibile a sua volta implementare il grafo delle risorse in diversi modi. Sono state provate due varianti:

- rappresentazione con nodi dispersi in memoria
- rappresentazione con nodi contigui in memoria

6.1.1 Rappresentazione con nodi dispersi in memoria

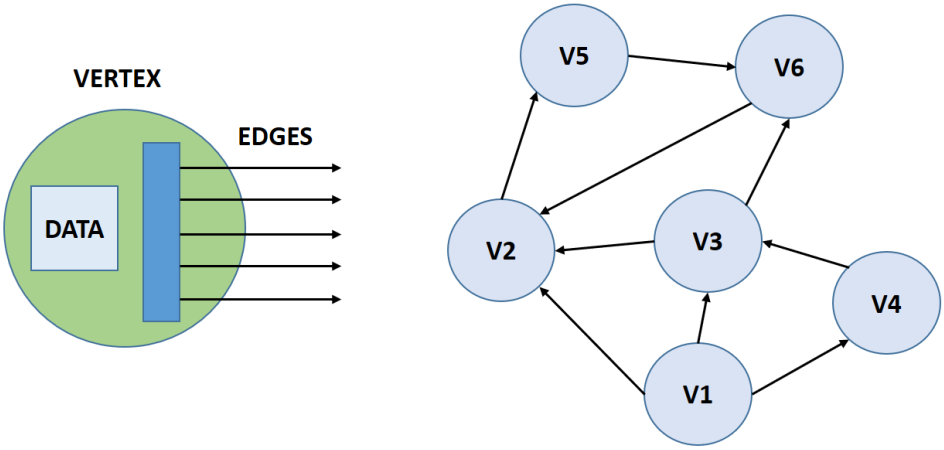

Figura 6.1: Esempio di grafo con nodi dispersi in memoria e di come sia implementato un suo vertice.

La rappresentazione con nodi dispersi, rispetto a una rappresentazione come la matrice di adiacenza, è un approccio particolarmente vantaggioso quando il grafo ha un numero significativo di nodi ma solo una frazione di essi è connessa da archi. Il grafo è rappresentato utilizzando come componente principale una struttura dati che ricopre il ruolo di vertice. Ciascuno dei vertici viene allocato dinamicamente nello heap e collocato su un indirizzo di memoria casuale, deciso dal sistema di allocazione della memoria. Internamente, i vertici racchiudono due componenti principali. La prima componente contiene le informazioni relative al vertice stesso, mentre la seconda è una struttura dati che raccoglie un insieme di archi diretti all'esterno del vertice, puntando agli altri vertici direttamente raggiungibili.
Il modo in cui sono implementate queste due componenti può variare di caso in caso, in base alle esigenze specifiche. Ad esempio, entrambe possono essere allocate internamente al vertice o altrove nello heap. Il set di archi, a sua volta, può essere implementato utilizzando strutture dati differenti, come array di dimensione statica o dinamica, oppure come una lista, o in altri modi ancora.
È stato provato empiricamente che questa prima implementazione del routing

resources graph non sia a adatta per essere usata su algoritmi scritti su GPU. Questa rappresentazione soffre di problemi di performance rispetto ad un'implementazione compatta sequenziale in memoria.
Infatti, le performance delle GPU si deteriorano drasticamente quando accedono a dati che non sono posizionati in modo contiguo in memoria, a causa del numero di letture [25]. Essendo nate per lavorare su array contigui di pixel, le architetture delle GPU sono progettate per sfruttare la località spaziale dei dati, e accedere a dati sparsi in memoria riduce le perfomance poichè non viene sfruttata pienamente la larghezza di banda potenziale della GPU.
In altre parole, mentre il trasferimento di una grande quantità di dati dalla RAM alla VRAM può essere gestito con un numero ridotto di operazioni sfruttando appieno la larghezza di banda della GPU per ogni singolo trasferimento, il trasferimento della stessa quantità di dati sparsi in memoria richiederà tanti trasferimenti RAM-VRAM quante sono le unità di memoria frammentate (in questo caso, pari al numero di vertici del grafo), non potendo così sfruttare appieno la larghezza di banda disponibile.
Per ottimizzare la fase di trasferimento dati da RAM a VRAM, è dunque essenziale implementare una rappresentazione contigua in memoria. La rappresentazione contigua implementata è descritta nella sezione seguente.

6.1.2 Rappresentazione contigua in memoria

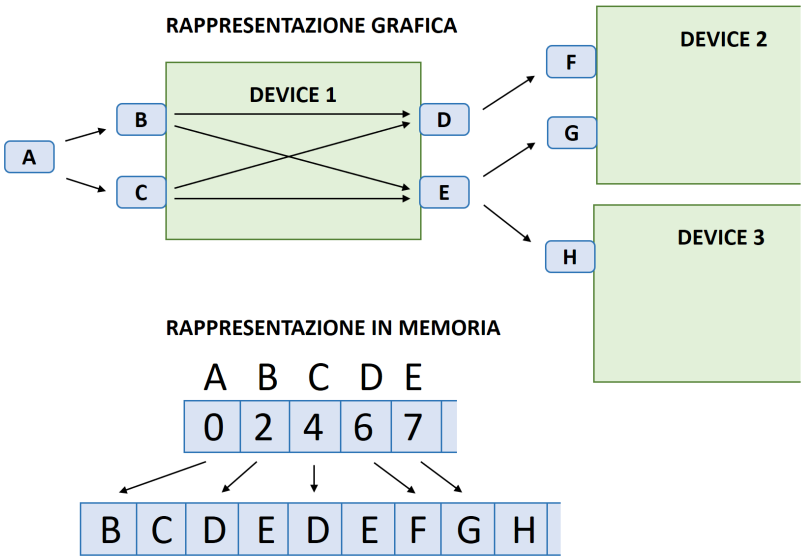

Figura 6.2: Esempio di porzione di grafo implementato sequenzialmente in memoria.

La rappresentazione contigua in memoria implementata per il routing resources graph $G(V, E)$ è illustrata in Fig. 6.2. $G(V, E)$, ed è implementata come una coppia di array globali read-only. Il primo array contiene informazioni su ogni vertice V_i di $G(V, E)$, mentre il secondo array contiene informazioni riguardanti ogni arco E_j. A partire da un indice i è possibile identificare ed accedere direttamente al vertice V_i. Per ciascun pin, l'array di pin contiene un indice da usare sull'array di archi. Tale indice indica la posizione sull'array di archi in cui sono memorizzati gli N_i indici relativi ai pin direttamente raggiungibili da V_i.
In altre parole, i primi N_0 elementi dell'array di archi contengono gli N_0 indici ai pin direttamente raggiungibili da V_0. I successivi N_1 elementi dell'array di archi contengono gli N_1 indici ai pin direttamente raggiungibili dal pin V_1, e così via.
N_i è calcolabile semplicemente come $V_{i+1} - V_i$.
I vantaggi principali nell'adottare tale rappresentazione sono:

- Accesso diretto al pin V_i a partire da un indice i.
- Accesso diretto alla sezione dell'array contenente i pin direttamente raggiungibili da V_i. Questo permette un'esplorazione efficiente di $G(V, E)$ a partire da qualsiasi pin V_i.
- Rappresentazione in memoria compatta. Rispetto alla rappresentazione del grafo con la matrice di adiacenze, questa rappresentazione memorizza solamente gli archi presenti, rendendola adatta a grafi sparsi (i.e. grafi con una quantità di archi molto inferiore al numero di archi che avrebbe un grafo fully connected) come il routing resources graph $G(V, E)$.
- Rappresentazione contigua in memoria. Questa caratteristica è fondamentale quando si impiega una GPU, per i motivi spiegati in precedenza.

Questa è la principale struttura dati coinvolta durante l'esplorazione dei possibili percorsi che connettono una net. Su un grafo di prova con 100 nodi si è dimostrato empiricamente che, qualora si usi una GPU, usare questa implementazione contigua è più veloce di due ordini di grandezza nel trovare il percorso minimo tra due nodi rispetto alla soluzione non contigua in memoria illustrata nella sezione precedente.
Nella nostra implementazione, inizialmente questi due array sono allocati dinamicamente nella memoria RAM. Successivamente, vengono copiati nella memoria GPU (VRAM) per sfruttare la maggiore capacità di elaborazione parallela della GPU durante l'esplorazione dei percorsi. Poiché l'esplorazione delle rotte avviene esclusivamente sulla GPU, l'allocazione di questi due array nella RAM viene immediatamente rilasciata una volta completata la copia nella VRAM, mantenendo solo la copia sulla GPU per le operazioni successive.
Il primo passo che deve fare il router è quindi quello di allocare dinamicamente in

RAM abbastanza spazio contiguo per questi due array, e riempirli con i dati che provengono dal database delle risorse. Tuttavia, non leggiamo i dati direttamente dal database, ma poniamo tra la lettura del database e la costruzione in memoria della struttura dati una fase di preprocessing. La sezione seguente illustrerà i motivi e l'obiettivo di questa fase di preprocessing.

6.2 Database Preprocessing

Attualmente, senza questa fase preliminare, ad ogni avvio il router dovrebbe caricare in RAM le tabelle del database, eseguire una rielaborazione dei dati e generare così la struttura dati per $G(V, E)$. Questo processo è computazionalmente costoso e rallenta significativamente l'inizializzazione del sistema.
Si è perciò anteposto al router una fase di preprocessing del database, che permetta di ottenere un formato che evita di eseguire ogni volta parte della computazione alla creazione della struttura dati. In altre parole, non si fa direttamente una lettura del database seguita dalla costruzione della struttura dati. Invece, si antepone uno script standalone che preprocessa il database in modo da ottenere un formato dei dati più comodo rispetto al formato del database originale.
L'obiettivo è quello di ottenere un formato che permetta di ridurre i tempi di costruzione della struttura dati all'avvio del router. Questa fase di preprocessing non ha bisogno di essere ripetuta più volte. Una volta eseguita, salva su due file distinti il formato modificato, e non c'è più bisogno di leggere il database e ripetere questo processo. Il router leggerà direttamente i dati dal nuovo formato, evitando computazioni dispendiose. I due file generati sono:

- **File relativo all'array dei pin**: Questo file ha il formato:
  `<nome risorsa> <indice all'array di archi> <tipo di risorsa>`
  Il tipo di risorsa servirà all'algoritmo di routing per capire se il pin può essere attraversato o meno dal segnale che sta cercando di instradare. I tipi di risorsa sono: *COMMON*, *LOW_SKEW*, *COMMON_O_LOW_SKEW*, *SCONOSCIUTO*. Un segnale che ricade sotto il tipo *LOW_SKEW* non potrà passare per risorse fisiche non adatte ad inoltrare tale segnale. Potrà attraversare esclusivamente pin del tipo *LOW_SKEW* e *COMMON_O_LOW_SKEW*. Non avendo il datasheet disponibile per l'architettura, il tipo *SCONOSCIUTO* è assegnato a quelle risorse di cui non si può presupporre il tipo.

- **File relativo all'array degli archi**: Questo file ha un formato che rispecchia semplicemente il contenuto dell'array di archi. In ogni riga è salvato un arco nella forma di indice al vettore di pin.

In sintesi, salvare su file l'output del preprocessing del database permette di ottimizzare il tempo di avvio del router, riducendo significativamente la necessità di

computazioni complesse e dispendiose al momento dell'inizializzazione. Grazie a questa fase preliminare, il sistema è in grado di avviarsi e funzionare più velocemente, sfruttando formati di dati preelaborati. All'avvio del router non è nemmeno più necessario allocare abbastanza memoria RAM da contenere le tabelle SQL del database, in quanto questi due nuovi file possono essere letti una riga alla volta per riempire le strutture dati impiegate. Oltre ad avviarsi in tempi più brevi, il router richiede all'avvio anche meno memoria disponibile.

6.3 Algoritmo di Routing

Descritto come viene rappresentata l'architeturra dell'FPGA in modo agevole ad affrontare il problema del routing, nelle sezioni successive saranno invece inizialmente illustrate le strutture dati globali e locali che permettono all'algoritmo di effetturare il processo di routing.
Successivamente, si parlerà brevemente del flow generale dell'intero router.
Infine, avendo il quadro completo delle strutture dati e del flow dell'algoritmo, si parlerà nel dettaglio di come sia stato implementato il routing su GPU e delle sue due varianti a parallelismo a granularità fine e parallelismo a granularità fine e grossolana.

6.3.1 Strutture Dati Globali

In questa sezione parleremo delle strutture dati globali usate per effetturare il routing. Consideriamo come strutture dati globali quelle strutture la cui visibilità spazia sull'intero scope del routing, e che possono essere usate nella costruzione di più net contemporaneamente.

- **Routing Resources Graph:**
  La più importante struttura dati che cade in questa categoria è quella appena presentata, ossia il *routing resources graph*. Questa struttura dati, allocata in VRAM, rappresenta la struttura fisica delle risorse dell'FPGA. E' strutturata in modo da permettere di navigare agevolmente la rete di vertici del grafo. E' usata in sola lettura, in quanto le risorse fisiche, l'architettura e la disposizione delle interconnessioni non è soggetta a cambiamenti.

- **Global Labeling:**
  Un'altra struttura dati fondamentale è l'array noto come *global labeling*. Dopo aver individuato il percorso ottimale tra il pin sorgente e il pin target della net, è necessario marcare tale percorso come utilizzato, rimuovendo quindi le risorse corrispondenti dal pool di risorse ancora disponibili per nuove rotte. Il

global labeling è l'array che si occupa di memorizzare quali vertici sono liberi e quali sono occupati da altri segnali. Questo array, oltre a marcare le risorse, indica specificamente quale segnale sta occupando ciascun vertice in base al valore del marcatore.
L'array viene allocato direttamente nella memoria VRAM e ogni suo elemento viene inizializzato per indicare che la risorsa non è ancora stata assegnata.

- **Tipo Nodo:**
  La struttura dati *Tipo Nodo* è un array da cui è possibile ricavare in modo diretto per un qualsiasi vertice i a che tipo appartiene tra *COMMON*, *LOW_SKEW*, *COMMON_O_LOW_SKEW*o *SCONOSCIUTO*. Il suo obiettivo è quello di indicare di che tipo siano i nodi trovati durante l'esplorazione su GPU dei nodi raggiungibili. Questo è importante per escludere le strade che passano per vertici di tipo non compatibile col tipo di segnale da instradare. Questo array è popolato durante la lettura del file, quando si costruisce il routing resouces graph. Analogamente al routing resources graph, esso viene inizialmente allocato e popolato in RAM, per poi essere copiato su VRAM. Dopo la copia, è possibile deallocare l'array dalla RAM per risparmiare memoria.

- **Nome Nodi**
  Infine, di importanza secondaria sono le strutture dati usate per la conversione dal nome del vertice all'indice corrispondente sull'array di pin, e viceversa. La conversione da indice al nome del vertice viene implementata come un array di String (conversione utile solamente ai fini della stampa dei nomi dei vertici di una route). La conversione inversa, ossia dal nome all'indice, viene implementata con una mappa da String ad Int. Questa conversione è necessaria poichè l'algoritmo di routing lavora identificando i vertici con il loro indice, e convertire da nome ad indice serve per tradurre una net inserita da tastiera o da file in una coppia di indici di start ed end usabili dall'algoritmo.

6.3.2 Strutture Dati Locali

Nella Sezione 6.3.1 abbiamo mostrato le principali strutture dati globali utilizzate. Durante il routing di una net, abbiamo bisogno di alcune strutture dati aggiuntive oltre a quelle globali descritte nella sezione precedente. La differenza tra le strutture dati che definiamo come globali e quelle che presenteremo ora è che, quando instraderemo N net contemporaneamente, le strutture dati che introdurremo ora dovranno essere allocate N volte e sono locali al routing della singola net.
Queste strutture dati locali tengono traccia dell'esplorazione corrente delle risorse disponibili e cooperano alla costruzione di una singola net. Invece, quelle globali non hanno bisogno di essere replicate più volte in memoria e l'accesso a queste può

essere condiviso tra più ricerche contemporaneamente.
Tutte le strutture dati locali qui presentate sono implementate come array, per gli stessi motivi per cui quelle globali sono implementate come array, ossia principalmente per l'accesso diretto agli elementi e una disposizione contigua in memoria:

- **Local Labeling**
  Un array che, durante l'esplorazione delle risorse disponibili, segna i vertici trovati per la prima volta e tiene traccia di quelli già esplorati nelle iterazioni precedenti della corrente esplorazione.

- **Previous-Next Buffers**
  I buffer Previous and next sono un doppio buffer. Uno contiene gli indici dei nuovi nodi scoperti nell'iterazione precedente, mentre l'altro gli indici dei nodi scoperti nell'iterazione corrente. Ad ogni nuova iterazione, i ruoli dei due buffer si invertono.

- **Backtracking**
  Un array popolato durante l'esplorazione delle risorse disponibili. Quando esploriamo un nuovo vertice, dobbiamo salvarci per quel vertice da quale altro vertice lo abbiamo raggiunto. L'array Backtracking salva, per ogni vertice scoperto, da quale nodo lo si è scoperto. Quando infine si scopre il nodo destinazione, queste informazioni sono essenziali per ricorstruire il path intrapreso ripercorrendo a ritroso il percorso effettuato.

6.3.3 Panoramica dell'Algoritmo

L'algoritmo di routing è suddiviso in due fasi distinte:

- Preparazione al routing
- Routing sulle net di input

Nella prima fase, in preparazione al routing, l'algoritmo alloca e popola le strutture dati precedentemente descritte. Vengono letti i file descritti nella Sezione 6.2 e vengono allocati in RAM l'array dei pin, l'array degli archi, l'array dei tipi di risorse e le strutture dati per la conversione da indice a nome e da nome a indice. Dopo aver allocato queste strutture in RAM, l'array dei pin, l'array degli archi e l'array dei tipi di risorse vengono copiati in VRAM. Una volta trasferiti in VRAM, questi array possono essere deallocati dalla RAM. Le strutture dati per la conversione, invece, non vengono copiate in VRAM e rimangono in RAM.
Conclusa la prima fase, si può passare al processo di routing. Sono state implementate alcune alternative:

- Routing di net inserite da tastiera
- Routing di net inserite da file
  - Con tecnica di parallelismo a granularità fine
  - Con tecnica di parallelismo a granularità fine e grossolana

La nostra implementazione permette di decidere se inserire le net interattivamente da tastiera, utile soprattutto in fase di debugging, oppure se leggere l'output prodotto dallo script di parser e placement. Tuttavia, indipendentemente se si inserisce l'input da tastiera o se viene preso dall'output del placement, l'algoritmo alla base della ricerca delle route rimane il medesimo.
Qualora le net fossero inserite da tastiera, il router chiede il nome di due risorse per formare una net. I due nomi sono tradotti in indici con un costo unitario O(1). A partire dagli indici, viene eseguito l'algoritmo di routing descritto nella Sezione 6.3.4. Il risultato dell'algoritmo restituisce la route generata per questa net. Il processo si ripete iterativamente chiedendo il nome di altre due risorse per comporre un'altra net.
Nel caso si scelga invece di inserire le net da file, bisogna seguire il formato dell'output del parser e placement (Si rimanda alla Sezione 5.2 per i dettagli sul parser e placement e di come sia formattato il suo output).
Mentre per le singole net inserite da tastiera è possibile usare una sola variante (quella a granularità fine), per un gruppo di net è possibile usare sia la variante a granularità fine della Sezione 6.3.4, sia la variante a granularità fine e grossolana, descritta nella Sezione 6.3.5. La differenza principale è che nel secondo approccio è possibile costruire più net contemporaneamente, mentre nel primo la costruzione delle net resta sequenziale.
In entrambe le varianti a granularità fine e a granularità fine e grossolana si può abilitare o disabilitare la funzionalità che permette di eliminare dal pool di risorse disponibili quelle appena utilizzate dalle net instradate. In altre parole, nel caso disabilitassimo questa funzionalità, si costruiscono le varie route ignorando la congestione e restituendo il risultato come se ogni risorsa fisica fosse sempre disponibile. Nel caso che invece la si abilitasse, trovare una route rimuoverebbe per le net successive le risorse assegnate alla route generata.
Se la funzione di rimozione delle risorse disponibili dal pool è attiva, si inizia con il rimuovere dal pool i vertici che corrisponono ai pin indicati nei file di entity input, entity output e segnali costanti. Questi vertici devono essere rimossi ancor prima di effettuare il routing, in quanto sono riservati per segnali costanti o per segnali di input o output dell'entità da implementare su FPGA.
Successivamente, sono instradati i segnali low skew in modo sequenziale. Qui è dove le due tecniche divergono algoritmicamente. Nella variante a granularità fine si instraderanno in modo sequenziale le net restanti, ossia quelle relative ai

segnali common. Invece, nella variante a granularità fine e grossolana, queste net sono suddivise in N+1 set distinti, in base alle caratteristiche delle net. N di questi set contengono net che hanno la coppie risorsa di start e risorsa target che cadono nella stessa Tile, mentre l'ultimo set contiene net che hanno start ed end che ricadono in Tile differenti. Questo permette di effettuare il routing degli N set in modo concorrente senza che si debbano sincronizzare per le risorse, in quanto il path seguito rimarrà all'interno della Tile, senza passare per le Mesh. Si rimanda alla Sezione 4.3 relativa al percorso che effettuano le route nell'architettura di NG-Medium e NG-Ultra. L'ultimo set invece non può essere instradato in modo concorrente agli altri N set, in quanto potenzialmente potrebbero presentarsi conflitti di risorse durante l'esplorazione del grafo. Le net che ricadono in questo set possono essere instradate prima o dopo le altre N net. L'importante per garantire la correttezza dell'algoritmo è che siano instradate in modo sequenziale rispetto le net degli altri set. Inoltre, tra le net dello stesso set, non è possibile instradare due net in modo concorrente per lo stesso motivo del conflitto di risorse. In altre parole, è possibile al massimo instradare contemporaneamente N net, ognuna delle quali deve appartenere ad un set diverso. Non possono essere instradate assieme due net dello stesso set, e le net dell'ultimo set non possono essere instradate in parallelo con alcun altra net.

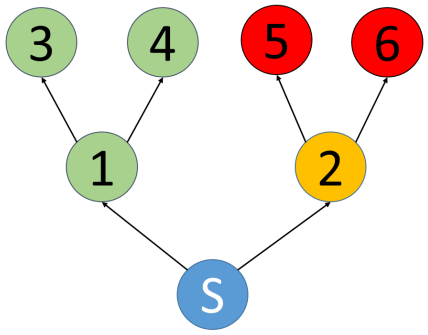

Figura 6.3: Invalidazione dei vertici che discendono dal vertice 2 dopo che tale vertice è stato rimosso dalle risorse disponibili

Il problema che si presenta quando questo vincolo non è rispettato, è che quando si instradano diverse net in parallelo e una di queste trova il path, questa deve rimuovere le risorse usate dal pool di risorse disponibili. Dopo aver marcato queste risorse come non più disponibili, le altre net che stanno calcolando la propria route in parallelo devono fare pruning di tutti i vertici scoperti a partire da una delle risorse rimosse. Questo perchè si percorrerebbe un path non più percorribile. Fig. 6.3 mostra graficamente il problema appena descritto.
Per evitare questo problema si è appunto diviso le net in N+1 gruppi, di cui N

compatibili per essere instradati contemporaneamente e un set che deve essere instradato in modo sequenziale.

6.3.4 Parallelismo a Granularità Fine

In questa sezione illustreremo la parte principale dell'algoritmo, ossia come sono generate e scelte le route per le net. L'algoritmo si ispira al Breadth-First Search (BFS), la ricerca in ampiezza su grafi a partire da un vertice.

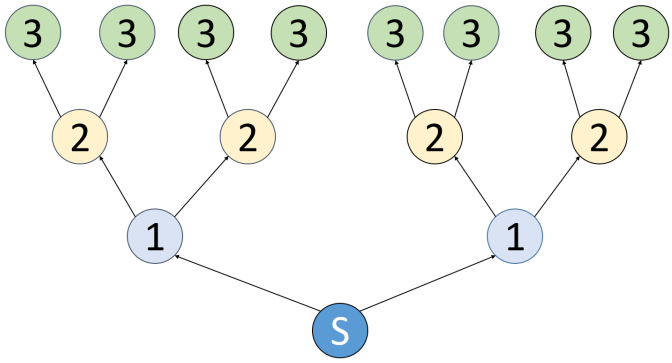

Figura 6.4: Ricerca in ampiezza (BFS). In figura, i vertici sono numerati secondo la loro distanza dal vertice sorgente S. La ricerca avviene esplorando i nodi in ordine di distanza crescente.

Il BFS è un algoritmo di ricerca che esplora i vertici di un grafo o un albero in modo sistematico e completo, livello per livello. Partendo da un nodo sorgente, il BFS visita tutti i suoi nodi adiacenti prima di procedere ai nodi di livello successivo. Questo approccio garantisce che tutti i nodi a una certa distanza dal nodo iniziale siano visitati prima di passare ai nodi più lontani. Questa caratteristica rende il BFS uno degli algoritmi più usati per la ricerca dei cammini minimi, ossia per la ricerca della sequenza più corta di vertici che consente di raggiungere un vertice destinazione.
Tipicamente, il BFS è un algoritmo implementato sequenzialmente. Questo vuol dire che i nodi disposti alla stessa distanza N dal vertice sorgente S sono esplorati iterativamente uno alla volta. Una volta esplorati sequenzialmente tutti i vertici a distanza N da S, si passa ai vertici a distanza N+1. Questo approccio sequenziale al BFS ha complessità algoritmica O(V), dove V rappresenta il numero dei vertici del grafo.
Per implementare questo router ci si è ispirati a questo algoritmo, cercando però di accellerarlo tramite GPU. L'accellerazione tramite GPU degli algoritmi sequenziali avviene sfruttando il potenziale di parallelizzazione del problema da affrontare. Nel

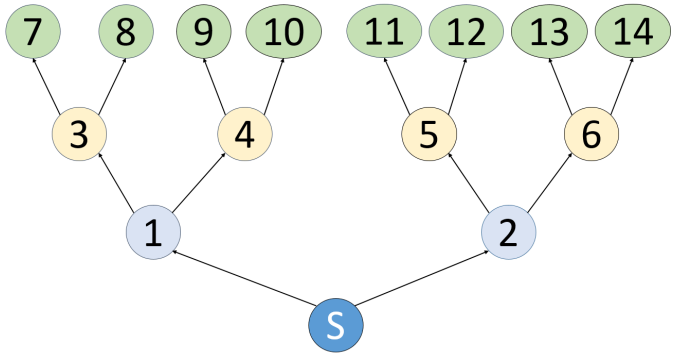

Figura 6.5: Ricerca in ampiezza (BFS) sequenziale. I vertici sono qui numerati secondo l'ordine di esplorazione. Se su questo grafo si ricerca un nodo a distanza 3 rispetto ad S, sono necessari dai 7 ai 14 passi.

caso del BFS, è possibile costruirne una versione concorrente che riduce la complessità algoritmica O(V) a O(N), dove N è la profondità massima del grafo a partire da un nodo sorgente S. I due approcci si equivalgono solo nel caso peggiore, ossia che il grafo disponga i suoi vertici in una struttra a lista, facendo così corrispondere il numero di vertici alla profondità massima del grafo. Tuttavia, per definizione, le architetture di NG-Medium e NG-Ultra sono ben lontane dal essere rappresentate da un grafo simile. Nel caso medio, la profondità N è di molto inferiore al numero di vertici V, anche di diversi ordini di grandezza. All'atto pratico però, si potrebbe non raggiunge una complessità O(N) nel caso che non ci fossero abbastanza risorse fisiche per computare in parallelo un intero livello del grafo.
Vediamo ora più nel dettaglio come sia stata realizzata la variante su GPU del BFS. Questa variante richiede come input gli indici del vertice sorgente e del vertice destinazione della net. Inizialmente, come preparazione all'esplorazione delle risorse, si allocano e inizializzano su GPU le strutture dati locali descritte nella Sezione 6.3.2, ossia gli array Local Labeling, Backtracking e il doppio buffer.
Local Labeling ha ogni suo elemento inizializzato al contenuto di Global Labeling. Questo permette di esplorare il grafo passando solamente per le risorse attualmente disponibili. Non si usa direttamente l'array Global Labeling per marcare l'esplorazione perchè in questo modo è possibile effettuare più esplorazioni in contemporanea su array Local Labeling differenti.
Il doppio buffer, che ricordiamo essere una coppia di array chiamati Previous e Next, è inizializzato per contenere nell'array Previous solamente il vertice sorgente, e nell'array Next nessun vertice. Questa configurazione di partenza simula un iterazione precedente in cui è stato scoperto il vertice sorgente, consentendo alla prima iterazione di esplorare i nodi a distanza 1 da esso.
L'array Backtracking, che serve ad indicare per ciascun vertice quale altro vertice

lo abbia scoperto, è invece inizializzato in modo che ogni elemento indichi che i vertici non sono stati ancora scoperti e quindi non esiste ancora alcun nodo che lo abbia scoperto.
Dopo aver finito le inizializzazioni, si adotta un approccio iterativo per trovare il path tra vertice sorgente e vertice destinazione della net. Il criterio utilizzato per determinare il percorso migliore è quello del cammino minimo, ossia si seleziona il percorso che attraversa il minor numero di risorse fisiche. Tuttavia, questo criterio, è altamente probabile che non sia ottimale per un algoritmo di routing complesso. Si devono infatti rispettare una serie di vincoli per ogni segnale instradato, tra cui latenze massime, la lunghezza del cammino critico, la capacità delle risorse, l'interferenza tra segnali, la tolleranza ai guasti e altri requisiti di qualità del servizio (QoS). Tuttavia, a nostra disposizione abbiamo solamente la topologia dell'architettura, senza conoscere dettagli come le latenze delle interconnessioni. Si procede dunque ad implementare il criterio del cammino minimo, presupponendo che ogni arco che collega due risorse abbia latenza unitaria.

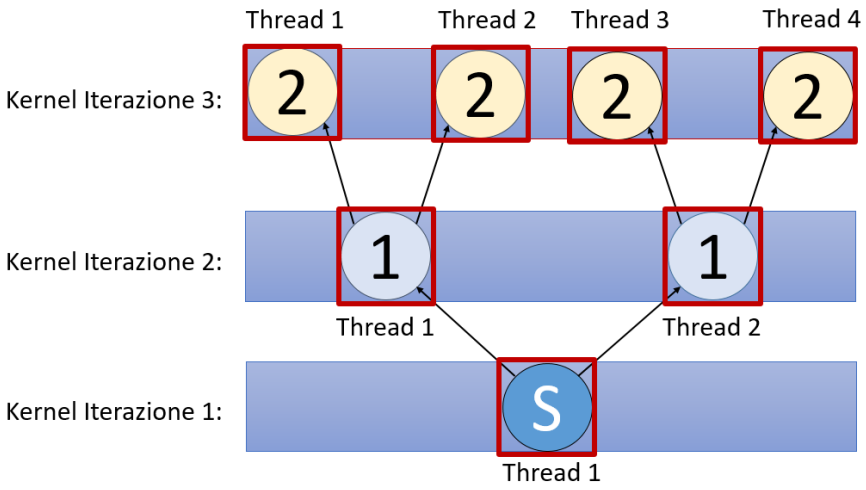

Figura 6.6: Rappresentazione grafica dei Kernel e Thread eseguiti nelle prime tre iterazioni di un BFS implementato su GPU.

L'approccio iterativo si interrompe qualora si trovi il vertice destinazione o non ci siano più nuovi nodi raggiungibili da esplorare. Ad ogni iterazione, viene lanciato un kernel CUDA per parallelizzare l'elaborazione. Figura 6.6 mostra graficamente i kernel lanciati nelle prime tre iterazioni dell'esplorazione di un grafo.
Ogni kernel genera abbastanza thread per poter gestire il numero di vertici scoperti all'interazione precedente. Tali vertici sono memorizzati nel buffer Previous del

doppio buffer. Nella nostra implementazione, sono eseguiti kernel con una dimensione di 128 thread per blocco. Un'iterazione che deve gestire 50 nodi scoperti allocherà un blocco di threads da 128, di cui solo 50 effettueranno una computazione. Un'iterazione che ne deve gestire 200 allocherà due blocchi di thread da 128 l'uno. Ognuno dei thread GPU che gestisce un vertice scoperto ha il compito di iterare i suoi archi ed inserire i vertici direttamente raggiungibili all'interno del buffer Next. Saranno inseriti solamente i vertici che rispettano determinate caratteristiche. I criteri sono i seguenti:

- **Vertici scoperti per la prima volta:**
  Un vertice raggiungibile è valido se non è stato ancora scoperto nelle iterazioni precedenti. Questo è effettuato controllando l'array Local Labeling.

- **Vertici occupati dallo stesso segnale:**
  Un vertice raggiungibile è considerato valido se è stato occupato da un'altra net che ha instradato lo stesso segnale. Non è invece valido se è occupato da una net relativa ad un altro segnale.
  I segnali propagati nell'FPGA partono da un vertice e ne possono raggiungerne molti. Questo forma, per ogni segnale, un albero di vertici. Le net che noi consideriamo sono invece punto-punto. L'insieme di net che formano un segnale partono dallo stesso vertice sorgente e terminano su vertici destinazione differenti. Durante il routing è permesso che le net dello stesso segnale attraversino le stesse risorse, perchè non ci sarebbero inconsistenze (ossia un pin assegnato a due segnali diversi). Anche questo criterio è implemento con l'ausilio del Local Labeling, che non solo indica se i vertici sono stati usati, ma registra anche l'ID di segnali che li occupano.

- **Tipo di Vertice:**
  Un vertice raggiungibile è considerato valido se il suo tipo è adatto al tipo di segnale che stiamo instradando.
  Se ad esempio il segnale relativo alla net è un segnale low-skew, saranno inseriti solamente i vertici di tipo LOW_SKEW e COMMON_O_LOW_SKEW nell'array Next, mentre verrano scartati i pin di tipo COMMON perchè vertici non adatti ad instradare tale segnale. Viceversa, se il segnale è di tipo common, saranno inseriti solamente i vertici di tipo COMMON e COMMON_O_LOW_SKEW, e scartati quelli di tipo LOW_SKEW.

Avendo molti thread che devono eseguire diverse scritture sul buffer Next, le scritture devono essere gestite in modo tale da evitare problemi derivanti dal multithreading. Questo viene gestito facendo uso di operazioni atomiche affinché ognuno degli N thread abbia un indice univoco su cui scrivere nel buffer Next, evitando perciò ogni tipo di accesso in scrittura concorrente e garantendo la coerenza dei dati. Dopo ogni scrittura sull'array Next, il thread scrive sul array Backtracking e marca su

Local Labeling il vertice come esplorato.
Terminato il kernel, il buffer Next risulta popolato dai vertici che dovranno essere esplorati nella iterazione successiva (da qui il nome Next). In preparazione all'iterazione successiva, si dovrebbe configurare il buffer Previous con il contenuto del buffer Next e andrebbe suotato il contenuto di Next. Per evitare copie e cancellazioni, che sarebbero dispendiose dal punto di vista computazionale, si realizza una tecnica nota come ping-pong buffering [26].
Questa tecnica consiste nello scambiare i puntatori, e quindi il ruolo, dei due buffer ad ogni iterazione. In questo modo, il buffer che prima era invocato tramite l'etichetta Next viene ora raggiunto con l'etichetta Previous e viceversa. Questo consente di avere in Previus il contenuto che aveva Next senza eseguire copie di dati. Quel che ora rimane da risolvere è lo svuotamento del contenuto di quello che ora assume il ruolo di buffer Next. Per farlo senza alcuna cancellazione o inizializzazione di dati, si imposta a zero una variabile che indica il numero di elementi validi all'interno del buffer. Questa tecnica consente di risparmiare ad ogni nuova iterazione operazioni che penalizzerebbero le prestazioni.

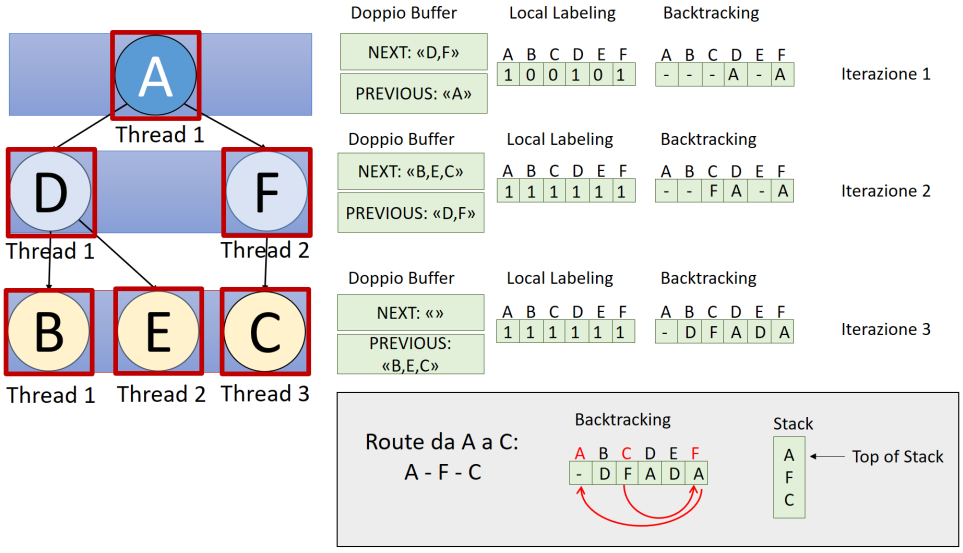

Figura 6.7: Illustrazione grafica dell'algoritmo.

Infine, prima di iniziare una nuova iterazione, si controlla la condizione di terminazione. Se il buffer Previous è vuoto abbiamo terminato l'esplorazione del grafo e non è possibile connettere i due vertici tramite un path. Questa è una condizione che si verifica in caso di congestione, ossia quando sono state instradate troppe net, oppure sono state instradate in modo da rendere il grafo delle risorse disponibile non più connesso.

L'altra condizione di terminazione è invece il caso in cui tra i vertici Previous troviamo il vertice destinazione della net. In questo caso, l'algoritmo garantisce per costruzione che è stato raggiunto con il numero minimo di passi. Questo controllo non è fatto scansionando il buffer Previous, che richiederebbe una complessità computazionale O(N) con N numero di nodi scoperti, ma con una complessità computazionele unitaria O(1) si controlla se l'array Local Labeling sia stato marcato all'indice corrispondente al vertice destinazione.
Nel caso trovassimo la destinazione, si procede col marcare su Global Labeling i vertici presi dal cammino minimo in modo da rimuovere tali risorse dal pool di risorse disponibili per le prossime net. Si conclude infine ricavando il cammino minimo a partire dall'array Backtracking.
L'array Backtracking è semplicemente percorso a ritroso a partire dall'indice che ne gestisce il vertice destinazione. Si seguono così iterativamente gli indici fino a fermarci quando troviamo l'indice relativo al vertice sorgente.
Nel caso esistessoro più cammini minimi differenti, per come è stato costruito l'algoritmo, l'array Backtrack ne registra solo uno.
Percorrendo in questo modo Backtracking, viene riempito uno stack di indici di nodi che rappresenta il percorso. In cima allo stack troviamo il nodo sorgente, e in fondo il nodo destinazione. Questo stack viene ritornato dalla funzione, e può essere gestito arbitrariamente, ad esempio per effettuare una stampa del percorso o per salvarlo.
Questa tecnica appena descritta sfrutta un approccio iterativo che fa uso della computazione parallela per esplorare ogni via possibile contemporaneamente sul grafo. Definiamo questa tecnica come *parallelismo a granularità fine* poichè a questa affiancheremo un ulteriore tecnica di parallelismo a livello più alto, che definiamo *parallelismo a granularità grossolana.*

```
1  Input: vertice sorgente S, vertice destinazione D
2
3  init Local Labeling
4  init Double Buffer
5  init Backtracking
6  while (D non trovato) && (esplorazione non terminata)
7      esecuzione del kernel (GPU)
8      swap puntatori Double Buffer
9      clean array Next
10     if D trovato
11         marking dei nodi su Global Labeling (rimozione risorse)
12         creazione dello stack di nodi a partire da Backtracking
13         return stack
14 return avviso di congestione
```

6.3.5 Parallelismo a Granularità Fine e Grossolana

La seconda tecnica di parallelismo che andremo ad implementare la definiamo *a granularità grossolana*. Questa tecnica può essere usata assieme alla tecnica descritta nella sezione precedente per ottenere una soluzione che sfrutta parallelismo a granularità fine e grossolana.
L'idea alla base della tecnica a granularità grossolana nasce osservando come la tecnica a granularità fine continui a creare route sequenzialmente, nonostante velocizzi il processo parallelizzando la ricerca della singola route. La domanda che ci poniamo per incrementare ulteriormente le perfomance è quindi: è possibile sfruttare una percentuale più alta delle risorse della GPU parallelizzando anche la generazione sequenziale delle route?
Il problema principale che si incontra quando si vogliono instradare più net contemporaneamente è che quando una route viene generata questa rimuove dal pool di risorse fisiche disponibili un certo numero di vertici del grafo. Questo diventa un problema per le altre net che stanno esplorando il grafo di risorse fisiche disponibili, come già mostrato in Figura 6.3. Queste net dovrebbero eliminare dall'array Previous qualunque vertice che discenda dall'esplorazione di uno dei vertici rimossi. Questo può essere fatto consultando strutture dati che tengono traccia, per ogni vertice scoperto, la storia di tali vertici (l'array Backtracking, nella nostra soluzione). Ma consultare tali strutture dati comporterebbe perdite in performance, in quanto per ognuno dei vertici di ogni altra net concorrente andrebbe ricercato se deriva da almeno uno dei nodi rimossi dal pool. Questo controllo causerebbe una perdita di performance significativa, proporzionale all'aumentare delle route computate in parallelo e al numero di nodi scoperti. Nel peggiore dei casi, l'algoritmo passerrebbe più tempo ad invalidare nodi rispetto che a cercare le route.
Per evitare questo controllo che degrada le prestazioni, è stato scelto un approccio alternativo, cioè quello di parallelizzare solamente la creazione di route nel caso abbiamo la certezza che le risorse rimosse appartengano a set disgiunti. Questa alternativa pone invece un limite al grado di parallelizzazione massimo, in quanto non tutte le net sono ora adatte ad essere instradate simultaneamente, ma devono rispettare un vincolo. Questa limitazione potrebbe essere eliminata cambiando approccio riguardo la rimozione delle risorse, non rimuovendo più i vertici presi dal grafo e accettando la congestione, per poi risolvere la congestione a posteriori reinstradando iterativamente le route fino a raggiungere una configurazione senza congestione. Questo approccio è implementato in algoritmi come il Path Finder Algorithm. Tuttavia si è deciso di non seguire questa strada e di tentare di implementare la prima alternativa proposta.
L'approccio che abbiamo implementato sfrutta le informazioni architetturali delle FPGA per suddividere le reti in set distinti. Ogni set è formato in modo tale che le reti al suo interno possono rimuovere vertici da un pool di risorse fisiche diverso

da quello utilizzato da qualsiasi altro set. Questo consente di instradare contemporaneamente le reti provenienti da set differenti. Tuttavia, le reti appartenenti allo stesso set non possono essere instradate simultaneamente, poiché condividono la stessa porzione geografica dell'FPGA e utilizzano lo stesso pool di risorse fisiche.
Il criterio che viene usato per creare i set trae ispirazione dai percorsi che le route effettuano. Nella Figura 4.6 si nota come le net composte da pin di inizio e destinazione che ricadono nella stessa Tile formino un percorso che non passa per le Mesh, garantendo che tale percorso utilizzi solo vertici del grafo appartenenti alla Tile stessa. Suddividere le net secondo questo criterio produrrà N set composti da net con pin di inizio e destinazione che cadono all'interno della stessa Tile, per N Tile differenti. Inoltre, tutte le net restanti, ossia quelle che sono formate da pin di inizio e destinazione non appartenenti alla stessa Tile, sono inserite in un ulteriore set con la caratteristica di avere net non instradabili in parallelo e che dovranno essere instradate in modo sequenziale. Questo accade perché le route di tali net attraversano le Mesh. Pertanto, non è possibile garantire l'assenza di conflitti geografici nel momento in cui queste route rimuovono i vertici presi dal grafo.
In altre parole, prima di iniziare col routing si suddividono le net in N+1 set secondo il criterio geografico appena descritto. N può variare e dipende dalla fase di placement, e sarà sempre minore o uguale al numero di Tile dell'FPGA. N è molto probabile che sia inferiore al numero fisico di Tile disponibili perchè il placement, per la maggior parte dei design, produrrà configurazioni che non usano tutte le Tile disponibili sull'FPGA.
Analizziamo ora più nel dettaglio l'algoritmo di instradamento simultaneo degli N set parallelizzabili. Come si può vedere dallo pseudocodice, l'approccio è molto simile a quello usato nella tecnica a granularità fine. La differenza principale risiede nell'uso degli stream Cuda e nell'allocazione delle strutture dati locali (Sezione 6.3.2) per ognuno degli N set.
Ad ogni iterazione, ogni set che non ha ancora finito di instradare tutte le proprie net esegue un kernel per computare uno step di BFS sulla net su cui sta lavorando. Ogni kernel è eseguito sul proprio stream Cuda in modo asincrono, permettendo a questi N step di avvenire contemporaneamente. La performance risultano migliori proprio per la parallelizzazione di questi N step di BFS.
Successivamente, tramite una primitiva di sincronizzazione, si aspetta che ogni set completi il proprio step di BFS, per poi controllare sequenzialmente se la route sia stata trovata. Nel caso affermativo, si estrae dal set la net e si passa alla successiva, reinizializzando nuovamente le strutture dati locali. Nel caso negativo, alla prossima iterazione si continuerà a cercare la route eseguendo un nuovo passo sul BFS.

```
Input: N set di net

init N array Local Labeling
init N Double Buffer
init N array Backtracking
init N cuda streams
while (rimane almeno una net all'interno di uno degli N set)
    for (ogni set con ancora net da instradare)
        esecuzione async (cuda stream) del kernel (GPU)

    sincronizzazione dei kernel
    for (ogni set con ancora net da instradare)
        swap puntatori Double Buffer
        clean array Next
        if D trovato
            marking dei nodi su Global Labeling (rimozione risorse)
            ottenimento della route
            init delle strutture dati locali Ni
            pop della net completata dal rispettivo set

Routing del set non parallelizzabile N+1 con la tecnica fine-grained
```

Le iterazioni continuano fino a che esiste almeno una net ancora da routare. Quando queste finiscono, si può passare ad instradare le net del set N+1 non parallelizzabile. Questa operazione può anche avvenire prima. La cosa importante è che avvenga in modo sequenziale riespetto gli altri N set.

Capitolo 7

Risultati

7.1 Performance e uso della memoria

Questa sezione è dedicata ai risultati misurati riguardanti le varie tecniche GPU di parallelizzazione del router. Sono stati testati quattro benchmark provenienti dalla suite di benchmark ITC'99 [23]. Questi benchmark sono stati scelti per la loro architettura ben nota, il comportamento prevedibile e per coprire un'ampia sezione dello spettro della complessità del routing. Nella Tabella 7.1, per ciascun benchmark è mostrata la quantità di net da instradare, premettendoci di avere un indicazione della dimensione del design del benchmark specifico.

Benchmark	**Numero di Nets** [#]
B03	253
B06	68
B09	284
B12	1688

Tabella 7.1: Numero di net da instradare per ogni benchmark.

Per quantificare i benefici che il nostro router può introdurre, per diverse configurazioni sono stati calcolati i tempi di instradamento di ciascun benchmark. La Tabella 7.2 mostra un confronto tra le prestazioni di un router che fa uso di un approccio totalmente sequenziale su CPU basato sul BFS, con questo router che implementa solo la tecnica fine-grained, e che implementa sia le tecniche fine-grained che coarse-grained (con quattro set di net).
Non è tuttavia stato possibile confrontare direttamente le prestazioni del nostro router con quelle del router all'interno della toolchain NanoXplore, a causa del suo algoritmo che mescola la fase di routing con quella di placement (problema esposto

nel dettaglio nella Sezione 5.1).

Algoritmi	**B03**	**B06**	**B09**	**B12**
Sequential router	143.17	62.73	300.05	433.14
GPU fine-grained router	0.81	0.47	0.89	4.62
GPU fine-coarse-grained router	0.60	0.42	0.58	2.64

Tabella 7.2: Confronto delle prestazioni tra diverse configurazioni di router.

I risultati mostrano performance migliori per la configurazione GPU a granularità fine e grossolana rispetto alle altre. Tuttavia, i tempi sembrano comunque scalare in modo proporzionale in base al numero di net da instradare. Questo è dovuto all'impossibilità di emettere un numero di kernel pari alla quantità di net, raggiungendo così una parallelizzazione massiccia del problema.

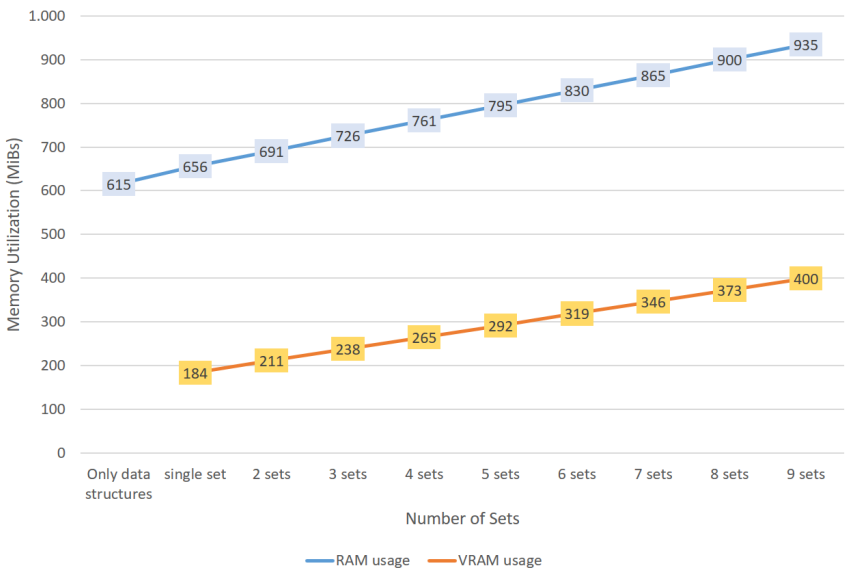

Figura 7.1: Consumo di memoria con l'aumentare dei set della tecnica a granularità grossolana.

Fig. 7.1 mostra invece l'utilizzo della memoria RAM e GPU per il router GPU a granularità fine e grossolana. L'analisi dell'utilizzo della RAM indica che il grafo delle risorse di instradamento per l'FPGA NG-Medium consuma 615 MiB, basato

sulla rappresentazione descritta in precedenza. Inoltre, per ogni set di net aggiuntivo utilizzato nella tecnica coarse-grained, il nostro algoritmo consuma ulteriori 35 MiB di RAM in strutture dati. Per quanto riguarda la VRAM, il calcolo richiede un totale di 184 MiB per instradare un singolo set di net, e approssimativamente ulteriori 25 MiB per ciascun altro set.
In conclusione, i risultati mostrano che sfruttare massivamente la programmazione concorrente tramite GPU è un modo promettente per migliorare le prestazioni laddove tale strategia sia implementabile. Crediamo che sfruttare ulteriormente il parallelismo, ad esempio con la tecnica del parallelismo dinamico, sia la strada giusta da seguire per migliorare ulteriormente le performance.

Capitolo 8

Lavori Futuri

8.1 Cambiamenti algoritmici

Per quanto riguarda i lavori futuri, sarà essenziale affrontare principalmente i problemi di congestione e scelta del percorso ottimale, che comporteranno alcuni cambiamenti nell'algoritmo.

8.1.1 Congestione

La gestione della congestione è cruciale per garantire l'efficienza e l'implementabilità del design e per prevenire configurazioni che possono compromettere le prestazioni complessive del sistema. Un possibile approccio è l'implementazione di un approccio iterativo simile a quello implementato dal Path Finder Algorithm, che consiste nell' iterare le configurazioni trovate fino al raggiungimento di una configurazione senza congestione.

8.1.2 Criterio di scelta del percorso ottimale

Al momento, nella ricerca del percorso abbiamo assunto che il percorso migliore fosse quello più breve in termini di numero minimo di risorse attraversate. Nonostante attraversando meno risorse si tende in generale ad ottenere percorsi con latenze più basse, trovare il percorso che attraversa meno risorse non garantisce che questo equivalga al percorso con latenza minima. Questo è vero solo nel caso che le interconnessioni tra risorse abbiano tutte la stessa latenza. In realtà, ogni interconnessione ha una sua latenza caratteristica, e per ciascun segnale andrebbe scelto un percorso idoneo alla latenza massima accettabile per tale segnale.
Attualmente non siamo in possesso delle informazioni relative alla latenza delle interconnessioni, che risultano essere informazioni architetturali riservate, costringendoci perciò ad adottare questo criterio. Nel caso ottenessimo in futuro tali

informazioni, la condizione di terminazione dell'algoritmo andrebbe modificata in modo da tener conto, per ogni percorso, la sua latenza cumulativa.
Questo vorrebbe dire modificare la variante GPU del BFS da noi implementata in una variante dell'algoritmo di Dijkstra su GPU.

8.2 Miglioramenti alle prestazioni

In termini di prestazioni, sono disponibili diverse strade da esplorare per migliorare ulteriormente il router GPU. Di seguito sono proposte alcune idee che migliorerebbero ulteriormente le performance dell'algoritmo.

8.2.1 Parallelismo dinamico

Nel nostro router, ad ogni iterazione viene lanciato un kernel GPU con un numero di thread Cuda pari al numero di nuovi vertici scoperti nell'iterazione precedente durante l'esplorazione del grafo.
Ogni thread Cuda gestisce uno di questi vertici e iterativamente esamina i vertici direttamente raggiungibili da questo vertice. Tuttavia, in presenza di vertici con un alto fanout, cioè con molti vertici raggiungibili, l'approccio iterativo dei singoli thread potrebbe non essere quello ottimale.
Parallelizzare queste iterazioni con il parallelismo dinamico migliorerebbe ulteriormente le performance. Con il parallelismo dinamico, ciascun thread Cuda che gestisce un vertice con alto fanout lancerebbe a sua volta un altro kernel Cuda, al fine di parallelizzare l'inserimento sull'array "Next" dei vertici direttamente raggiungibili. Ognuno dei thread Cuda qui lanciati gestiranno un singolo vertice raggiungibile, e avranno il compito di inserirlo sull'array "Next" o scartarlo sulla base di criteri personalizzati.
Questo metodo ha le potenzialità di ridurre significativamente i tempi complessivi se l'architettura presenta mediamente un fanout alto per i suoi vertici.

8.2.2 Uso della CPU per pochi nodi

Potremmo valutare l'uso della CPU invece della GPU per calcolare quelle iterazioni del BFS che esplorano un numero estremamente ridotto di nuovi nodi. Questo approccio potrebbe migliorare sensibilmente le prestazioni, dato che l'esecuzione di un kernel Cuda comporta un certo overhead. Se i nuovi nodi scoperti sono molto pochi, il tempo perso nel lanciare il kernel Cuda potrebbe superare il guadagno prestazionale della parallelizzazione.

Bibliografia

[1] it.rs-online.com. *Cosa sono gli FPGA e come programmarli*. [Online]. 2023 (cit. a p. 4).

[2] slideshare.net. *Fpga architectures and applications*. [Online]. 2020 (cit. a p. 9).

[3] Jason H. Anderson e Tomasz S. Czajkowskii. «Computer-Aided Design for FPGAs: Overview and Recent Research Trends». In: *Proceedings of the 35th annual conference on Design automation* (1998), pp. 269–274 (cit. alle pp. 9, 11, 13).

[4] Ali Umut Irturk. *GUSTO : general architecture design utility and synthesis tool for optimization*. 2009 (cit. a p. 10).

[5] From Wikipedia. *Field-programmable gate array*. [Online] (cit. a p. 14).

[6] S. Azimi, C. De Sio, A. Portaluri, D. Rizzieri e L. Sterpone. «A comparative radiation analysis of reconfigurable memory technologies: FinFET versus bulk CMOS». In: *Microelectronics Reliability* 138 (2022). 33rd European Symposium on Reliability of Electron Devices, Failure Physics and Analysis, p. 114733. ISSN: 0026-2714. DOI: https://doi.org/10.1016/j.microrel.2022.114733. URL: https://www.sciencedirect.com/science/article/pii/S0026271422002578 (cit. a p. 14).

[7] Dirk Stroobandt Yun Zhou Dries Vercruyce. «Accelerating FPGA Routing Through Algorithmic Enhancements and Connection-aware Parallelization». In: *ACM Transactions on Reconfigurable Technology and Systems* 13 (ago. 2020), pp. 1–26 (cit. alle pp. 27, 58).

[8] Jason H. Anderson Marcel Gort. «Accelerating FPGA Routing Through Parallelization and Engineering Enhancements Special Section on PAR-CAD 2010». In: *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 31 (gen. 2012), pp. 61–74 (cit. alle pp. 27, 28, 57).

[9] Guojie Luo; Nong Xiao Minghua Shen Wentai Zhang. «Serial-Equivalent Static and Dynamic Parallel Routing for FPGAs». In: *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 39 (feb. 2020), pp. 411–423 (cit. alle pp. 28, 57).

[10] Jason H. Anderson Marcel Gort. «Deterministic multi-core parallel routing for FPGAs». In: *2010 International Conference on Field-Programmable Technology* (dic. 2010) (cit. alle pp. 28, 57).

[11] Deshanand Singh Doris Chen. «Parallelizing FPGA Technology Mapping Using Graphics Processing Units (GPUs)». In: *2010 International Conference on Field Programmable Logic and Applications* (set. 2010) (cit. a p. 29).

[12] Deborah Stacey Christian Fobel Gary Grewal. «GPU-Accelerated Wire-Length Estimation for FPGA Placement». In: *2011 Symposium on Application Accelerators in High-Performance Computing* (lug. 2011) (cit. a p. 29).

[13] Deheng Ye Nachiket Kapre. «GPU-accelerated high-level synthesis for bitwidth optimization of FPGA datapaths». In: *FPGA '16: Proceedings of the 2016 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays* (feb. 2016), pp. 185–194 (cit. a p. 29).

[14] Nong Xiao Minghua Shen Guojie Luo. «Exploring GPU-Accelerated Routing for FPGAs». In: *IEEE Transactions on Parallel and Distributed Systems* 30 (dic. 2018), pp. 1331–1345 (cit. a p. 29).

[15] Guojie Luo Minghua Shen. «Corolla: GPU-Accelerated FPGA Routing Based on Subgraph Dynamic Expansion». In: *FPGA '17: Proceedings of the 2017 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays* (feb. 2017), pp. 105–114 (cit. a p. 29).

[16] nanoxplore.com. *NanoXplore*. [Online] (cit. a p. 31).

[17] Sarah Azimi Andrea Portaluri e Luca Sterpone. «NXRouting: a GPU-enhanced CAD Tool for European Radiation-Hardened FPGAs». In: *Electronics* (2024, in press) (cit. alle pp. 32–34).

[18] nanoxplore.com. *NanoXplore announces their new Head of R&D Suzie Marin*. [Online]. Feb. 2024 (cit. alle pp. 32, 35).

[19] nanoxplore.com. *NanoXplore's NG-ULTRA officially supported by OpenOCD*. [Online]. Nov. 2023 (cit. alle pp. 32, 35).

[20] E. Vacca, S. Azimi e L. Sterpone. «Failure rate analysis of radiation tolerant design techniques on SRAM-based FPGAs». In: *Microelectronics Reliability* 138 (2022). 33rd European Symposium on Reliability of Electron Devices, Failure Physics and Analysis, p. 114778. ISSN: 0026-2714. DOI: https://doi.org/10.1016/j.microrel.2022.114778. URL: https://www.sciencedirect.com/science/article/pii/S002627142200302X (cit. a p. 33).

[21] Eleonora Vacca, Sarah Azimi e Luca Sterpone. «A Comprehensive Analysis of Transient Errors on Systolic Arrays». In: *2023 26th International Symposium on Design and Diagnostics of Electronic Circuits and Systems (DDECS)*. 2023, pp. 175–180. DOI: 10.1109/DDECS57882.2023.10139763 (cit. a p. 33).

[22] Eleonora Vacca, Giorgio Ajmone e Luca Sterpone. «RunSAFER: A Novel Runtime Fault Detection Approach for Systolic Array Accelerators». In: *2023 IEEE 41st International Conference on Computer Design (ICCD)*. 2023, pp. 596–604. DOI: 10.1109/ICCD58817.2023.00095 (cit. a p. 33).

[23] Github. *Polito ITC99 (I99T)*. [Online]. 2018 (cit. alle pp. 47, 77).

[24] C. Ebeling L. McMurchie. «PathFinder: A Negotiation-Based Performance-Driven Router for FPGAs». In: *Third International ACM Symposium on Field-Programmable Gate Arrays* (feb. 1995) (cit. a p. 58).

[25] Ian Buck. *Chapter 32. Taking the Plunge into GPU Computing*. [Online]. Apr. 2005 (cit. a p. 60).

[26] David B. Kirk e Wen-mei W. Hwu. *Programming Massively Parallel Processors: A Hands-on Approach*. Morgan Kaufmann Publishers, 2021 (cit. a p. 72).