ALMA MATER STUDIORUM - UNIVERSITÀ DI BOLOGNA

SCUOLA DI INGEGNERIA E ARCHITETTURA

CORSO DI LAUREA MAGISTRALE IN INGEGNERIA ELETTRONICA

TESI DI LAUREA IN

Hardware - Software Design of Embedded Systems M

FRAMEWORK PER IL BENCHMARKING COMPARATIVO DEI COMPONENTI SOFTWARE SU PIATTAFORME EMBEDDED MULTI-CORE

Candidato: Dario Chechi

Relatore: Luca Benini

Correlatore: Giuseppe Tagliavini

Anno Accademico 2020-2021

Sommario

1 Introduzione

Il mercato dei dispositivi embedded low-power ha subìto negli ultimi anni una forte crescita, la quale ha portato allo sviluppo di tecnologie e architetture innovative: il bisogno di prestazioni sempre più elevate ha fatto sì che le frequenze di clock fossero sempre più spinte, ma ciò ha comportato anche un aumento dei consumi. Poiché le condizioni operative di queste macchine prevedono spesso l'alimentazione mediante batterie, tali consumi così elevati non erano sostenibili e perciò è stato necessario intraprendere un'altra strada: quella delle piattaforme multi-core.

Anziché avere un solo processore centrale con prestazioni elevate ma dai consumi eccessivi, le nuove architetture propongono molti processori che lavorano contemporaneamente in parallelo ma a frequenze molto più basse, con il risultato di avere un sistema con consumi sensibilmente ridotti.

Consumi contenuti, uniti allo sfruttamento dell'*edge processing* e del funzionamento *near-threshold*, hanno consentito la riduzione delle dimensioni dei chip e l'applicazione di tali tecnologie in svariati settori: l'industria, l'agricoltura, l'intrattenimento e tanti altri hanno goduto dei vantaggi introdotti dai sistemi embedded low-power, spianando la strada per la crescita dell'Internet of Things (IoT), una realtà in cui tutti i dispositivi sono connessi e intercomunicanti.

Tuttavia, pur avendo ottenuto risultati notevoli, il mercato è ancora in forte crescita e le nuove tecnologie in ambito low-power puntano ad incrementare la propria efficienza energetica mediante un connubio di ottimizzazione e flessibilità: non più hardware statico e sempre attivo, ma una soluzione dinamica che permette di scegliere quante e quali risorse utilizzare all'occorrenza, ibernando quelle superflue e portando ad un risparmio energetico inimmaginabile fino a poco tempo fa.

Ed è proprio da questi sviluppi che nasce PULP **[1]**, la nostra piattaforma di riferimento, la quale presenta un'architettura altamente configurabile, a partire dal

numero di core selezionabili, e che rappresenta l'evoluzione perfetta dei sistemi embedded ultra low-power.

Questa piattaforma sfrutta l'ISA RISC-V **[2]**, ossia un set di istruzioni open-source che permette di abbassare notevolmente i costi di produzione dell'hardware; infatti una buona parte del costo è normalmente legata alle licenze (si pensi alle note architetture ARM o x86), mentre nel caso di licenza open-source questa componente del costo viene azzerata.

Dal punto di vista hardware, come già detto, PULP presenta un sistema fortemente programmabile: il codice viene elaborato dal *fabric controller* **[3]**, il quale esamina quali siano le risorse necessarie (i vari core, le periferiche, le memorie ecc.) e le prepara per l'esecuzione; dopodiché avviene l'esecuzione parallela vera e propria, durante la quale i vari core comunicano mediante una memoria condivisa di primo livello, mentre da un'altra memoria (quella di secondo livello) leggono le istruzioni da eseguire.

Altro aspetto importante da tenere in considerazione è la presenza dei contatori delle performance **[4]**, dei contatori hardware che mediante dei flag permettono di tenere conto dei fattori di non idealità senza avere alcun impatto sulle performance, o comunque con un impatto fortemente trascurabile legato alle funzioni di attivazione e lettura.

Avendo parlato della versatilità hardware di PULP, conviene poi fare qualche considerazione anche sulla sua versatilità software: qui entra in gioco GVSoC **[5]**, una *virtual platform* **[6]** (i.e. un simulatore) che ha come pregio quello di essere una via di mezzo fra le due soluzioni attualmente presenti sul mercato; da una parte abbiamo simulatori molto accurati, che simulano l'esecuzione del programma ciclo per ciclo ma richiedendo tempi molto estesi, dall'altra ci sono simulatori più rapidi ma molto meno precisi.

GVSoC si pone quindi a metà strada e propone un livello di accuratezza piuttosto elevato ma, al tempo stesso, anche tempi di simulazione contenuti.

Risulta quindi chiaro che lo spazio di esplorazione sia molto vasto, poiché le configurazioni possibili sono in numero elevatissimo; sviluppare un progetto su questa piattaforma significa anche capire quali siano le configurazioni che facciano al caso nostro in base alle nostre esigenze: è più importante avere prestazioni elevate o consumi ridotti?

È evidente che esaminare ogni possibile soluzione una per una diventi un problema non indifferente, specialmente se si intendono raccogliere tutti i dati e magari inserirli in un grafico per avere un'immagine di ciò che abbiamo ottenuto; per questo motivo nasce il *framework* oggetto di questo lavoro: l'obiettivo è quello di fornire agli sviluppatori uno strumento che permetta loro di eseguire tutte queste comparazioni in maniera semplice e veloce.

Il programma, infatti, prevede il seguente flusso di lavoro:

Figura 1 - Workflow del framework

L’esecuzione del framework viene lanciata da riga di comando: in questa prima fase un file di configurazione in formato JSON **[7]**, preparato precedentemente dall’utente e contenente le indicazioni per la compilazione ed esecuzione del programma, viene passato in ingresso allo script *manager*, ossia uno script Python che legge il tipo di confronto da eseguire e lancia lo script specifico passandogli il file.

Tale script, leggendo i parametri contenuti nel file di configurazione, prima prepara l’ambiente di esecuzione (impostando le *environment variables* della toolchain di compilazione e della virtual platform), poi lancia una serie di sottoprocessi con i quali vengono compilate ed eseguite le varie versioni da confrontare del programma; fatto ciò, l’output di ogni sottoprocesso viene letto e i dati scompattati, andando a riempire i campi di un dizionario contenente tutti i risultati.

Infine questi risultati vengono processati, calcolando ulteriori dati quali i consumi e lo *speedup* **[8]**, per poi essere presentati in due finestre diverse: una contenente i grafici a barre relativi alle performance ed una contenente i grafici a torta che rappresentano i risultati delle non idealità, le quali portano a risultati diversi da quelli teorici.

2 Piattaforma di riferimento

Nel corso degli anni la complessità dei programmi è aumentata in maniera esponenziale; tale crescita, unita ad altre necessità come ridurre i consumi energetici e migliorare la latenza nelle trasmissioni di dati, ha portato ad una sistematica evoluzione delle piattaforme hardware.

Inizialmente nell'ambito **ULP** (Ultra Low-Power) le piattaforme di riferimento erano i sistemi embedded a **microcontrollore**, i quali offrivano performance in linea con le specifiche richieste ma con consumi veramente ridotti.
Un esempio è **Cortex-M4 [9]**, un core con architettura ARM con prestazioni avanzate di controllo digitale ideali per applicazioni **DSP** (Digital Signal Processing) ma che offre al tempo stesso la semplicità operativa tipica della famiglia Cortex-M.

Ridotto assorbimento energetico e bassi costi lo hanno reso una delle piattaforme migliori per applicazioni nel settore *automotive* e industriale.

Come già accennato, però, col passare del tempo il costo computazionale dei vari programmi è aumentato e tali soluzioni hardware si sono rivelate essere non più ottimali; essendo dispositivi **single-core**, l'unica soluzione per avere un aumento delle performance era quella di aumentare il clock della CPU, che però avrebbe portato un aumento notevole dei consumi rendendo quindi la piattaforma non più adatta per applicazioni ULP.

In questi casi si parla spesso di ***near-sensor processing***, ossia dell'elaborazione dei dati vicino ai nodi sensore; un esempio di questo tipo di approccio si può riscontrare nell'ambito agricolo, dove i moderni sistemi IoT raccolgono dati utili tramite apposite stazioni (i **nodi sensore**, appunto) e dei piccoli droni passano a raccogliere le informazioni grezze, le elaborano e le inviano al server principale.

Risulta chiaro che i sistemi di raccolta dei dati (droni, in questo esempio) siano spesso alimentati a batteria e di conseguenza i consumi dell'hardware a bordo

devono essere fortemente ridotti, ma al tempo stesso l'elaborazione dei dati deve essere veloce.

La soluzione più promettente a questo problema è stata trovata nell'utilizzo di sistemi **multi-core**: andando a parallelizzare l'esecuzione dei programmi su più core a basso consumo (soluzione ***near-threshold***, si lavora a tensioni prossime alla soglia di accensione), si migliora sensibilmente il tempo necessario al completamento dell'elaborazione; si parla dunque di ***race to idle***, ossia si preferisce consumare di più ma per un tempo molto ridotto, in modo da terminare velocemente tutte le istruzioni e far tornare i core in una condizione di riposo in cui i consumi sono trascurabili.

Tutti questi requisiti hanno portato allo sviluppo della piattaforma sulla quale ci andremo a focalizzare nel resto di questo elaborato di tesi: l'architettura PULP **[1]**.

2.1 L'architettura PULP

L'architettura PULP (Parallel Ultra Low Power) è un'architettura il cui core è basato su ISA RISC-V **[2]**, ossia un set di istruzioni ridotte (RISC = Reduced Instruction Set Computer) il cui vantaggio è quello di essere allo stesso tempo sia semplice che pratico e performante, permettendo una implementazione hardware veramente veloce; la natura open-source di questa ISA la rende veramente versatile, tanto che negli ultimi anni è riuscita a prendere piede anche in diversi ambiti industriali.

Il grosso vantaggio rispetto ai principali concorrenti (ARM in primis, avendo come ambito di riferimento quello dei dispositivi low-power) è che promuove lo sviluppo di un ecosistema hardware **open-source**, il quale rappresenta un nuovo modello di business: l'idea è di abbattere i costi dei chip x86 e ARM, legati principalmente a questioni di licenza.

L'obiettivo di RISC-V è quello di rendere più liberi i produttori di hardware, di liberarli dalle catene che li tengono legati e permettere loro di realizzare dispositivi

performanti ma dal costo contenuto, esattamente come è successo in ambito software con l'introduzione delle licenze *open-source*.

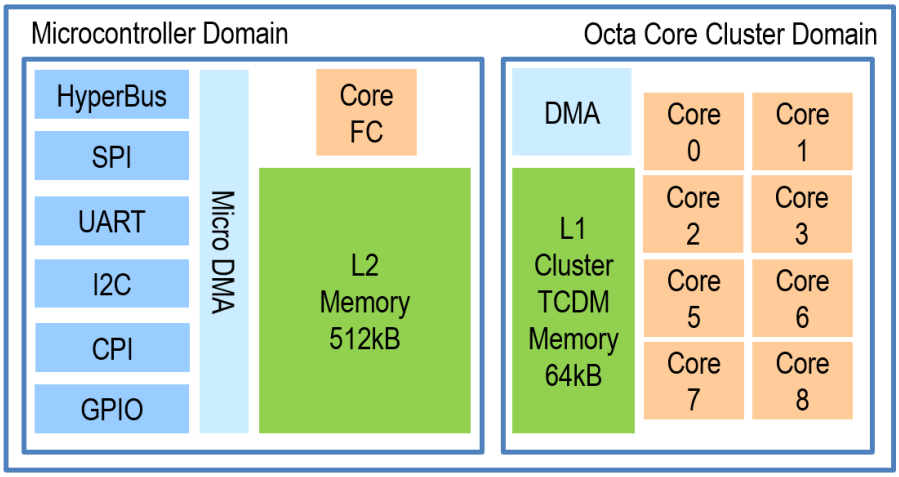

Figura 2 - Architettura PULP

Fra le caratteristiche hardware principali dell'architettura PULP troviamo la presenza di un *fabric controller* **[3]** come core MCU e l'utilizzo di un *cluster* con un numero parametrico di core per la computazione parallela: nella pratica il *fabric controller* esamina le istruzioni e programma un numero variabile e programmabile di core affinché questi le eseguano, permettendo di raggiungere un'elevata parallelizzazione e di conseguenza delle prestazioni molto spinte, ma all'occorrenza anche di usare meno risorse e abbattere i consumi energetici.

L'utilizzo di un numero di core inferiore rispetto a quello disponibile è un approccio che in determinati casi può risultare vantaggioso per due motivi principali. Il primo è semplicemente legato al consumo energetico, in quanto è scontato che far lavorare meno core significa mettere "a riposo" quelli in eccesso ed abbattere i consumi. Il secondo, meno intuitivo, è legato alle performance: nel caso ideale il codice che viene eseguito è infinitamente parallelizzabile, cioè in grado di sfruttare al massimo tutti i core disponibili, fornendo quindi performance sempre più spinte; nella realtà questo non si verifica e ciò significa che non tutte le sezioni di codice sono pensate per scalare su un alto numero di core.

Il risultato è che in molti casi (specialmente con codice non ottimizzato) ci sono delle sequenze di istruzioni che creano i cosiddetti **overhead**, cioè delle situazioni in cui alcuni core possono eseguire le istruzioni mentre altri devono attendere, sprecando quindi dei cicli di clock senza alcuna utilità.

Il problema degli overhead fa sì che in alcune situazioni non solo un numero di core maggiore porta benefici minimi o nulli, ma che addirittura porti a performance peggiorative rispetto ai casi con meno core.

Fra le caratteristiche della sezione microcontrollore possiamo notare una vasta serie di porte di comunicazione quali I2C, SPI, UART ed anche pin generici, i cosiddetti GPIO (General Purpose Input Output); è presente poi una memoria di secondo livello (L2).

Inoltre è presente un componente Micro DMA **[10]**, che permette di gestire l'interazione con le periferiche esterne (tipicamente trasferimenti di dati da/verso la memoria L2) lasciando libere le risorse del processore e permettendo a questo di eseguire altre operazioni in contemporanea.

Nella sezione del cluster si può osservare, invece, la presenza di un DMA controller "classico", che viene usato per lo scambio diretto di dati fra le memorie L1 e L2.

Nella architettura di PULP, la memoria L1 è condivisa dai vari core del cluster per lo scambio e la gestione dei dati, mentre il codice risiede in L2 e viene acceduto tramite una cache dedicata alle istruzioni. Questa caratteristica è fondamentale in quanto incide direttamente sui modelli di programmazione utilizzati.

Per quanto riguarda infatti i trasferimenti di memoria tra i vari livelli, questi sono espliciti: ciò significa che non è presente una memoria per ogni singolo core, ma che tutti i core condividono una certa area di memoria denominata TCDM (Tightly-Coupled Data Memory).

L'utilizzo di questa memoria condivisa permette un rapidissimo scambio di dati fra i core in un singolo ciclo, ma al tempo stesso si verifica il problema delle

contese: quando più core vogliono scrivere su questa memoria, solamente uno potrà ottenerne il controllo ed eseguire la scrittura, pertanto gli altri dovranno attendere il completamento dell'operazione e questo porta ad un rallentamento nell'esecuzione del programma.

2.2 Come si programma PULP

La programmazione su PULP, il quale dispone di un proprio **SDK** (Software Development Kit) **[11]**, avviene esattamente come con qualsiasi altro dispositivo di classe MCU, ossia scrivendo il programma mediante un linguaggio di programmazione (generalmente si usa il linguaggio C) il quale viene poi compilato attraverso appositi tool ed eseguito sulla macchina, nel nostro caso dal fabric controller.

Trattandosi di un sistema embedded parallelo a memoria condivisa (nel nostro caso la memoria L1 del cluster), il paradigma di programmazione prevede la scelta fra due possibili approcci: il ***fork-join*** e il ***SPMD*** (Single Program Multiple Data) **[12]**.

Nel caso del paradigma *fork-join*, l'esecuzione del programma parte con un singolo *thread* e continua così finché non trova una regione parallela, delimitata dall'istruzione di *fork*; a questo punto il codice viene eseguito su più core in contemporanea fino a che la regione parallela non termina, ossia quando si presenta un'istruzione di *join* con la quale si vanno a sincronizzare i vari thread per poi tornare ad un'esecuzione sequenziale.

L'alternativa a questo approccio è data dal paradigma *SPMD*, in cui tutti i core fanno partire l'esecuzione del programma in maniera simultanea. Single Program Multiple Data significa che tutti i core eseguono le medesime istruzioni (Single Program) ma lo fanno su blocchi di dati diversi (Multiple Data).

Per la piattaforma PULP è inoltre possibile utilizzare il modello di programmazione ad alto livello **OpenMP [13]**, un modello di programmazione per piattaforme a

memoria condivisa che consiste nell'uso di direttive interpretate dal compilatore, chiamate direttive *pragma*, unite ad una libreria di supporto.

Tale modello risulta molto versatile in quanto chi sviluppa il software deve solamente aggiungere delle annotazioni per il compilatore e questo provvederà ad eseguire le dovute trasformazioni per parallelizzare il codice nel modo più efficiente per l'architettura in esame.

Per esempio, se si vuole distribuire l'esecuzione di un ciclo *for* su più core si può utilizzare la direttiva ***#pragma omp parallel for*** prima del ciclo ed il compilatore si occuperà di trasformare il codice di conseguenza, distribuendo le singole iterazioni sui core disponibili.

Chiaramente affinché la parallelizzazione sia possibile occorre che le istruzioni eseguite all'interno del ciclo non siano dipendenti tra loro: se l'istruzione all'iterazione richiede il risultato di quella all'iterazione k-1, per esempio, risulta chiaro che il codice non possa essere eseguito in parallelo ma in maniera sequenziale, e questo deve essere verificato dal programmatore.

OpenMP prevede due *runtime* principali: ***OMP-base*** e ***OMP-opt***.

La prima è un'implementazione base per un sistema embedded e ha lo scopo di ridurre il *footprint* del codice (i.e. lo spazio occupato in RAM) e il suo tempo di esecuzione; la seconda invece punta a spremere al meglio l'hardware a disposizione al fine di migliorare la sincronizzazione e il *core idling*, ossia cerca di far completare le istruzioni ai core molto velocemente in modo che possano mettersi a riposo energetico il prima possibile.

Vi è però anche un terzo approccio basato sul paradigma SPMD che prende il nome di **OMP-SPMD**, il quale permette di applicare la sintassi OpenMP al paradigma SPMD.

Questo approccio, di recente introduzione, permette di sfruttare al meglio l'hardware minimizzando gli overhead: come si vede nella figura successiva, in molti casi le differenze rispetto agli approcci precedenti sono notevoli.

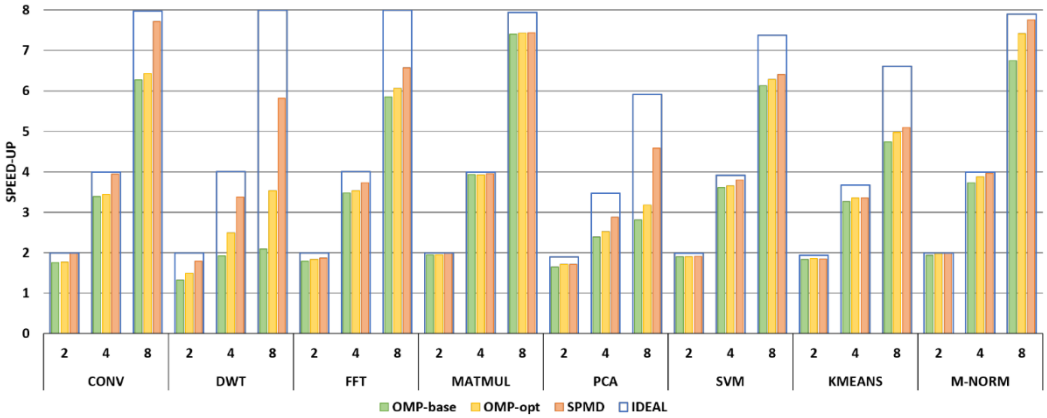

Figura 3 - Confronto fra approcci precedenti e OMP-SPMD in varie applicazioni

Il punto forte di OMP-SPMD risiede nelle sue API (Application Programming Interface) di basso livello che permettono di fornire indicazioni al FC su come gestire le istruzioni, quindi su come mandarle in esecuzione sui vari core del cluster in maniera trasparente per il programmatore.

Tutte le funzionalità di basso livello sono fornite da una libreria di supporto alla piattaforma chiamata **PMSIS** (PULP Microcontroller Software Interface Standard). Tramite questa libreria il fabric controller vede il cluster come un dispositivo e lo può aprire e chiudere come una porta di comunicazione; il ciclo vitale del dispositivo è gestito secondo i seguenti step:

- *Configurazione e inizializzazione*: fase preliminare in cui il dispositivo viene, appunto, configurato e le sue risorse allocate
- *Preparazione*: tramite la funzione **open( )** il dispositivo viene "aperto", cioè si prepara all'esecuzione del codice
- *Esecuzione*: il dispositivo esegue le operazioni richieste
- *Chiusura*: tramite la funzione **close( )** vengono liberate le risorse.

Il codice eseguito sul cluster parte su un solo core (esecuzione sequenziale) fino a che non viene raggiunta un'istruzione di *fork*, nella quale si indica il numero di core da utilizzare, il puntatore alla funzione da eseguire e il puntatore ai dati.

L'esecuzione continua quindi secondo il modello SPMD, con funzioni come il recupero dell'identificativo numerico del core ed altre funzionalità di base: un esempio è l'utilizzo dell'istruzione *barrier( )* per la sincronizzazione, con la quale si attende che i vari core abbiano terminato le loro istruzioni per tornare poi all'esecuzione sequenziale su singolo core, oppure le istruzioni *critical_enter( )* e *critical_exit()* con le quali si delimitano zone critiche, cioè sezioni di codice a cui può accedere un solo core per volta durante l'esecuzione parallela.

2.3 Ambiente di compilazione ed esecuzione

Una volta scritto il programma con il linguaggio scelto, si deve compilare per creare un file binario eseguibile: per fare ciò si utilizza una *toolchain* di compilazione, letteralmente "catena di strumenti", ossia un insieme di programmi dove il file di uscita di un programma diventa il file di ingresso del successivo fino a che non si ottiene il risultato finale.

In questo caso specifico è stata utilizzata la toolchain GCC **[14]** standard per la ISA RISC-V, la quale fa parte di un progetto open-source (disponibile su GitHub) e che quindi viene continuamente aggiornata, offrendo grande supporto a un elevato numero di piattaforme.

Occorre però specificare che la toolchain GCC non è l'unica opzione disponibile per la compilazione: esiste infatti un'altra possibilità data dalla toolchain LLVM **[15]**.
Poiché il compilatore gestisce la conversione del codice in linguaggio macchina e tutte le relative ottimizzazioni, usare una toolchain diversa significa poter ottenere risultati nettamente diversi, sia in senso migliorativo che peggiorativo.
Si deve inoltre tenere conto del fatto che i vari compilatori accettano anche i

cosiddetti ***flag di compilazione***, ossia dei parametri che l'utente può specificare prima di lanciare la compilazione e che vanno ad indicare al compilatore come comportarsi durante il processo (per esempio, usando il flag *-verbose* si specifica di fornire un output più ricco di informazioni, mentre col flag *-O2* o *-O3* si indica il livello di ottimizzazione di interesse).

Per quanto riguarda l'esecuzione, normalmente si andrebbe ad utilizzare una board hardware, quindi una vera e propria scheda fisica, tuttavia nel caso preso in esame ciò non è possibile: l'intento di questo lavoro, infatti, è quello di fornire uno strumento veloce di comparazione dell'esecuzione dei programmi, i quali possono essere eseguiti anche su tipi diversi di board.

Pertanto è stato necessario l'utilizzo di una **virtual platform**, ossia un'interfaccia software che fa da simulatore ed esegue il programma come lo farebbe una scheda fisica; in questo modo è possibile variare la configurazione hardware con estrema facilità ed eseguire confronti fra diverse board in maniera molto rapida.

2.4 GVSoC

Il nome della virtual platform **[6]** di PULP è **GVSoC [5]**, ma per comprendere a fondo quanto performante e ottimale essa sia bisogna prima fare una panoramica sulle altre piattaforme di simulazione presenti in questo ambito.
Tali piattaforme si possono dividere idealmente in due categorie:

- *Simulatori cycle-by-cycle*: sono quelli più accurati in quanto vanno a simulare l'esecuzione ciclo per ciclo e forniscono una notevole quantità di informazioni, come per esempio dati sul *timing* oltre che sull'esecuzione logica dei programmi, ma questa accuratezza comporta dei tempi di simulazione molto lunghi
- *Simulatori rapidi*: sono simulatori molto rapidi in quanto forniscono i risultati in tempi brevi, ma per farlo rinunciano a un'accuratezza elevata, per cui le informazioni raccolte sono molte meno (generalmente simulano

solo le operazioni logiche, senza eseguire alcune analisi sulle performance temporali dell'hardware che vanno a simulare).

In questo panorama è proprio in GVSoC che si trova il punto d'incontro fra le due categorie: tale piattaforma, infatti, è in grado di simulare piattaforme hardware molto complesse (multi-core, multi-IO, con acceleratori ecc.) con un errore veramente limitato, ma lo fa in maniera molto rapida, con tempi di simulazione che sono fino a 2500 volte inferiori rispetto a quelli dei simulatori più accurati.

GVSoC rientra nell'insieme dei simulatori ***event-driven*** in quanto non riproduce esattamente l'esecuzione ciclo per ciclo, ma utilizza un concetto più astratto denominato **evento**, il quale è inteso come un cambio di stato nel sistema che avviene in un determinato istante temporale.

L'obiettivo di questo simulatore è quello di permettere agli sviluppatori di testare nuove funzionalità dell'architettura (i.e. le estensioni ISA) in maniera semplice e veloce, oltre che aiutare questi nell'analisi del design per l'aggiunta di ulteriori *chip features*.

La sua struttura include tre componenti principali:

- *Modelli C++*: descrivono i comportamenti dei componenti del sistema simulato (es. core, memorie, DMA, periferiche…)
- *File JSON di configurazione* **[7]**: configurano i parametri dell'architettura, come la larghezza di banda e la latenza delle interconnessioni
- *Set di generatori Python*: un insieme di generatori, appunto, che istanziano tutti i componenti della specifica piattaforma target.

Questa struttura modulare permette la compilazione dei modelli C++ all'inizio e successivamente vengono eseguite delle variazioni semplicemente mediante la modifica dei file JSON.

In tali file viene descritta la piattaforma da emulare e tutti i suoi moduli, inclusi il fabric controller, la memoria principale e i set di periferiche; tutti questi modelli

fanno parte di una libreria che può essere "assemblata" a *run-time* durante la compilazione per costruire il sistema che deve essere simulato.

La gestione degli eventi avviene nel seguente modo: quando una richiesta viene inviata da un componente X ad un componente Y, quest'ultimo estrae le informazioni che gli servono e risponde con un opportuno valore di ritorno in base alla natura della richiesta; se tale richiesta deve essere inoltrata a un altro componente, un metodo specifico è incaricato di creare una nuova richiesta e passarla a tale componente attraverso una delle porte principali.

Ogni componente intermedio che viene aggiunto può andare ad aumentare la latenza della richiesta iniziale; la catena che si crea sarà conclusa quando viene raggiunto il componente ricevitore oppure quando viene generato un nuovo evento.

Ciò che è appena stata descritta è la simulazione **funzionale**, ma come già detto GVSoC esegue anche un'analisi **temporale** della piattaforma; tale analisi permette di modellare attività importanti come l'esecuzione di istruzioni, trasferimenti DMA e accessi alla memoria.

In tale contesto, un *time engine* globale gestisce il tempo totale (in una scala di picosecondi), mentre un *clock engine* modella la sorgente di clock come un contatore al quale viene associata una coda di eventi correlati; ogni evento include un insieme di dati (*data payload*) e un puntatore ad una funzione di *callback* associata.

Nella pratica il *clock engine* va a definire una finestra temporale T_W nella quale gli eventi sufficientemente vicini vengono inclusi in un buffer circolare e poi eseguiti ciclo per ciclo; nel caso di eventi sovrapposti, questi vengono eseguiti in maniera sequenziale senza tener conto dell'ordine della coda in cui sono inseriti.

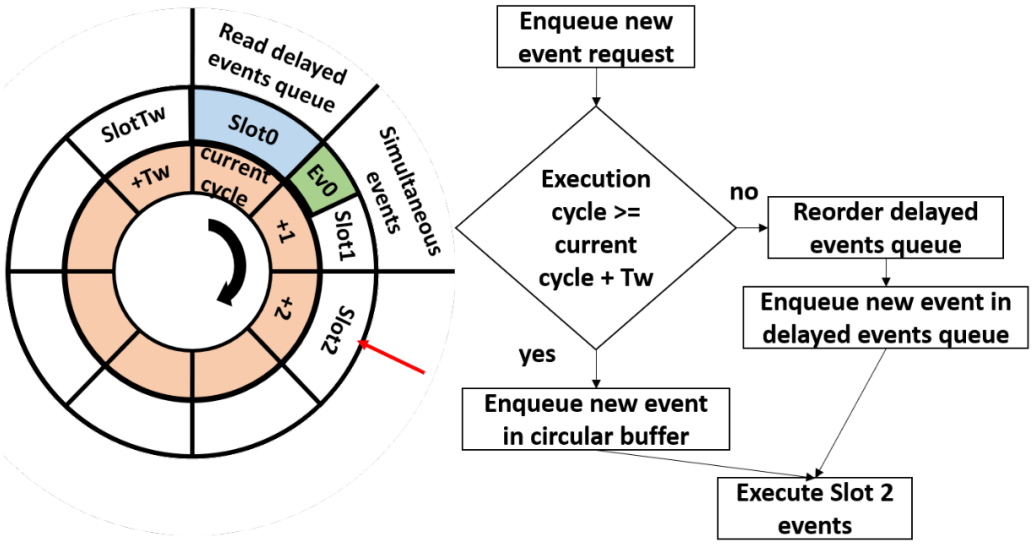

Figura 4 - Gestione degli eventi di GVSoC, con buffer circolare che contiene gli eventi nella coda di esecuzione

Per poter infine riportare questi eventi nel dominio temporale globale, il periodo di clock viene moltiplicato per la differenza fra il contatore del clock corrente e il tempo di clock associato all'evento in questione.

Le informazioni ottenute da questa analisi sono contenute anche in un set di ***system traces***, le quali sono fondamentali per la procedura di debug in quanto contengono dati riguardo ciò che succede durante l'esecuzione nei moduli di maggior rilievo. Ogni *trace* può coincidere con un singolo evento oppure, nel caso di eventi particolarmente critici, più tracce possono essere associate allo stesso evento.

2.5 Misurazione delle performance

Per poter capire quali siano le reali prestazioni delle schede quando eseguono il programma, sono stati definiti ed implementati i cosiddetti ***performance counter*** **[4]**: questi non sono altro che dei contatori hardware che vengono incrementati ogni qual volta venga alzato il corrispettivo flag all'interno della scheda, di conseguenza

non vanno ad impattare in alcun modo le prestazioni di essa (o comunque lo fanno con un overhead fortemente ridotto).

I performance counter principali sono:

- **Number of cycles**: il numero di cicli di clock richiesti per portare a compimento l'esecuzione del codice profilato
- **Number of active cycles**: numero dei cicli in cui il core è attivo, ovvero non si trova in stato di clock-gating
- **Load stalls**: gli stalli di caricamento, si verificano quando una istruzione vuole accedere ad un registro il cui valore è stato caricato da un precedente istruzione di load che non ha ancora terminato. La latenza di accesso alla L1 è di un ciclo, quella della L2 è un parametro che varia da 8 a 12 cicli
- **TCDM contentions**: le contese della Tightly-Coupled Data Memory, che si verificano quando un core cerca di accedere ad uno stesso banco della TCDM ma lo trova già in uso da un altro core

Vi sono poi altre possibili cause di rallentamenti, come per esempio le operazioni di branch o le instruction miss, che sono problemi minori e pertanto vengono riportate tutte insieme con il nome di **Others**.

3 Definizione dello spazio di esplorazione

Come detto già in precedenza, l'obiettivo del progetto è quello di permettere un rapido confronto fra diverse configurazioni; occorre dunque specificare in che cosa tali configurazioni possano differire per capire poi come approcciarsi al problema.

3.1 Virtual platform

Il primo fattore che può variare è sicuramente la configurazione della **virtual platform**: può essere interessante, infatti, vedere quali risultati si possano ottenere variando la piattaforma hardware.

Come visto nel paragrafo 2.1, PULP è una piattaforma estremamente configurabile e ciò permette di spaziare facilmente fra tante opzioni differenti: oltre al numero di core, che sarà discusso in uno dei paragrafi successivi, le varie piattaforme possono differire in vari componenti come le memorie, i sensori e le periferiche.

Se per esempio consideriamo lo scambio di dati con l'esterno, gestito dalle periferiche di I/O, o fra le varie memorie dello stesso dispositivo, il modo in cui i dati vengono gestiti può impattare sensibilmente sulle prestazioni finali; se l'hardware simulato non implementa, sempre per esempio, una tecnica di scambio dati basata su DMA (Direct Memory Access, cioè accesso diretto alla memoria) sarà la CPU a dover gestire la transazione: se la CPU si deve occupare di questo, dovrà ritardare l'esecuzione delle altre istruzioni, portando quindi ad un aumento del numero di cicli richiesti per completare l'esecuzione del programma e dunque delle performance inferiori.

3.2 Toolchain di compilazione

Un altro fattore di interesse è la **toolchain di compilazione**: il processo di compilazione, infatti, è quel punto della catena in cui il codice scritto dal programmatore viene convertito in linguaggio macchina, con tutte le ottimizzazioni

del caso; di conseguenza una variazione nel compilatore potrebbe introdurre sia dei vantaggi che degli svantaggi, che vanno valutati caso per caso.

La toolchain di compilazione ufficiale del progetto PULP si basa su GCC 7.1. Recentemente gli sviluppatori stanno valutando un passaggio a una versione più recente del compilatore (GCC 11); parallelamente, ci sono esperimenti per utilizzare una toolchain di compilazione basata su LLVM 12. Questi contributi rendono necessaria l'adozione di una opportuna metodologia per valutare le prestazioni, in quanto queste attività di sviluppo risultano alquanto onerose e oltre alla correttezza funzionale è importante valutare l'impatto sulle prestazioni.

3.3 Flag di compilazione

Un aspetto direttamente correlato alla scelta della toolchain di compilazione è la scelta dell'insieme dei **flag di compilazione**: quando si converte il codice di alto livello in codice macchina, si possono specificare delle opzioni, delle "regole" da seguire per ottenere un determinato risultato.

I primi compilatori introdotti negli anni '50 erano pensati per offrire set di istruzioni più ricchi di quelli forniti dalle macchine native, ma ciò comportava spesso che il risultato fosse più lento rispetto a quanto ottenuto scrivendo il codice assembly a mano; si pensò dunque di investire molto più tempo e risorse nell'ingegnerizzazione di procedure per migliorare la qualità del codice: da ciò sono nate diverse tecniche di ottimizzazione.

Col tempo molte di queste tecniche furono implementate per i diversi compilatori, ma ci si rese ben presto conto del fatto che vi era un problema da risolvere: le varie tecniche, durante il processo di ottimizzazione, interagivano fra loro; poiché ognuna di esse applicava diverse trasformazioni al codice al fine di migliorarne l'efficienza, non era raro che alcune di queste modifiche andassero a collidere, generando codice poco ottimizzato e vanificando gli sforzi del compilatore.

Risulta chiaro che, al fine di ottenere un discreto livello di efficienza, si debba scegliere quali modifiche applicare e quali no: le varie tecniche di ottimizzazione utilizzano delle risorse in comune (es. registri della macchina) e necessitano ognuna di specifiche condizioni nel codice per essere applicate; è proprio grazie ai flag che possiamo decidere come gestire tali risorse e dunque quali ottimizzazioni applicare.

Inoltre occorre far presente che è significativo anche l'ordine con il quale vengono applicate le ottimizzazioni: come già spiegato, durante il processo di compilazione il codice viene modificato, quindi cambiare l'ordine di queste operazioni significa che lo stesso processo di ottimizzazione verrà applicato ad istruzioni diverse.

Vediamo adesso alcuni esempi di questi flag e i loro effetti sul codice generato:

- **O1**: ottimizza; questo è il primo livello di ottimizzazione, quindi può richiedere tempo e molta memoria per funzioni particolarmente grandi
- **O2**: ottimizza di più; applica tutte le ottimizzazioni previste da O1 più altri flag aggiuntivi, migliorando il tempo di compilazione e le performance finali
- **O3**: massimo livello di ottimizzazione; applica tutti i flag applicati da O2 e altri aggiuntivi
- **Os**: ottimizza le dimensioni; applica le ottimizzazioni di O2 tranne quelle che portano ad un significativo aumento delle dimensioni del codice
- **Ofast**: ottimizza la velocità; applica tutte le ottimizzazioni di O3 più alcune aggiuntive non standard al fine di ottenere la massima velocità
- **Og**: ottimizza l'esperienza di debug; paragonabile al flag di default O0 (nessuna ottimizzazione), ma con l'implementazione di funzionalità utili in fase di debug del codice.

Ovviamente questi sono solo alcuni degli innumerevoli flag a disposizione del programmatore, tuttavia ci permettono di capire quanto sia vasto lo spazio di esplorazione.

Specificato cosa sono e cosa fanno i flag di compilazione, occorre spiegare come possono essere utilizzati mediante **Make**.

Make è un tool per la *build automation*, ossia costruisce programmi eseguibili e librerie dal codice sorgente leggendo dei file chiamati *makefiles*, i quali possono essere visti come delle "ricette" da seguire per ottenere il programma desiderato.

All'interno del makefile troviamo cinque elementi principali:

- *Regole esplicite*: dicono quando e come ricompilare uno o più file
- *Regole implicite*: dicono quando e come ricompilare una classe di file in base al loro nome
- *Definizioni di variabili*: specificano delle variabili, cioè i valori di alcune stringhe di testo che possono essere sostituite in seguito
- *Direttive*: istruzioni che indicano a Make di fare qualcosa di speciale, come leggere altri makefile, decidere se usare o meno alcune parti del makefile stesso ecc.
- *Commenti*: semplice testo per commentare ciò che si sta facendo.

Fra questi elementi, ciò che a noi interessa maggiormente sono le variabili, in particolare **CFLAGS**, la quale contiene tutti i flag di compilazione che devono essere applicati in fase di compilazione di un programma C.

Tale variabile viene prima inizializzata nel seguente modo

$$CFLAGS = [flag\ di\ partenza]$$

dopodiché vi vengono sommati i flag aggiuntivi in questo modo

$$CFLAGS \mathrel{+}= [flag\ aggiuntivi]$$

A questo punto manca solo un aspetto da specificare: come aggiungere i parametri di configurazione da linea di comando.

I makefile, infatti, sono dei file "statici", ovvero vengono scritti, salvati e poi forniti a Make per la compilazione: in questa ottica, se si volesse provare dei parametri

diversi occorrerebbe modificare il makefile a mano ogni volta e ciò comporterebbe notevoli rallentamenti nello sviluppo del software.

Fortunatamente Make supporta il cosiddetto **override** delle variabili, cioè la possibilità di sovrascriverle sul momento quando si lancia la compilazione; basterà dunque lanciare la compilazione col seguente comando

$$make\ CFLAGS\ +=\ [flag\ di\ interesse]$$

per applicare dei parametri aggiuntivi al processo (oppure usare "=" al posto di "+=" se si vuole utilizzare solo ed esclusivamente i flag specificati).

3.4 Numero di core

Ulteriore elemento da esplorare è il numero dei core che vengono utilizzati: utilizzare un elevato numero di core può portare ad un notevole incremento delle prestazioni, tuttavia si deve tener conto ai fattori di non idealità (stalli, accessi alla memoria ecc.) che possono introdurre degli overhead e penalizzare anche in maniera sensibile le performance. Da un punto di vista pratico, il numero di core può essere passato come un ulteriore parametro al comando make e quindi la sua gestione diventa analoga agli altri parametri del compilatore.

In caso di utilizzo di più core viene quindi calcolata una metrica fondamentale, detta **speedup [8]**, mediante la seguente formula:

$$Speedup = \frac{n°\ cicli\ esecuzione\ sequenziale}{n°\ cicli\ esecuzione\ parallela}$$

Questo significa che, per ottenere un significativo incremento delle performance, il numero di cicli per portare a termine tutte le operazioni nel caso multi-core deve essere nettamente inferiore a quelli richiesti nel caso sequenziale (i.e. configurazione a singolo core).

Chiaramente lo speedup ha un limite teorico, il quale è pari al numero di core utilizzati: in altre parole, se si utilizzano N core, l'incremento delle performance potrà essere al massimo di N volte rispetto al caso sequenziale.

3.5 Git e le varianti algoritmiche

Un ulteriore fattore da tenere in conto sono le possibili varianti dello stesso algoritmo: in determinati casi ci possono essere più possibilità di implementazione per risolvere lo stesso problema, da qui nascono le varianti algoritmiche. Tali varianti sono generalmente gestite mediante **Git [16]**, un programma che tiene traccia delle modifiche che vengono apportate ai file e che permette agli sviluppatori di creare più rami di sviluppo (detti **branch**) in modo da tenere le varianti separate ed indipendenti tra loro.

Il motivo principale per cui Git è ampiamente utilizzato è il suo modo di gestire le variazioni dei vari file: a differenza di altri programmi di versionamento, che generalmente si riducono a creare una nuova copia del file ogni volta, Git archivia le varie versioni come **modifiche** del file originale: in pratica è come se a ogni *commit* (salvataggio di versione) il programma scattasse una foto del file in quell'istante e la mettesse a confronto con le foto dei salvataggi precedenti.

In questo modo non si hanno una serie di file copia disgiunti, ma un'intera catena di *snapshots* grazie ai quali si possono ripercorrere le modifiche effettuate: nel caso in cui volessimo tornare ad una versione precedente, Git non andrà a rimpiazzare del tutto i file, ma semplicemente li confronterà con i dati della versione scelta e applicherà le modifiche necessarie.

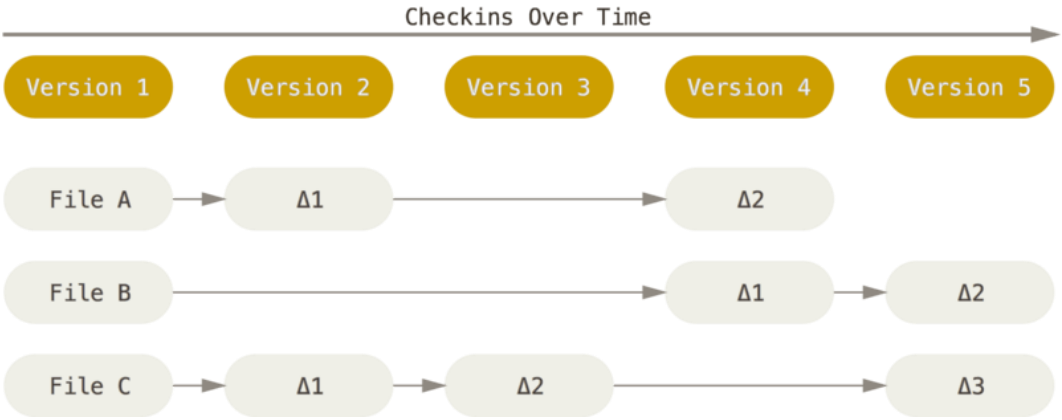

Figura 5 - Gestione delle versioni come modifiche del file di origine

Come accennato all'inizio del paragrafo, ci sono dei casi in cui lo sviluppatore ha bisogno di testare varianti dello stesso algoritmo e i motivi possono essere molteplici: per esempio potrebbe star sviluppando un programma in maniera sequenziale, funzionante, ma decide che vuole provare anche ad implementare l'esecuzione parallela, oppure sta sviluppando il programma in modo da usare il minimo di risorse possibile e decide che vuole creare una variante che invece ha performance nettamente superiore senza badare alle risorse utilizzate.

Qui entra in gioco la funzionalità di **branching** di Git: il tool offre infatti la possibilità di dividere il percorso di sviluppo in più rami e scegliere in quali di questi salvare le modifiche; in questo modo si possono fare tutte le prove che si vogliono senza rischiare di corrompere i file originali con modifiche non soddisfacenti.

Una volta ottenuti i risultati desiderati, si può scegliere se mantenere i rami separati oppure eseguire l'operazione di **merge** per riunirli; se invece i risultati non sono soddisfacenti, si può eliminare il branch in questione oppure eseguire il **rollback** per tornare ad una versione stabile.

Insieme al branching è presente anche la funzionalità di **tagging**, che permette allo sviluppatore di assegnare dei tag, appunto, ai singoli commit: questo può essere particolarmente utile per identificare, per esempio, le versioni principali del programma sviluppato, oppure per identificare la presenza di bug e altri problemi.

Prima di passare al capitolo successivo, occorre evidenziare un aspetto: al momento della stesura di questo elaborato, il framework sviluppato non gestisce la sincronizzazione dei file utilizzati per i test, per cui sta all'utente tenere questi aggiornati per assicurarsi il corretto funzionamento.

Tuttavia esistono tool di **continuous integration**, ossia dei software che più volte al giorno controllano le modifiche fatte ai file dai vari utenti che ci lavorano sopra e vanno a integrarle in maniera automatica in una linea di sviluppo principale: una funzionalità che potrebbe essere aggiunta in futuro nel framework è proprio l'integrazione con uno di questi tool, in modo che l'utente non si debba preoccupare di tenere aggiornati i file in quanto è il framework stesso che lo fa in maniera automatica.

4 Progettazione del tool di benchmarking

Il tool in questione è stato realizzato mediante il linguaggio Python (versione 3.7), poichè si tratta di un linguaggio molto versatile e che supporta una vasta gamma di librerie.

In particolare le librerie utilizzate sono:

- **Subprocess**: utilizzata per lanciare shell dal programma
- **Matplotlib**: utilizzata per la rappresentazione grafica dei risultati
- **Regex**: utilizzata per la gestione e separazione delle stringhe
- **Json**: utilizzata per leggere ed elaborare file JSON
- **Optparse**: utilizzata per permettere al tool di ricevere file di ingresso dal terminale.

4.1 Formato di input

Per specificare al tool quali test eseguire e dove trovare tutti i file e le informazioni necessarie, si deve fornire come input un file **JSON**.
I file JSON hanno un formato che prevede, all'interno di parentesi graffe, delle coppie chiave-valore separate da virgole; quando si legge il file, si specifica la chiave e si ottiene il relativo valore.

La struttura è quindi la seguente:

{"chiave1":"valore1", "chiave2":"valore2"...}

Nel caso particolare di questo tool, il formato del JSON prevede i seguenti campi:

- ***configuration_name***: nome della configurazione, usato come titolo
- ***makefile_path***: percorso del makefile, ossia la directory in cui richiamare il make per la compilazione
- ***platform***: percorso del file (o dei file) per la configurazione della virtual platform da testare

- ***toolchain***: percorso del file di configurazione della toolchain
- ***comparison_type***: tipo di confronto da eseguire (cores/config/platform)
- ***compilation_parameters***: insieme dei parametri di compilazione da passare al make, separati da virgole per ogni configurazione
- ***freq***: la frequenza di clock, espressa in MHz, del processore
- ***active_en***: energia attiva, ossia l'energia consumata dai vari core durante l'esecuzione delle operazioni, espressa in pJ
- ***idle_en***: energia in idle, cioè l'energia consumata dai core quando sono a riposo in attesa di eseguire le istruzioni
- ***uncore_en***: energia fuori dai core, ossia i consumi del resto della scheda
- ***operation_name***: nome dell'operazione da eseguire (es. MAC).


```
{"configuration_name":"Kernel type: Convolution, precision: 888",
    "makefile_path":"/home/dariolinux/Desktop/Tesi/pulp-nn/mixed/XpulpV2/32bit/test",
    "platform":"/home/dariolinux/Desktop/Tesi/pulp-sdk/configs/pulp-open.sh",
    "toolchain":"/home/dariolinux/Desktop/Tesi/v1.0.16-pulp-riscv-gcc-ubuntu-18/sourceme.sh",
    "comparison_type":"cores",
    "compilation_parameters":["perf=1 cores=1 kernel=888", "perf=1 cores=2 kernel=888", "perf=1 cores=4 kernel=888", "perf=1 cores=8 kernel=888"],
    "freq":450,
    "active_en":12,
    "idle_en":2,
    "uncore_en":8,
    "operation_name":"MAC"
}
```


Figura 6 - Esempio di file JSON di configurazione

4.2 Formato di output

Affinché venga eseguita correttamente la procedura di **parsing** (i.e. l'estrazione dei singoli dati dalla stringa di uscita dell'esecuzione dei file), l'output deve avere un formato standard e fornire questi risultati:

- ***num_cycles***
- ***num_instr_miss***
- ***num_ext_load***
- ***num_ext_Store***
- ***num_tcdm_contentions***
- ***num_instrs***
- ***num_active_cycles***
- ***num_load_stalls***

- ***num_jumpr_stalls***
- ***num_branch***
- ***errors***

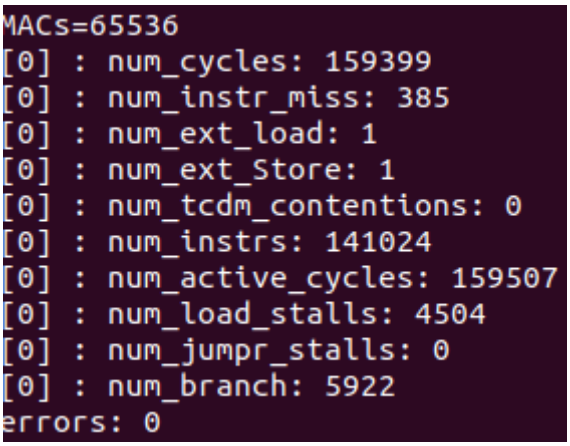

Figura 7 - Esempio di output (1 core)

Questo è l'output nel caso di una configurazione a singolo core (lo zero fra parentesi quadre indica infatti l'id del core, che essendo solo uno è 0), mentre nel caso multi-core si avrebbe lo stesso insieme di risultati ma in numero proporzionale al numero di core.

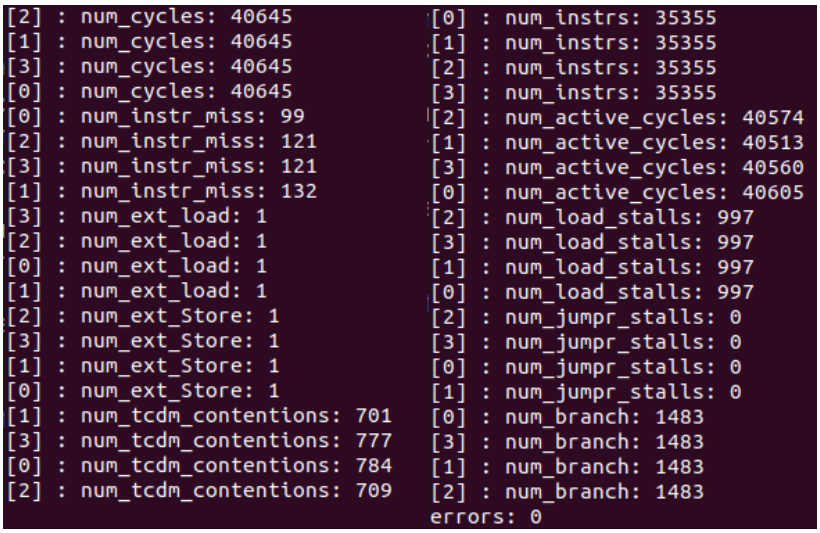

Figura 8 - Esempio di output (4 core)

4.3 Flow di esecuzione

Il framework realizzato è composto da:

- ***multimake_manager.py***, lo script Python principale che prende in ingresso il file JSON di configurazione e poi richiama un altro script in base al tipo di confronto da eseguire
- ***multimake_cores.py***, script utilizzato per confrontare lo stesso programma eseguito su un numero diverso di core
- ***multimake_config.py***, script utilizzato per confrontare possibili variazioni del codice e/o dei flag del compilatore a parità di numero di core
- ***multimake_platform.py***, script utilizzato per confrontare l'esecuzione su diverse virtual platform.

Una volta preparati i file da compilare ed il/i JSON di configurazione, il programma viene lanciato tramite il comando da terminale

python3 multimake_manager.py -f [nome].json

dove [nome].json è il file JSON di configurazione.
Lo script legge il file ed identifica il campo "*comparison_type*" al fine di capire quale confronto deve essere eseguite, dopodiché passerà il JSON allo script necessario (es. *multimake_cores* se si vuole confrontare numeri diversi di core).

A questo punto parte l'esecuzione di questo secondo script.
Nella primissima fase vengono estratti dal JSON i parametri necessari alla compilazione, ossia il percorso del makefile e i parametri delle singole configurazioni; successivamente viene lanciata la compilazione vera e propria mediante la funzione *run* del modulo *subprocess*, il quale output viene catturato e salvato in una variabile.

Questo output viene quindi sottoposto alla procedura di **parsing**, mediante la quale si va ad estrarre dalla stringa i risultati dell'esecuzione del file compilato (risultati che devono essere presentati come indicato nel paragrafo precedente).

I dati ottenuti vengono inseriti in una struttura dati nota come **dictionary**, in particolare il dictionary definito nel programma prende il nome di *resultlist*.

La struttura di un dictionary è molto simile a quella di un file JSON: ogni blocco è composto da una **chiave**, utilizzata per l'identificazione del suddetto, e da un **valore**, ossia il contenuto di esso; occorre specificare che con il termine "valore" non si intende per forza un singolo elemento (es. numero intero), ma può essere anche associato ad un insieme di valori (ciò che in Python è definito come **lista**).

Di seguito si può osservare un esempio del dictionary implementato nel framework: fra parentesi graffe sono contenuti i risultati della singola configurazione testata, all'interno di queste si possono notare le chiavi scritte fra apici ed il rispettivo valore preceduto dai i due punti; come si può notare, in diversi casi il valore non è un solo elemento, ma più elementi raccolti fra parentesi quadre e separate tra virgole.

Per esempio, *'num_cycles': [79923, 79923]* raccoglie il numero di cicli eseguiti da ogni core nella configurazione a 2 core, di conseguenza si hanno due valori anziché uno solo.


```
[{'op': 65536, 'num_cycles': [159399], 'num_instr_miss': [385], 'num_ext_load':
[1], 'num_ext_Store': [1], 'num_tcdm_contentions': [0], 'num_instrs': [141024],
'num_active_cycles': [159507], 'num_load_stalls': [4504], 'num_jumpr_stalls': [0
], 'num_branch': [5922], 'errors': 0}, {'op': 65536, 'num_cycles': [79923, 79923
], 'num_instr_miss': [143, 143], 'num_ext_load': [1, 1], 'num_ext_Store': [1, 1]
, 'num_tcdm_contentions': [460, 448], 'num_instrs': [70526, 70526], 'num_active_
cycles': [79697, 79713], 'num_load_stalls': [1993, 1993], 'num_jumpr_stalls': [0
, 0], 'num_branch': [2963, 2963], 'errors': 0}, {'op': 65536, 'num_cycles': [406
45, 40645, 40645, 40645], 'num_instr_miss': [99, 121, 121, 132], 'num_ext_load':
 [1, 1, 1, 1], 'num_ext_Store': [1, 1, 1, 1], 'num_tcdm_contentions': [701, 777,
 784, 709], 'num_instrs': [35355, 35355, 35355, 35355], 'num_active_cycles': [40
574, 40513, 40560, 40605], 'num_load_stalls': [997, 997, 997, 997], 'num_jumpr_s
talls': [0, 0, 0, 0], 'num_branch': [1483, 1483, 1483, 1483], 'errors': 0}, {'op
': 65536, 'num_cycles': [21231, 21231, 21231, 21231, 21231, 21231, 21231, 21231]
, 'num_instr_miss': [77, 22, 55, 55, 88, 55, 33, 55], 'num_ext_load': [1, 1, 1,
1, 1, 1, 1, 1], 'num_ext_Store': [1, 1, 1, 1, 1, 1, 1, 1], 'num_tcdm_contentions
': [1098, 1124, 1049, 1069, 1113, 1055, 1179, 1195], 'num_instrs': [17769, 17769
, 17769, 17769, 17769, 17769, 17769, 17769], 'num_active_cycles': [21134, 21244,
 21100, 21035, 21292, 21258, 21231, 21150], 'num_load_stalls': [564, 564, 564, 5
64, 564, 564, 564, 564], 'num_jumpr_stalls': [0, 0, 0, 0, 0, 0, 0, 0], 'num_bran
ch': [743, 743, 743, 743, 743, 743, 743, 743], 'errors': 0}]
```


Figura 9 - Esempio di resultlist, il dictionary implementato nel framework

Fatto questo, lo script procede con il calcolo dei risultati indiretti, ossia quei parametri che non sono forniti direttamente come output dall'esecuzione del

programma compilato ma che si ottengono dai dati contenuti nel dictionary e nel JSON.

I parametri in questione sono:

- ***speedup***: (solo nel caso di confronto fra più configurazioni di cores) indica l'aumento di performance rispetto all'esecuzione sequenziale.
  Come già mostrato in precedenza, la formula per il suo calcolo è:

$$Speedup = \frac{n° \; cicli \; esecuzione \; sequenziale}{n° \; cicli \; esecuzione \; parallela}$$

- ***power consumption***: indica il consumo di potenza della configurazione (espresso in mW) e viene calcolato nel seguente modo:

$$\frac{\sum_i^N \{active_{en} \times active_{cycles[core_i]} + idle_{en} \times idle_{cycles[core_i]}\} + uncore_{en} \times cycles \times 10^{-9}}{\frac{cycles \times 10^6}{freq}}$$

  dove N è il numero di core della configurazione, mentre gli altri parametri sono già stati trattati nella sezione relativa al formato di input

- ***op/cycle***: indica quante operazioni di un certo tipo (es. MAC, ossia multiply-accumulate) sono state eseguite per ogni ciclo.

A questo punto lo script passa all'esecuzione dell'ultima sezione di codice, nella quale utilizza i dati raccolti per preparare dei grafici e li mostra a schermo.

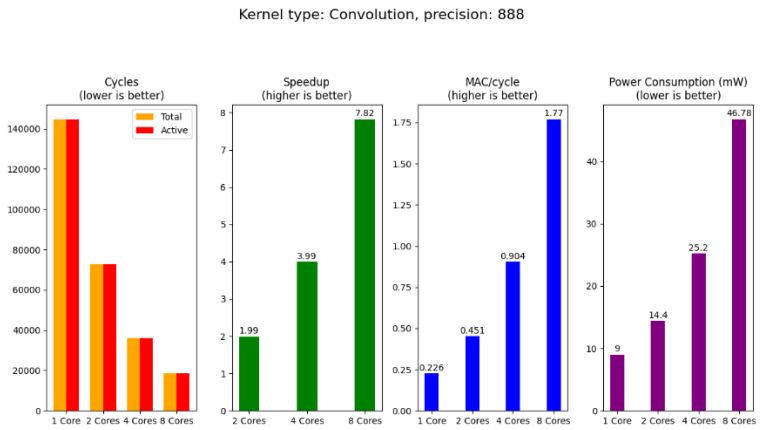

Figura 10 - Grafici a barre che mostrano le performance

La prima figura che viene mostrata contiene dei grafici a barre indicanti le performance delle diverse configurazioni, gli elementi da notare sono:

- titolo della simulazione, centrato in alto
- titolo del singolo grafico, dove viene riportato *higher/lower is better* per indicare se si vuole ottenere un risultato più alto o più basso possibile
- nome della configurazione, sull'asse delle ascisse, per ogni barra
- valore delle barre, sull'asse delle ordinate.

A titolo di esempio, si vede che il consumo di potenza (grafico con le barre viola) deve essere il più basso possibile, quindi in questo caso la configurazione ad un solo core è quella che ottiene il risultato migliore; se osserviamo però il grafico delle operazioni per ciclo (barre verdi) notiamo che questa stessa configurazione è quella che ottiene il risultato peggiore, in quanto in questo caso il valore deve essere più alto possibile.

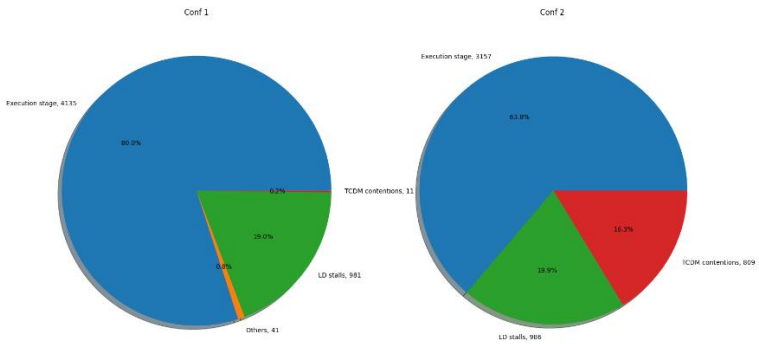

Figura 11 - Grafici a torta che mostrano gli overhead

La seconda figura che viene poi mostrata rappresenta invece dei grafici a torta con i quali si vogliono far vedere i fattori di overhead, ossia le cause che portano ad avere performance peggiori rispetto a quelle attese.

In questi grafici la parte di *execution stage*, rappresentata in blu, indica la percentuale di cicli durante i quali sono state effettivamente eseguite delle istruzioni, quindi i risultati sono tanto migliori quanto più ampia è questa porzione di grafico; al contrario, tutte le altre parti rappresentano dei fattori negativi che rallentano l'esecuzione del programma, quindi si vuole cercare di minimizzarle.

5 Risultati sperimentali

Per testare il corretto funzionamento del tool sono stati fatti diversi esperimenti.

5.1 Esperimento 1

Repository GIT: GitHub - pulp-platform/pulp-nn

Il primo esperimento, particolarmente interessante, riguarda **pulp-nn**, ossia un insieme di programmi per la generazione di reti neurali.

In questo caso viene utilizzato lo script ***pulp_nn_test_setup.py*** per generare la rete neurale di nostro interesse e poi il framework esegue la compilazione mediante il makefile come spiegato precedentemente.
La rete neurale generata in questo esempio ha le seguenti caratteristiche:

- ***ISA***: XpulpNN
- ***Tipo di kernel***: convolution
- ***Input activations precision***: 8
- ***Output activations precision***: 8
- ***Weight precision***: 8
- ***Tipo di quantizzazione***: shift clip

Il file JSON di configurazione è:

```
{"configuration_name":"Kernel type: Convolution, precision: 888",
    "makefile_path":"/home/dariolinux/Desktop/Tesi/pulp-nn/mixed/XpulpV2/32bit/test",
    "platform":"/home/dariolinux/Desktop/Tesi/pulp-sdk/configs/pulp-open.sh",
    "toolchain":"/home/dariolinux/Desktop/Tesi/v1.0.16-pulp-riscv-gcc-ubuntu-18/sourceme.sh",
    "comparison_type":"cores",
    "compilation_parameters":["perf=1 cores=1 kernel=888", "perf=1 cores=2 kernel=888", "perf=1 cores=4 kernel=888", "perf=1 cores=8 kernel=888"],
    "freq":450,
    "active_en":12,
    "idle_en":2,
    "uncore_en":8,
    "operation_name":"MAC"
}
```

Figura 12 - JSON di configurazione per Esempio 1

I risultati delle performance sono i seguenti:

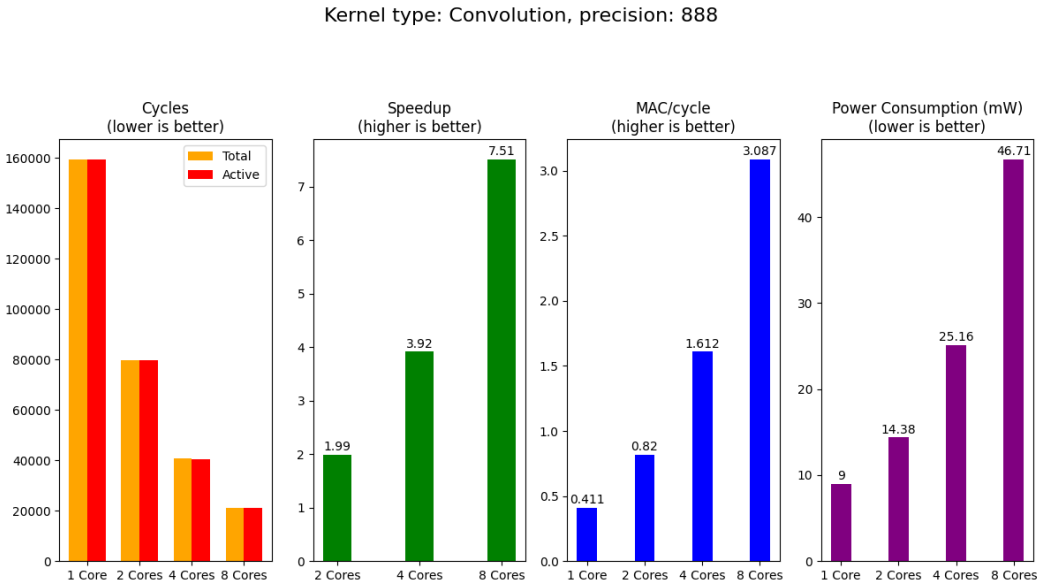

Figura 13 - Risultati di performance di Esperimento 1

Dal grafico si possono notare alcuni aspetti chiave per questo esperimento in particolare:

- Il numero di cicli totali e di quelli attivi è circa uguale, quindi i cicli di idle sono ridotti al minimo; questo significa che in tutti i casi i core vengono sfruttati in maniera ottimale
- Lo speedup (incremento di performance rispetto al caso single-core) è molto buono, ma si può notare come all'aumentare del numero di core questo si vada allontanando dal risultato ideale; difatti, se con N core ci aspettiamo un aumento teorico di performance di N volte rispetto al caso sequenziale, si può vedere bene che questo risultato diventi sempre più lontano man mano che il numero di core aumenta. Un risultato di questo tipo può indicare al programmatore che è necessario effettuare un'analisi più approfondita sul codice, a partire da uno studio dei fattori di overhead (descritta nel seguito)
- Op/cycle e il consumo di potenza mostrano risultati in linea con quanto ci si potesse aspettare, quindi un aumento lineare con il numero di core.

Adesso vediamo invece quanti e quali sono i fattori di overhead che non permettono il raggiungimento dei limiti prestazionali teorici.

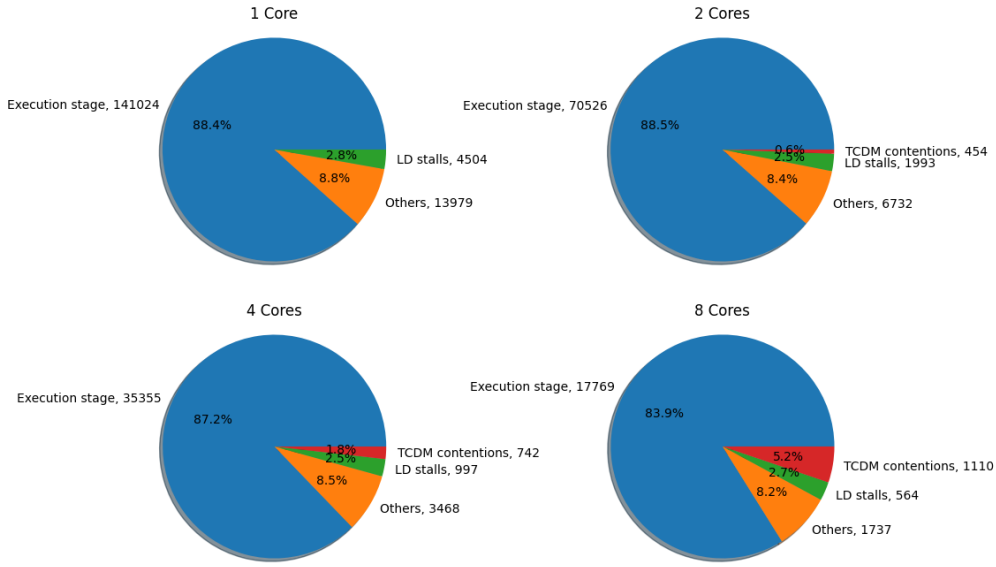

Figura 14 - Risultati di overhead di Esempio 1

Il primo aspetto che possiamo notare è che nel caso a singolo core non è presente l'elemento **TCDM contentions**: questo perché le contese della memoria avvengono quando un core cerca di accedere ad essa ma la trova già "occupata" da un altro core; nel caso di un solo core, quindi, non si possono verificare tali contese.

Man mano che il numero di core aumenta, le contese aumentano in maniera sempre maggiore allontanando i risultati ottenuti da quelli ideali.

Nonostante gli altri fattori di overhead diminuiscano, le contese della memoria diventano sempre più preponderanti: infatti, se si considera la percentuale di **execution stage** (cioè il numero di cicli attivi di esecuzione), questa diventa sempre inferiore rispetto ai casi di configurazioni con numero minore di core.

Volendo trovare una soluzione al problema, si può ipotizzare che ottimizzando maggiormente il codice si potrebbe fare in modo che siano necessari molti meno accessi alla memoria, riducendo così le contese e di conseguenza aumentando le prestazioni.

5.2 Esperimento 2

Repository GIT: GitHub - gtagliavini/MatrixMultiplicationPULP at main

Il secondo esperimento riguarda invece il programma MatMul, che utilizza la piattaforma PULP per eseguire operazioni matriciali, in particolare la moltiplicazione fra matrici; tale programma prevede due varianti: una con ***unrolling*** ed una senza.

L'unrolling non è altro che un'operazione che permette di "srotolare" i cicli di istruzioni al fine di ottenere del codice più parallelizzabile (e quindi sfruttare meglio un elevato numero di core).

In questo esempio viene confrontato il caso senza unrolling (configurazione 1) con il caso unrolling = 4 (configurazione 2).

Il file JSON di configurazione è:


```
{"configuration_name":"Matrix Multiplication",
    "makefile_path":["/home/dariolinux/Desktop/Tesi/MatMul/branch_main/MatrixMultiplicationPULP",
        "/home/dariolinux/Desktop/Tesi/MatMul/branch_unrolling/MatrixMultiplicationPULP"],
    "platform_path":"/home/dariolinux/Desktop/Tesi/pulp-sdk/configs/",
    "platform":"pulp-open.sh",
    "toolchain":"/home/dariolinux/Desktop/Tesi/v1.0.16-pulp-riscv-gcc-ubuntu-18/sourceme.sh",
    "comparison_type":"config",
    "compilation_parameters":["CORES=8"],
    "freq":450,
    "active_en":12,
    "idle_en":2,
    "uncore_en":8,
    "operation_name":"MACs"
}
```


Figura 15 - JSON di configurazione per Esperimento 2

I risultati delle performance sono i seguenti:

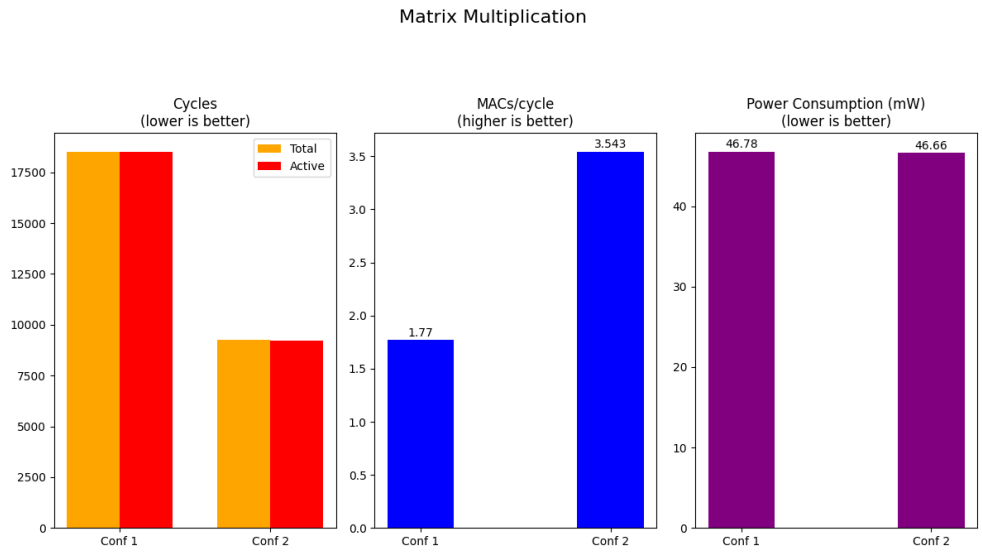

Figura 16 - Risultati di performance di Esperimento 2

Come si può notare, l'introduzione dell'unrolling ha permesso di sfruttare maggiormente il numero di core a disposizione (8, in questo caso), ottenendo un notevole incremento delle performance mentre il consumo energetico è rimasto praticamente invariato.

Anche dal punto di vista degli overhead c'è un netto miglioramento:

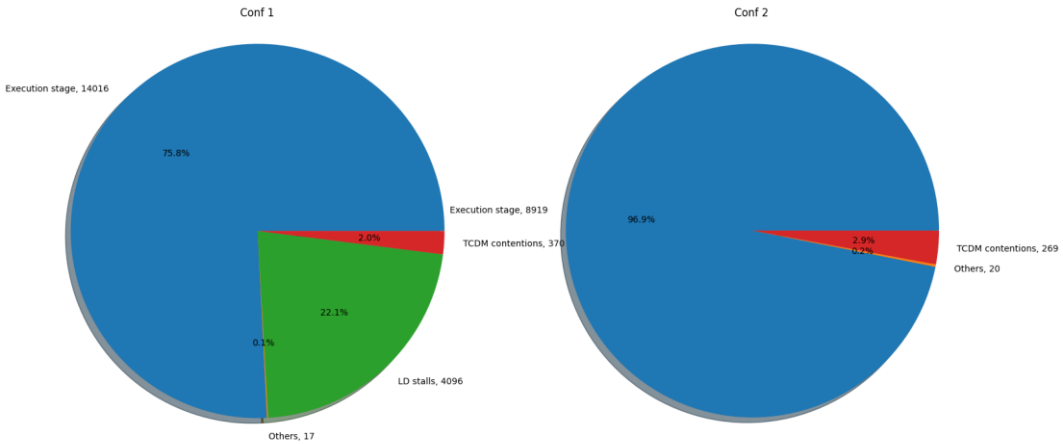

Figura 17 - Risultati di overhead di Esperimento 2

A fronte di un trascurabile incremento delle contese della memoria, la percentuale di *execution stage* è sensibilmente aumentata grazie all'utilizzo dell'unrolling.

5.3 Esperimento 3

Repository GIT: https://git.eees.dei.unibo.it/enrico/ml_nl_parallel/-/tree/master/svm/gap8/1vsAll/hor/single_buffer/parallel

Il terzo esperimento riguarda il programma SVM, che utilizza la piattaforma PULP per eseguire operazioni legate alle **Support-Vector Machines**, dei modelli di apprendimento supervisionato associati ad algoritmi di apprendimento per la regressione e la classificazione.

In questo caso il numero di core è fissato ed è pari a 8, la differenza fra le due configurazioni risiede nei *flag* del compilatore: nel primo caso viene specificato soltanto il flag **-O3** che riguarda il livello di ottimizzazione, nel secondo invece viene anche specificato il flag **-flto**, il quale richiede al compilatore di eseguire delle ottimizzazioni a *link-time*, cioè nel momento in cui va a mettere insieme i vari file oggetto.

In questo caso, però, i vari file sorgente sono già stati convertiti in codice macchina (generando appunto i file oggetto), per cui il linker riesce a vedere il programma nella sua interezza e ad applicare delle ottimizzazioni; ciò è veramente difficile e non sempre i risultati ottenuti sono soddisfacenti: come già spiegato per i flag di compilazione, c'è il rischio che si generino dei conflitti che portano a un peggioramento dei risultati.

Il file JSON di configurazione è:


```
{"configuration_name":"PMSIS SVM 1vsAll Sequential",
    "makefile_path":"/home/dariolinux/Desktop/Tesi/examples/svm_parallel/svm/gap8/1vsAll/hor/single_buffer/parallel",
    "platform_path":"/home/dariolinux/Desktop/Tesi/pulp-sdk/configs/",
    "platform":"pulp-open.sh",
    "toolchain":"/home/dariolinux/Desktop/Tesi/v1.0.16-pulp-riscv-gcc-ubuntu-18/sourceme.sh",
    "comparison_type":"config",
    "compilation_parameters":["FLAGS=\"-O3\"", "FLAGS=\"-O3 -flto\""],
    "freq":450,
    "active_en":12,
    "idle_en":2,
    "uncore_en":8,
    "operation_name":""
}
```


Figura 18 - JSON di configurazione per Esperimento 3

I risultati delle performance sono i seguenti:

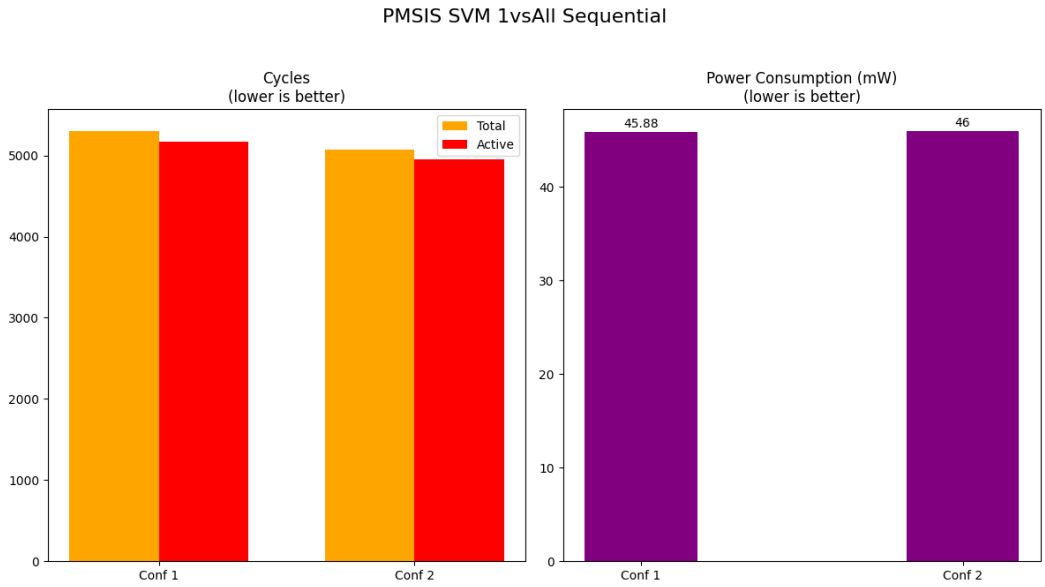

Figura 19 - Risultati di performance di Esperimento 3

In questo caso l'applicazione del flag aggiuntivo non ha avuto degli effetti apprezzabili, semplicemente il numero di cicli totali e attivi è leggermente più basso (la configurazione 2 è quella con -flto); anche la proporzionalità fra cicli totali e cicli attivi è praticamente la medesima.

Anche in questo caso è la differenza negli overhead ad essere interessante:

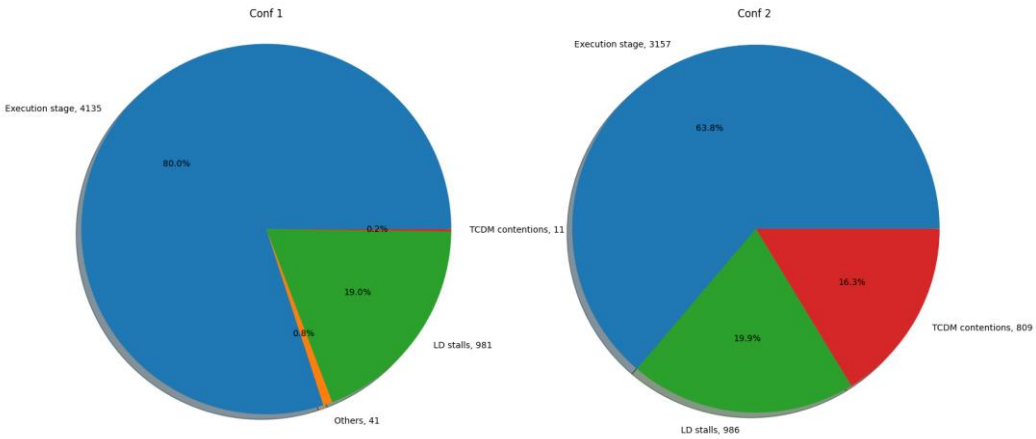

Figura 20 - Risultati di overhead di Esperimento 3

La differenza è netta: l'utilizzo di tale flag non solo non ha apportato dei benefici apprezzabili, ma ha sensibilmente incrementato il numero di contese della memoria; risulta chiaro che in questo caso la configurazione da preferire sia la prima, con l'utilizzo del solo flag di ottimizzazione -O3.

5.4 Esperimento 4

Repository GIT: https://git.eees.dei.unibo.it/enrico/ml_nl_parallel/-/tree/automatic_testing/gaussianNB/gap8/hor/double_buffer/pulp_math/div/parallel

Il quarto esperimento riguarda un programma che implementa i *naive Bayes classifiers*, ossia una famiglia di classificatori probabilistici basati sull'applicazione del teorema di Bayes con forti assunzioni di indipendenza fra le variabili.

Anche in questo caso il numero di core è fissato a 8 e nella seconda configurazione testata viene applicato il flag -flto, ma a differenza dell'esperimento precedente i risultati con l'applicazione di tale flag sono migliorativi.

Il file JSON di configurazione è:


```
{"configuration_name":"PMSIS Parallel Gaussian NB + Double-Buffering",
    "makefile_path":"/home/dariolinux/Desktop/Tesi/examples/gaussianNB_math_parallel/gaussianNB/gap8/hor/double_buffer/pulp_math/div/parallel",
    "platform_path":"/home/dariolinux/Desktop/Tesi/pulp-sdk/configs/",
    "platform":"pulp-open.sh",
    "toolchain":"/home/dariolinux/Desktop/Tesi/v1.0.16-pulp-riscv-gcc-ubuntu-18/sourceme.sh",
    "comparison_type":"config",
    "compilation_parameters":["FLAGS=\"-O3\"", "FLAGS=\"-O3 -flto\""],
    "freq":450,
    "active_en":12,
    "idle_en":2,
    "uncore_en":8,
    "operation_name":""
}
```


Figura 21 - JSON di configurazione per Esperimento 4

I risultati delle performance sono i seguenti:

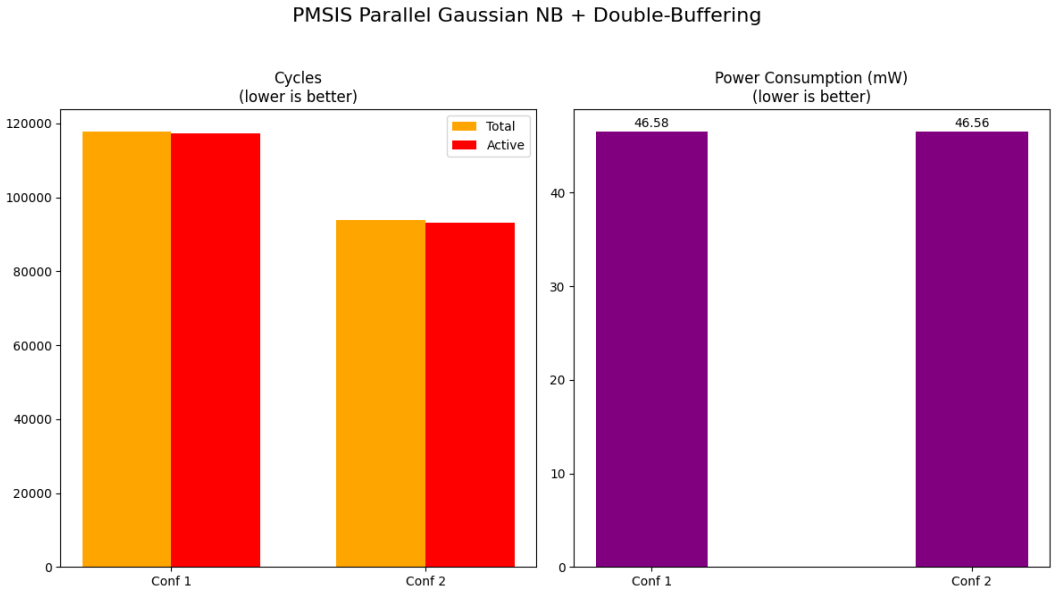

Figura 22 - Risultati di performance di Esperimento 4

Stavolta l'applicazione del flag aggiuntivo ha permesso di ridurre il numero di cicli totali quasi del 20%, portando un evidente miglioramento delle performance.

Osserviamo adesso il risultato degli overhead:

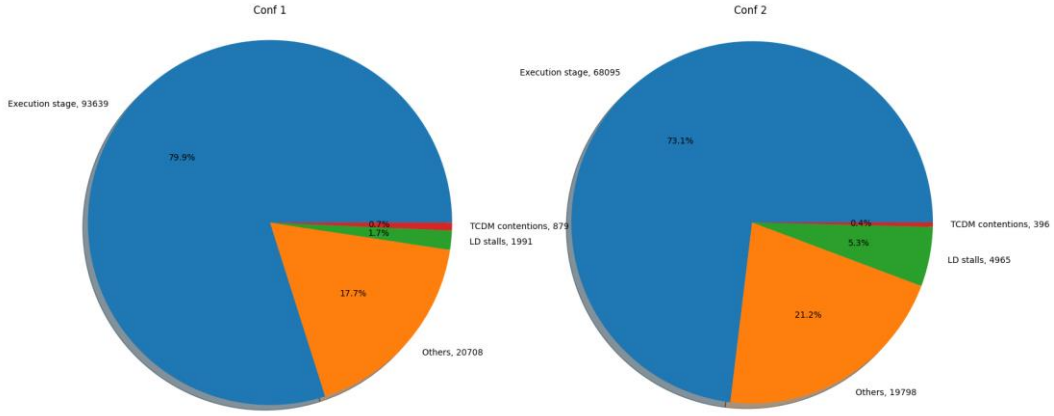

Figura 23 - Risultati di overhead di Esperimento 4

Come si può notare dalla figura, utilizzando il flag -flto la situazione degli overhead è leggermente peggiorata: da questi risultati possiamo capire che, se si riuscisse a ridurre ulteriormente gli overhead della seconda configurazione, si potrebbe ottenere un incremento prestazionale molto migliore rispetto al 20% ottenuto in questo caso.

6 Conclusioni

In conclusione possiamo affermare che il framework realizzato sia uno strumento molto potente per gli sviluppatori, soprattutto per la velocità con cui permette di eseguire i confronti; in un mercato sempre più competitivo è essenziale ridurre al minimo il *time-to-market*, cioè li tempo che passa dall'inizio dello sviluppo al rilascio del prodotto sul mercato, e tale strumento è sicuramente di grande aiuto.

Altro fattore fondamentale è la sua flessibilità verso le piattaforme supportate: al momento del suo rilascio sono già presenti diverse configurazioni, ma il settore è in forte sviluppo e in breve tempo se ne aggiungeranno molte altre; per questo motivo il framework è stato sviluppato con un approccio modulare e *general purpose*.

Una buona parte dei parametri sono forniti dal file JSON di configurazione e vengono quindi gestiti a run-time, permettendo quindi all'utilizzatore di specificare la configurazione di interesse senza particolari vincoli; l'approccio modulare permette poi, nel caso in cui ce ne sia bisogno, di creare nuovi script ad hoc ed aggiungerli al framework con facilità, aprendo le porte a nuovi tipi di test ma senza minare la compatibilità con quanto realizzato in precedenza.

Ovviamente non sarà solo il settore dell'hardware ad evolversi, ma anche quello del software andrà di pari passo: ciò che è stato qui descritto non è altro che la versione 1.0 del programma, ma trattandosi di un progetto open-source col tempo si andrà a espandere, passando tra le mani di diversi sviluppatori ed aggiungendo un numero molto elevato di funzioni.

Risulta difficile prevedere quali saranno gli sviluppi futuri dell'applicazione, in quanto di funzionalità interessanti da inserire ce ne sarebbero in quantità: un esempio potrebbe essere l'implementazione dell'analisi parallela, con la quale si andrebbe a ridurre il tempo necessario per i testi (al momento le configurazioni vengono simulate in maniera sequenziale, per cui il tempo di test è pari al tempo necessario per eseguire la singola configurazione moltiplicato per il numero di

configurazioni da testare); altra funzione utile potrebbe essere l'aggiunta di opzioni a run-time, per cui il test viene lanciato partendo dal JSON ma poi altre opzioni possono essere inviate da terminale e aggiunte dinamicamente al test, variando i risultati visualizzati sui grafici in tempo reale.

7 Riferimenti

[1] Davide Rossi, Francesco Conti, Andrea Marongiu, Antonio Pullini, Igor Loi, Michael Gautschi, Giuseppe Tagliavini, Alessandro Capotondi, Philippe Flatresse, Luca Benini. *PULP: A parallel ultra low power platform for next generation IoT applications*. IEEE Hot Chips 27 Symposium (HCS), pp. 1-39. IEEE, 2015.

[2] Waterman, Andrew Shell. *Design of the RISC-V instruction set architecture*. University of California, Berkeley, 2016.

[3] Pasquale Schiavone, Davide Rossi, Antonio Pullini, Alfio Di Mauro, Francesco Conti, Luca Benini. *Quentin: an ultra-low-power pulpissimo soc in 22nm fdx*. IEEE SOI-3D-Subthreshold Microelectronics Technology Unified Conference (S3S), pp. 1-3. IEEE, 2018.

[4] Weaver, Vincent M., and Sally A. McKee. *Can hardware performance counters be trusted?*. IEEE International Symposium on Workload Characterization, pp. 141-150. IEEE, 2008.

[5] GitHub GVSOC Repository: https://github.com/pulp-platform/gvsoc

[6] Akram, Ayaz, Lina Sawalha. *A survey of computer architecture simulation techniques and tools*. IEEE Access 7 (2019): 78120-78145.

[7] Crockford, Douglas. *ECMA-404 The JSON Data Interchange Standard*. 2017.

[8] Hill, Mark D., Michael R. Marty. *Amdahl's law in the multicore era*. Computer 41, no. 7 (2008): 33-38.

[9] Arm Cortex-M4 IP: https://www.arm.com/products/silicon-ip-cpu/cortex-m/cortex-m4

[10] Antonio Pullini, Davide Rossi, Germain Haugou, Luca Benini. *µDMA: An autonomous I/O subsystem for IoT end-nodes*. 27th International Symposium on Power and Timing Modeling, Optimization and Simulation (PATMOS), pp. 1-8. IEEE, 2017.

[11] GitHub pulp-sdk Repository: https://github.com/pulp-platform/pulp-sdk

[12] Javier Diaz, Camelia Munoz-Caro, Alfonso Nino. *A survey of parallel programming models and tools in the multi and many-core era*. IEEE Transactions on parallel and distributed systems 23, no. 8 (2012): 1369-1386.

[13] Dagum, Leonardo, and Ramesh Menon. *OpenMP: an industry standard API for shared-memory programming*. IEEE computational science and engineering 5, no. 1 (1998): 46-55.

[14] GCC - the GNU Compiler Collection: https://gcc.gnu.org/

[15] The LLVM Compiler Infrastructure: https://llvm.org/

[16] Jon Loeliger, Matthew McCullough. *Version Control with Git: Powerful tools and techniques for collaborative software development*. O'Reilly Media, Inc., 2012

8 Ringraziamenti

Questa avventura è giunta finalmente al termine e, se devo essere onesto, non credevo che sarei riuscito ad arrivare fino in fondo.
Arrivare alla fine non è stato facile, anche perché nel mentre non ci siamo certo fatti mancare una bella pandemia che lascerà un segno indelebile nelle nostre vite, ma ce l'ho fatta: questa tesi è la chiave che chiude la porta del mio percorso accademico, è ciò che segna il mio salto di qualità, che identifica il passaggio dai miei studi al primo giorno del resto della mia vita.

Una tesi che è stata capitanata da quel fantastico relatore che è stato Giuseppe, sempre gentile e disponibile e che, come un angelo custode, ha vegliato sul mio percorso dandomi dritte e rincuorandomi dei miei sforzi.

E saltando su questo ottovolante di emozioni, non posso che partire con i ringraziamenti dalla persona che mi è sempre stata più vicina, che mi ha fatto capire il significato del verbo "amare": la mia dolce metà Sara.
Poteva tremare la terra, eruttare un vulcano, piombare un meteorite sulla Terra, ma lei è sempre stata lì, accanto a me, pronta a darmi una spalla su cui piangere e una spinta per ripartire.
Lo dico senza timore: senza di lei io ora non sarei qui, non sarei l'uomo che sono, ed è per questo che le dedico un posto speciale su questa carta bianca, ma soprattutto nel mio cuore.

Passo poi alla mia famiglia: madre, padre e fratello che hanno dovuto sopportare i miei deliri senza prendermi a schiaffi (anche se me lo sarei meritato), ma che già sanno quanto tengo a loro e quanto voglio loro bene, dato che mi hanno cresciuto con affetto e che mi hanno sempre riportato in carreggiata nei momenti bui.

Poi è il turno della famiglia acquisita, con Gianna, Alice, Samuele e Rosanna: non sapevo cosa mi sarebbe capitato scegliendo la Sarina, ma per fortuna sono cascato in piedi, non avrei potuto chiedere di meglio.

Ovviamente non posso non citare i miei compagni di corso, in particolare Nando, Ale, Riccardo, Marco e Danilo: fra chi mi ha aiutato nello studio, chi mi ha riempito la pancia con quintali di salumi e chi mi ha fatto ridere a crepapelle, mi avete dato la carica per arrivare in fondo.

Un ringraziamento va anche agli amici di giù, ossia Lorenzo, Stefano e Gennaro, i quali sono la prova vivente che le grandi amicizie possono nascere anche da dietro uno schermo: se non esisteste dovrebbero inventarvi, uè.

Chiaramente non mi sono scordato di ringraziare anche gli amici più vicini, quindi Sandro, Samantha e Letizia per le serate passate in compagnia, che mi hanno permesso di svagare la testa di tanto in tanto.

È doveroso ringraziare anche l'Ufficio Tecnico della Everex s.r.l, in particolare il team junior con Martina, Gabriele, Francesco e Sadamal: ci conosciamo da poco, ma sento che è l'inizio di una grande amicizia che ci permetterà di spiccare il volo.

Dell'UT mi preme ringraziare in separata sede Fabiooooh e Marco, i quali sono stati i miei mentori in ambito firmware e software e mi hanno permesso di avere una crescita professionale a livelli esponenziali.

Chiudo la carrellata con un pensiero rivolto ai cari nonno Aldo e nonna Lia: ormai non ci siete più, ma una parte di voi rimarrà sempre con me grazie ai principi che mi avete insegnato.
Umiltà, sacrificio e bontà, vi porto nel cuore.

Grazie di tutto.