



SAPIENZA
UNIVERSITÀ DI ROMA

Caratterizzazione della scheda di comunicazione NaNet e suo utilizzo nel Trigger di Livello 0 basato su GPU dell'esperimento NA62

Facoltà di Scienze Matematiche, Fisiche e Naturali
Corso di Laurea in Fisica

Candidato

Luca Pontisso
Matricola 673468

Relatore

Dr. Alessandro Lonardo

Anno Accademico 2013/2014

Caratterizzazione della scheda di comunicazione NaNet e suo utilizzo nel Trigger di Livello 0 basato su GPU dell'esperimento NA62

Tesi di Laurea. Sapienza – Università di Roma

© 2014 Luca Pontisso. Tutti i diritti riservati

Questa tesi è stata composta con L^AT_EX e la classe Sapthesis.

Email dell'autore: pontissoluca@gmail.com

*Ci sono persone a cui voglio cosmicomicamente bene.
È dedicata a loro.*

Indice

Introduzione	v
1 L'esperienza NA62	1
1.1 Una panoramica	1
1.2 RICH	5
1.2.1 Ricostruzione dei cerchi	5
1.3 TDAQ	6
1.4 Il Trigger di Livello 0	8
1.4.1 Problematrice connesse	8
1.4.2 Un possibile approccio risolutivo: NaNet + GPU	9
2 Considerazioni sulle tecnologie di interesse	11
2.1 Il link dati	11
2.1.1 Socket	11
2.1.2 Lo stack di rete in Linux	12
2.1.3 Protocollo UDP	13
2.2 GPU	14
2.2.1 Limiti delle CPU	14
2.2.2 Una diversa filosofia	15
2.2.3 L'architettura CUDA	16
Single instruction multiple threads	17
Organizzazione della memoria	18
PCI express: collegamento tra host e device	19
Programmazione in un sistema eterogeneo: il CUDA C	20
Kernel	21
Funzioni di base riguardanti la memoria	21
2.2.4 Fermi - Kepler: differenze fra le architetture	21
<i>Fermi</i> (GF100)	22
<i>Kepler</i> (GK110)	23
2.3 Alcune considerazioni su SO real-time	24
3 Configurazione in <i>loopback</i>	26
3.1 Perché in <i>loopback</i> ?	26
3.1.1 Sockperf	26
Loopback: il problema dei pacchetti <i>local</i>	27
3.2 Test con due NIC Ethernet	30

4	Architettura della scheda di rete NaNet	32
4.1	La logica della scheda	32
4.1.1	<i>Interfaccia I/O</i>	32
4.1.2	<i>Router</i>	33
4.1.3	<i>Network interface</i>	34
4.1.4	<i>Micro controller</i>	34
4.1.5	<i>GPU I/O Accelerator</i>	35
4.1.6	La <i>Custom logic</i>	37
4.1.7	Performance	37
	Latenza di attraversamento	37
	Latenza della comunicazione e banda	38
4.1.8	Componenti software	40
	Organizzazione dei buffer di ricezione sulla GPU in una lista circolare	40
	Applicazioni su host e NaNet	40
4.1.9	NaNet: una famiglia di schede	41
5	NaNet come elemento del Trigger di Livello 0	42
5.1	L0TP con NIC ethernet standard	42
5.2	L0TP con NaNet	43
5.2.1	Il formato dati	44
5.2.2	Software utilizzato per i test	44
	Struttura del programma host	44
	Kernel CUDA MATH	45
5.2.3	Misure con oscilloscopio	47
5.2.4	Misure nella configurazione standard del trigger	47
	Conclusioni	50
	Bibliografia	51
	Ringraziamenti	53

Introduzione

Obiettivo dell'esperimento NA62 è sottoporre a verifica le previsioni del Modello Standard riguardante il rapporto di decadimento del processo $K^+ \rightarrow \pi^+ \nu \bar{\nu}$.

Tale valore può essere dedotto dalla teoria con grande precisione. Inoltre, i canali di decadimento sono pochi e la molteplicità degli stati finali è ben conosciuta.

Queste caratteristiche lo rendono un ottimo banco di prova per verificare l'eventuale esistenza di Nuova Fisica.

Il decadimento preso in considerazione è molto raro, rendendo così necessaria la produzione di una grande quantità di kaoni. È richiesta, quindi, un'attenta strategia di selezione degli eventi per raggiungere la sensibilità necessaria e ridurre la mole di dati da trasferire nelle memorie di massa, non essendo la banda disponibile sufficiente a gestire i dati derivanti dall'elevata frequenza di eventi. Un sistema di trigger a più livelli è stato previsto proprio per questo scopo.

Nel Capitolo 1 vengono descritti gli elementi principali dell'apparato sperimentale e il sistema di trigger e acquisizione dati. Una particolare attenzione verrà prestata al rivelatore Ring-Imaging Čerenkov (RICH) per la sua importanza nella catena del trigger di livello 0. Questo è deputato a una prima selezione degli eventi in tempo reale e deve garantire una risposta in meno di 1 ms.

Le latenze di comunicazione (L_{comm}), associate al trasferimento dati dalle schede di Read-Out del RICH alla memoria di un L0 Trigger Processor, e di calcolo (L_{calc}), associate all'elaborazione dell'algoritmo di trigger, devono quindi essere comprese in questa finestra temporale.

Nei trigger di basso livello degli esperimenti di fisica delle alte energie, che richiedono tempi di risposta ridotti e stabili, la parte computazionale è solitamente deputata a Field-Programmable Gate Array (FPGA).

Negli ultimi anni, però, i processori grafici (GPU) hanno visto crescere sempre di più il loro impiego nel calcolo ad alte prestazioni.

Le caratteristiche che hanno favorito questo successo sono: performance molto alte nel calcolo in virgola mobile, facilità di programmazione, rapida evoluzione tecnologica trainata dall'enorme mercato dei videogiochi, costi ridotti grazie all'economia di scala, ottimo rapporto tra il consumo energetico e le prestazioni. Inoltre, di recente sono state introdotte tecnologie che permettono di trasferire dati nella memoria delle GPU con latenze sempre minori.

Perché si possa utilizzare un processore grafico in un trigger di livello 0, è necessario dimostrare che sia possibile trasferire i dati attraverso un collegamento ethernet dal RICH alla memoria della GPU NVIDIA, eseguire i calcoli e comunicare la risposta entro il limite temporale fissato.

Il link ethernet e l'architettura CUDA (*Compute Unified Device Architecture*) delle GPU saranno esaminate nel Capitolo 2, mettendo in evidenza anche quali sono le problematiche specifiche relative al loro impiego in un trigger di basso livello. In tale ambito si offrirà anche una breve panoramica delle caratteristiche dei Sistemi Operativi real-time.

Una serie di test di latenza e banda svolti in configurazione di loopback tra schede ethernet standard, illustrati nel Capitolo 3, hanno permesso di darne una caratterizzazione sia utilizzando un kernel Linux standard che real-time.

Nel Capitolo 4 viene presentata NaNet, scheda di comunicazione a bassa latenza, con la possibilità di accedere direttamente alla memoria GPU attraverso il protocollo GPUDirect peer-to-peer, cioè senza intervento da parte dell'host.

NaNet deriva dal progetto APENet+, una scheda di interconnessione sviluppata dall'“Istituto Nazionale di Fisica Nucleare” (INFN) per un cluster ibrido CPU/GPU a topologia toroidale 3D.

L'utilizzo della scheda NaNet come elemento del trigger di livello 0 è descritto in dettaglio nel Capitolo 5. Qui vengono anche presentati i risultati relativi a banda e latenza ottenuti con GPU NVIDIA di classe Fermi (GF100) e Kepler (GK110).

Tale caratterizzazione ha permesso di verificare l'aderenza del sistema di trigger di livello 0 per il rivelatore RICH di NA62 basato su GPU e NaNet ai requisiti dell'esperimento.

Capitolo 1

L'esperimento NA62

In questo capitolo verrà descritto brevemente l'esperimento NA62, con una particolare attenzione al rivelatore Ring-Imaging Čerenkov (RICH) e al sistema di trigger di livello 0 ad esso associato.

1.1 Una panoramica

L'esperimento NA62 ha come obiettivo la rivelazione di eventi $K^+ \rightarrow \pi^+ \nu \bar{\nu}$ e la misurazione del relativo rapporto di decadimento ("branching ratio", BR), il cui valore previsto dal Modello Standard (SM) è $(8.7 \pm 0.7) \times 10^{-11}$ [1].

Questo raro decadimento (sono attesi circa 100 eventi in 2/3 anni di raccolta dati, a fronte di una frequenza di circa 10^7 decadimenti/sec) è caratterizzato dall'azione del meccanismo Glashow-Iliopoulos-Maiani (GIM) e dal fatto che le incertezze relative a contributi adronici possono essere eliminate sperimentalmente. Di conseguenza, esso risulta un ottimo banco di prova dello stesso SM (vedi fig. 1.2), dando la possibilità, inoltre, di migliorare la nostra conoscenza della matrice Cabibbo-Kobayashi-Maskawa (CKM) [2].

Per raggiungere questo scopo verrà utilizzato il Super Proton Synchrotron (SPS) del CERN che permette di utilizzare un fascio di protoni da 400 GeV/c incidenti su un bersaglio di Berillio e che presenta due notevoli vantaggi:

- poiché la sezione d'urto di produzione dei kaoni aumenta al crescere dell'energia dei protoni, anche con un limitato numero di questi ultimi è possibile ottenere un consistente flusso di kaoni, riducendo così l'eventuale attività non dovuta a tali particelle;
- la produzione di kaoni con un'elevata energia rende più agevole la soppressione del fondo di fotoni prodotti come conseguenza dei decadimenti del tipo $K^+ \rightarrow \pi^+ \pi^0$.

La composizione del fascio risultante è illustrata nella Tabella 1.1.

Al fine di limitare interazioni secondarie dei prodotti di decadimento e del fascio principale, viene utilizzato come spazio di decadimento una camera a vuoto di forma cilindrica, lunga 120 m e dal diametro di 3.7 m.

Per ottenere i dati necessari alla valutazione del fondo e del segnale, si utilizzano i seguenti strumenti (vedi fig. 1.1):

Impulso	$75 \pm 0.9 \text{ GeV}/c$
Rate	750 MHz
Composizione	70% π^+
	23% p^+
	6% K^+
	1% altri

Tabella 1.1. Caratteristiche del fascio dopo la collisione con il bersaglio

- la componente K^+ del fascio è identificata dal contatore differenziale Čerenkov (CEDAR);
- coordinate e impulso delle particelle del fascio sono registrate prima dell'ingresso nella zona di decadimento da 3 detector al silicio, detti Gigatracker (GTK);
- uno spettrometro magnetico nel vuoto (STRAW tracker) rileva e misura coordinate e impulso delle particelle generate nella regione di decadimento;
- un contatore Čerenkov per la separazione di π e μ (RICH);
- una serie di rivelatori dedicati al veto dei fotoni: calorimetro elettromagnetico (LKr) per i fotoni nella direzione del fascio, rivelatore ad anelli di vetro piombato per i fotoni a grandi angoli (LAV), due ulteriori rivelatori a coprire angoli intermedi (IRC) e piccoli (SAC);
- un rivelatore dedicato al veto dei muoni (MUV);
- intorno ai rivelatori GTK sono posizionati dei contatori per identificare particelle che si muovano nella direzione opposta a quella del fascio;
- tra il RICH e il calorimetro LKr è posizionato un rivelatore a scintillazione (CHOD).

Poiché è attesa una frequenza di eventi pari a 10 MHz, si rende necessaria un'attenta strategia di triggering, così da ridurre la quantità di dati da salvare su nastro e far sì che l'acquisizione dati sia praticamente senza perdite ("loss-less") per evitare di introdurre condizioni di veto artificiali che porterebbero ad errori nella stima del fondo.

Il sistema di trigger e acquisizione dati (TDAQ) è basato su una gestione unificata e completamente digitale di questi ultimi per permetterne un totale controllo anche offline.

Si deve considerare inoltre la necessità di sincronizzare tutti i rivelatori e i relativi sotto-sistemi. Questa si ottiene mediante un segnale generato da un oscillatore ad alta stabilità, il cui segnale viene distribuito otticamente a tutti i sistemi attraverso il sistema di Time, Trigger and Control (TTC) usato dal CERN¹.

¹<http://ttc.web.cern.ch/TTC/intro.html>

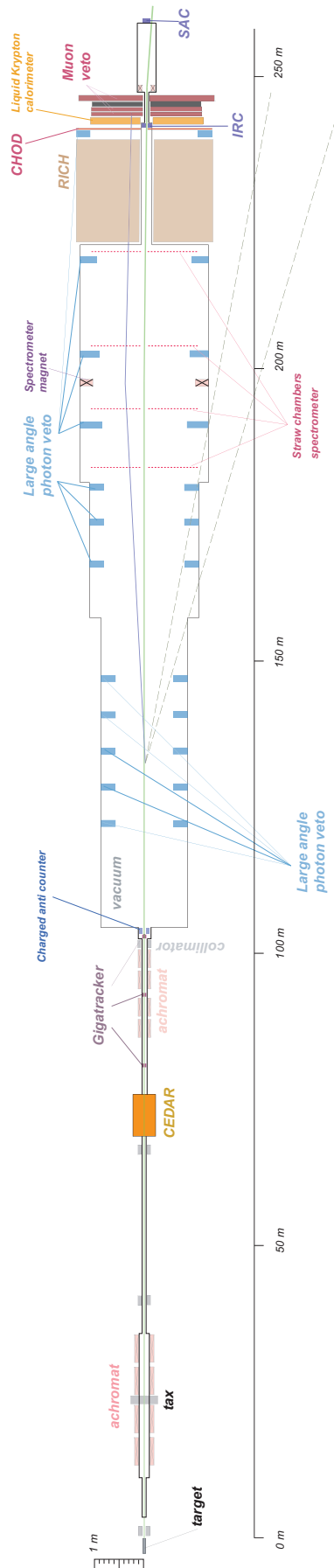


Figura 1.1. Schema dell'esperimento NA62

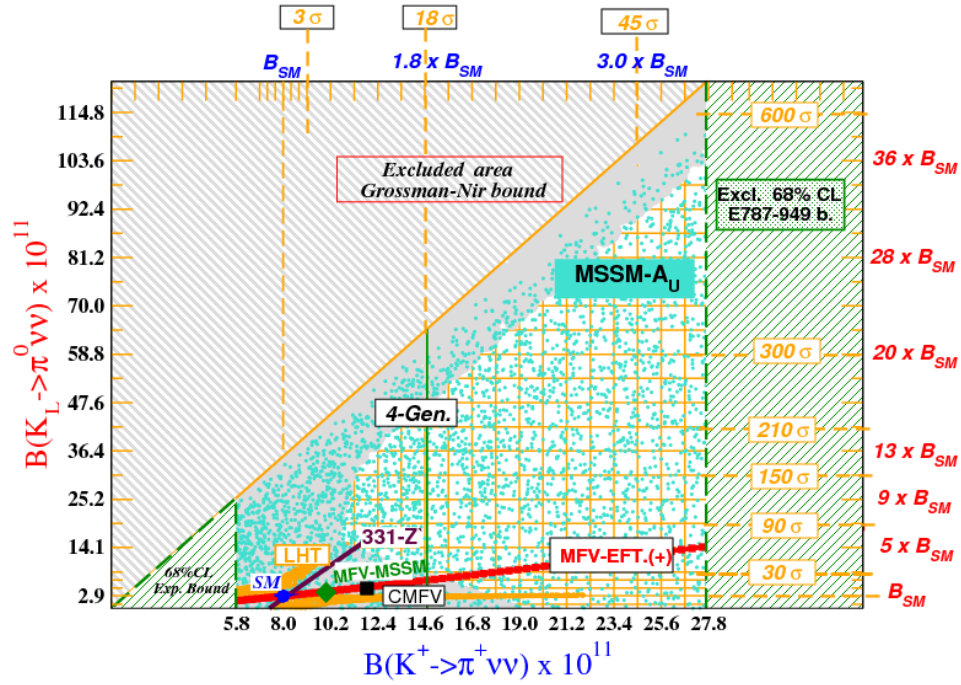


Figura 1.2. Il grafico *Mescia-Smith* [3] illustra i possibili BR, oltre lo SM, per $K^+ \rightarrow \pi^+ \nu \bar{\nu}$ e $K_L^0 \rightarrow \pi^0 \nu \bar{\nu}$

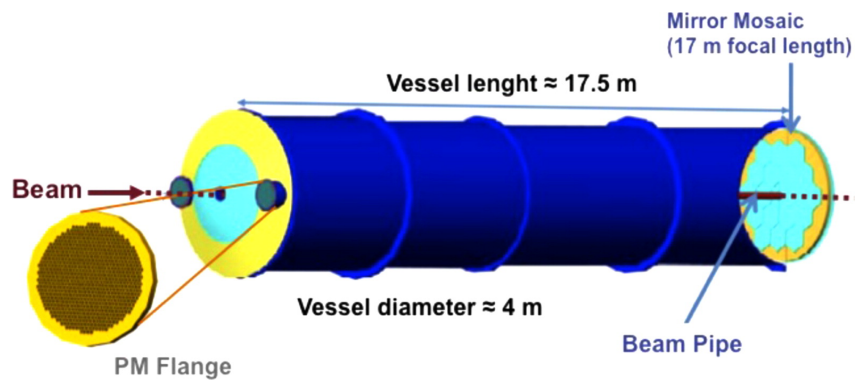


Figura 1.3. Il rivelatore RICH.

1.2 RICH

Il rivelatore RICH ha il compito di ridurre di 10^{-2} il fondo muonico prodotto dai decadimenti $K^+ \rightarrow \mu^+ \pi^+$ nell'intervallo di impulsi $15 \div 35$ GeV/c. Esso garantisce, inoltre, una risoluzione di 100 ps nel tracciamento dei pioni e svolge la funzione di trigger di Livello 0 (L0) per le particelle cariche.

È costituito da un cilindro dal diametro di 2.8 m e lungo 17 m (fig. 1.3), riempito con gas neon a pressione atmosferica. Dal lato di uscita del fascio si trova un mosaico di 18 specchi esagonali e 2 semiesagonali, che formano due superfici sferiche con una lunghezza focale totale di 17 m, mentre 2000 fotomoltiplicatori (ognuno con un diametro di 18 mm), suddivisi su due flange di supporto, sono posizionati dalla parte opposta in corrispondenza dei fuochi, a formare un reticolo esagonale [4].

Si prevede che un tipico anello generato da un π^+ all'interno del range di valori accettabili attivi in media 20 fotomoltiplicatori (fig. 1.4) [5]. Risalendo a centro e raggio, che dipendono da direzione e velocità della particella, è possibile raggiungere la necessaria accuratezza nella rimozione del fondo [6].

Risulta quindi necessario ricostruire una circonferenza, partendo da una serie di punti, cioè i fotomoltiplicatori (PM) attivati, appartenenti a una matrice di 1000.

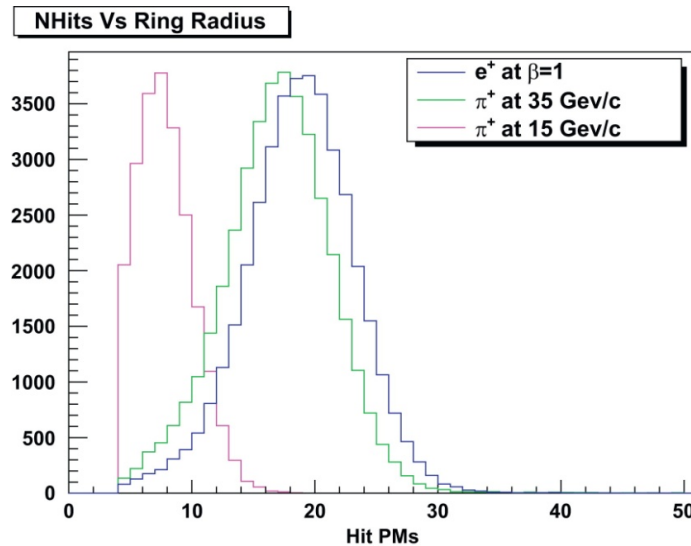


Figura 1.4. Numero di fotomoltiplicatori attivati da diverse particelle cariche a seconda del loro impulso.

1.2.1 Ricostruzione dei cerchi

Nel trigger di Livello 0, al fine di determinare centro e raggio del cerchio rivelatore del passaggio della particella [1], è stato utilizzato il metodo di Crawford [7].

Questo permette di scrivere, per un cerchio di raggio R e centro (x_0, y_0) , le seguenti relazioni:

$$x_o^2 + y_o^2 - R^2 = \frac{1}{N} \{ 2x_0 \sum x_i + 2y_0 \sum y_i - \sum x_i^2 - \sum y_i^2 \} \quad (1.1)$$

$$x_0 \left\{ \sum x_i^2 - \frac{(\sum x_i)^2}{N} \right\} + y_0 \left\{ \sum x_i y_i - \frac{\sum x_i \sum y_i}{N} \right\} = \frac{1}{2} \left\{ \sum x_i^3 + \sum x_i y_i^2 - \sum x_i \frac{\sum x_i^2 + \sum y_i^2}{N} \right\} \quad (1.2)$$

$$x_0 \left\{ \sum x_i y_i^2 - \frac{\sum x_i \sum y_i^2}{N} \right\} + y_0 \left\{ \sum y_i^2 - \frac{\sum y_i^2}{N} \right\} = \frac{1}{2} \left\{ \sum x_i^2 y_i + \sum y_i^3 - \sum y_i \frac{\sum x_i^2 + \sum y_i^2}{N} \right\}. \quad (1.3)$$

dove le sommatorie sono sui punti coinvolti nella ricostruzione.

Le eq. 1.2 e 1.3 possono essere risolte per x_0 e y_0 , mentre R potrà essere determinata dall'eq. 1.1.

L'algoritmo è suddiviso nei seguenti passi, dove si calcolano:

- media delle coordinate:

$$x_m = \frac{\sum x_i}{N} \quad y_m = \frac{\sum y_i}{N} \quad (1.4)$$

- differenze delle coordinate per ogni PM

$$u_i = x_i - x_m \quad v_i = y_i - y_m \quad (1.5)$$

- i valori $u_i^2, v_i^2, u_i v_i, u_i^3, v_i^3, u_i^2 v_i, u_i v_i^2$
- le somme $\sum u_i^2, \sum v_i^2, \sum u_i v_i, \sum u_i^3, \sum v_i^3, \sum u_i^2 v_i, \sum u_i v_i^2$
- x_0, y_0, R , ricavandole dalle equazioni 1.1, 1.2, 1.3.

Poiché più eventi possono essere processati contemporaneamente (vedi par. 2.2.3) e le somme (mediante processi di riduzione) si giovano della parallelizzazione, questa procedura si presta a una implementazione su un'architettura di tipo SIMD (*single-instruction, multiple data*) come quella delle GPU.

Questo algoritmo (definito "MATH") verrà impiegato nei test del Cap. 5.

1.3 TDAQ

Come accennato nel par. 1.1, l'elevato numero di eventi genera un enorme flusso di informazioni che richiede un sistema di triggering e acquisizione dati altamente efficiente.

In linea di principio si potrebbe pensare ad un sistema completamente "triggerless", in cui i dati vengano trasferiti senza restrizioni ai vari computer da dedicare allo scopo, ma considerata la mole delle informazioni, risulterebbe una soluzione eccessivamente costosa.

Si è deciso quindi di utilizzare tre livelli di trigger (si veda fig. 1.5):

- Livello 0 (L0), basato sui dati di RICH e CHOD combinati con i veti di LKr, MUV e LAV, che deve ridurre la frequenza a 1 MHz con una latenza fissata a 1 ms;
- Livello 1 (L1), che riduce la frequenza a circa un centinaio di kHz basandosi su un singolo rivelatore (STRAW o RICH);
- Livello 2 (L2), l'intero evento è ricostruito e l'ulteriore selezione permette di ridurre la frequenza a decine di kHz.

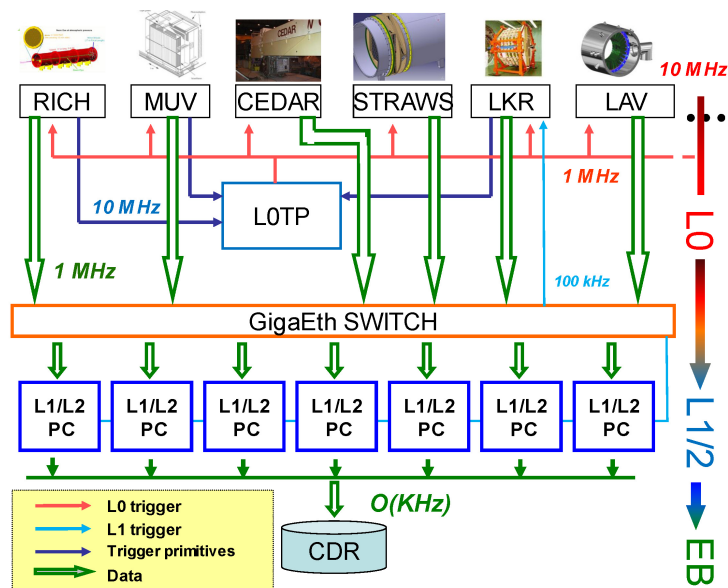


Figura 1.5. Schema sistema triggering

Nel caso del RICH, per soddisfare i requisiti indicati nella sezione 1.2 (risoluzione temporale di 100 ps e impulsi nel range $15 \div 35$ GeV/c), i segnali provenienti dall'elettronica di front-end saranno digitalizzati dalle Time-to-Digital Cards (TDC), montate su un totale di quattro schede TEL62 (fig. 1.6) che ospitano le Pre-Processing FPGA (PP-FPGA). In questo modo possono essere gestiti 2048 canali, sufficienti per il numero di fotomoltiplicatori utilizzati nel rivelatore. Un buffer circolare, di oltre 2GB, permette di memorizzare eventi in attesa della decisione del trigger.

Le FPGA, inoltre, sono in grado di generare primitive di L0 trigger valutando il numero di hits all'interno di una prefissata finestra temporale (Fine Time).

A loro volta le unità di Pre-Processing sono collegate a un'ulteriore FPGA (SyncLink, SL) che ha il compito di raccogliere tutte le primitive di trigger in un unico pacchetto dati (Multi Event Packet, MEP) per massimizzare la velocità di trasferimento (vedi par. 5.2.1).

Tali primitive, vengono inviate dalla TEL62 (che svolge la funzione di scheda di Read-Out, RO) a un L0 Trigger Processor (L0TP) attraverso un collegamento di tipo Gigabit ethernet. Se le condizioni di trigger sono soddisfatte, viene segnalato alle TEL62 che tutti i dati relativi alla finestra temporale dell'evento corrispondente andranno trasmessi ai livelli successivi di trigger.

Trattandosi di comunicazioni punto-punto, non è necessario preventivare perdite di pacchetti o il loro riordino. Quindi i dati possono essere trasmessi utilizzando il protocollo UDP (User Datagram Protocol) che presenta notevoli vantaggi:

- l'header di soli 8 bytes riduce l'overhead nella comunicazione rispetto, ad esempio, al TCP (Transmission Control Protocol);
- è di facile implementazione in macchine a stati finiti su FPGA.

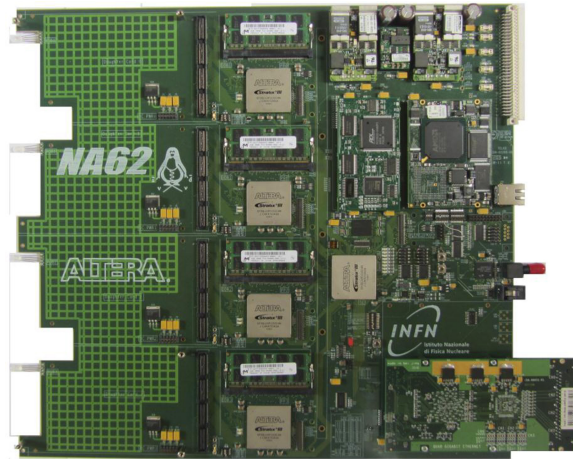


Figura 1.6. La scheda TEL62

1.4 Il Trigger di Livello 0

1.4.1 Problematiche connesse

Come descritto nel paragrafo 1.3 , i dati relativi agli eventi del RICH vengono memorizzati sulle TEL62 in un buffer per forza di cose limitato. Questo ovviamente implica che essi possano essere mantenuti in memoria per non più di un certo tempo prima di dover essere sostituiti con i nuovi. Risulta anche necessario che la latenza totale dei vari step del trigger sia quanto più deterministica possibile, minimizzando le cause di eventuali fluttuazioni.

Da notare che questo vincolo così stringente è comune a tutti gli esperimenti di fisica delle alte energie e ha sempre limitato gli strumenti utilizzabili per il trigger di livello 0 a dispositivi hardware dedicati, con conseguente aumento di costo, tempo per lo sviluppo e la manutenzione rispetto a soluzioni commerciali.

Sarebbe possibile soddisfare i requisiti di latenza massima con un sistema (Personal Computer) standard, quindi facilmente reperibile e dai costi più contenuti?

Da non trascurare, inoltre, che così non ci sarebbero ulteriori tempi “morti” di trasferimento, in quanto tutti i calcoli verrebbero effettuati in un sistema tutto sommato compatto.

Le caratteristiche del trigger di livello 0 associato al RICH dell'esperimento NA62, in particolare la richiesta di una latenza $< 1\text{ms}$, lo rendono un ottimo banco di prova per esplorare questa possibilità.

Osserviamo che una volta disponibili le primitive di trigger, i passaggi che portano alla decisione sono:

- invio dati dalla TEL62 al L0TP;
- ricezione ed elaborazione dati nel L0TP;
- invio messaggio da L0TP a TEL62 con istruzioni per salvare i dati relativi all'evento.

La finestra temporale di queste fasi dovrà essere compatibile con quella richiesta dall'hardware dell'esperimento.

I sottosistemi di un PC standard coinvolti sono ovviamente:

- l'interfaccia con la rete (Network Interface Card, NIC), nel caso più semplice la scheda ethernet;
- il processore (Central Processing Unit, CPU) e la memoria di sistema (RAM).

Il sistema di gestione della memoria di massa (Hard Disk) non interviene in maniera rilevante durante lo svolgimento di queste operazioni.

1.4.2 Un possibile approccio risolutivo: NaNet + GPU

Negli ultimi anni, molte attenzioni si sono concentrate intorno all'evoluzione di un particolare dispositivo: l'unità di elaborazione grafica (Graphics Processing Unit, GPU).

Verranno esaminate in maniera più approfondita le sue caratteristiche nel par. 2.2.1. Per ora è sufficiente sottolineare che le GPU offrono:

- un'elevata potenza di calcolo (possono essere utilizzati algoritmi più selettivi anche se particolarmente esosi in termini di risorse);
- un'ampia banda verso il sistema che le ospita (si possono trasferire bidirezionalmente grandi quantità di dati nell'unità di tempo);
- versatilità (può essere programmata per svolgere compiti diversi);
- scalabilità (sono possibili sistemi multi-GPU);
- costi contenuti dovuti all'economia di scala;
- tecnologia sempre all'avanguardia poiché trainata dal mercato videoludico.

Analizziamo un po' più in dettaglio ciò che accade utilizzando come L0TP un sistema "off-the-shelf" con GPU (vedi fig. 1.7).

I dati inviati dalla scheda di RO vengono ricevuti dalla NIC ethernet, copiati dal driver nella memoria riservata al dispositivo in RAM (kernel space), dopo di che viene effettuata una nuova copia, sempre in RAM, nella regione accessibile alle applicazioni (user space), poi i dati vengono copiati in memoria GPU, i calcoli

eseguiti e i risultati vengono copiati di nuovo in RAM, per intraprendere poi il percorso inverso verso la scheda di RO.

Questi passaggi, inoltre, dipendono fortemente dal comportamento del sistema operativo (SO) installato sulla macchina, il quale si trova a gestire inevitabilmente anche tutta una serie di altri processi concorrenti, introducendo variazioni non prevedibili della latenza.

È evidente, quindi, che se si potessero evitare le multiple copie in memoria di sistema e si riducesse al minimo l'intervento del SO, si abbasserebbe di molto la latenza totale.

In particolare, la situazione più favorevole è quella in cui sia possibile trasferire i dati direttamente dalla scheda di rete nella memoria della GPU.

Nel caso di GPU Nvidia, un meccanismo di copia diretta pensato esclusivamente tra schede dello stesso produttore, montate sulla medesima macchina (GPUDirect v.2 con peer-to-peer) è stato rilasciato nel 2011².

L'anno successivo, la scheda APENet+, una NIC basata su FPGA per cluster con topologia toroidale 3D dedicati al calcolo ad alte prestazioni, è stata il primo dispositivo non NVIDIA a implementare e sfruttare tale possibilità per effettuare trasferimenti dati su una rete (vedi Cap. 4) [8].

Il progetto della scheda NaNet, derivata da APENet+, si pone l'obiettivo di realizzare un sistema di trigger a bassa latenza, real-time, basato su GPU.

La caratterizzazione di tale sistema è argomento di questa Tesi.

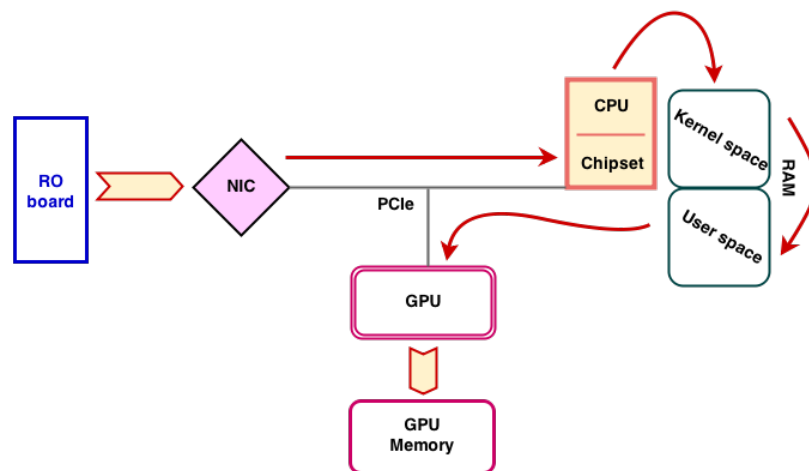


Figura 1.7. Diagramma sinttico degli step necessari a trasferire i dati dalla scheda di RO alla memoria GPU in un sistema standard

²<https://developer.nvidia.com/gpudirect>

Capitolo 2

Considerazioni sulle tecnologie di interesse

In questo capitolo verranno descritti alcuni degli strumenti hardware e software che, nelle loro più recenti evoluzioni, hanno motivato l'esplorazione della fattibilità di un possibile trigger di livello 0 di nuova concezione: il link attraverso il quale avviene il trasferimento delle primitive dalle schede di RO all'LOTP e la sua gestione da parte del sistema operativo; i processori grafici, a cui è demandata la parte di calcolo del trigger; l'interfaccia di comunicazione PCI express tra il computer ospite e la stessa GPU.

Si presenteranno anche brevemente le caratteristiche di un sistema operativo real-time, in previsione della descrizione dei test svolti nel capitolo successivo.

2.1 Il link dati

Come detto nel par. 1.3, la comunicazione tra le TEL62 e il LOTP avviene tramite protocollo ethernet, utilizzando pacchetti UDP.

Nella configurazione finale verranno utilizzati 4 distinti canali GbE, uno da ogni TEL62, che poi verranno aggregati prima di raggiungere la NIC.

Per tutti i test di fattibilità realizzati ai fini di questa Tesi si è utilizzato un singolo canale da 1 Gb. Ciò ha permesso di ottenere risultati significativi e utili al progetto del sistema completo.

Esaminiamo ora come viene gestita la ricezione di un pacchetto UDP da un sistema Linux.

2.1.1 Socket

Il canale di comunicazione tra processi diversi, in Linux, viene stabilito attraverso un descrittore di file (File Descriptor, FD) detto socket. Nel caso di interfaccia tra processo utente e lo stack di rete del kernel (vedi par. 2.1.2), questo metodo è caratterizzato da *dominio*, *tipo* e *protocollo*. Ad esempio:

```
socketId = socket(AF_INET, SOCK_DGRAM, 0);
```

crea un socket, il cui dominio è IPv4 (AF_INET), di tipo datagramma (per invio "connectionless" di pacchetti). Il campo del protocollo è 0 poiché in questo caso

esiste solo un protocollo (UDP) supportato da quel tipo di socket, all'interno del dominio specificato.

Da sottolineare che attraverso *setsockopt()* è possibile modificare importanti proprietà di un socket, quali la possibilità di riutilizzare un indirizzo a cui si è già connessi oppure collegare il socket a un'interfaccia di rete specifica. Questo tornerà utile quando verranno illustrati i risultati ottenuti in *loopback* con due schede ethernet nel par. 3.1.

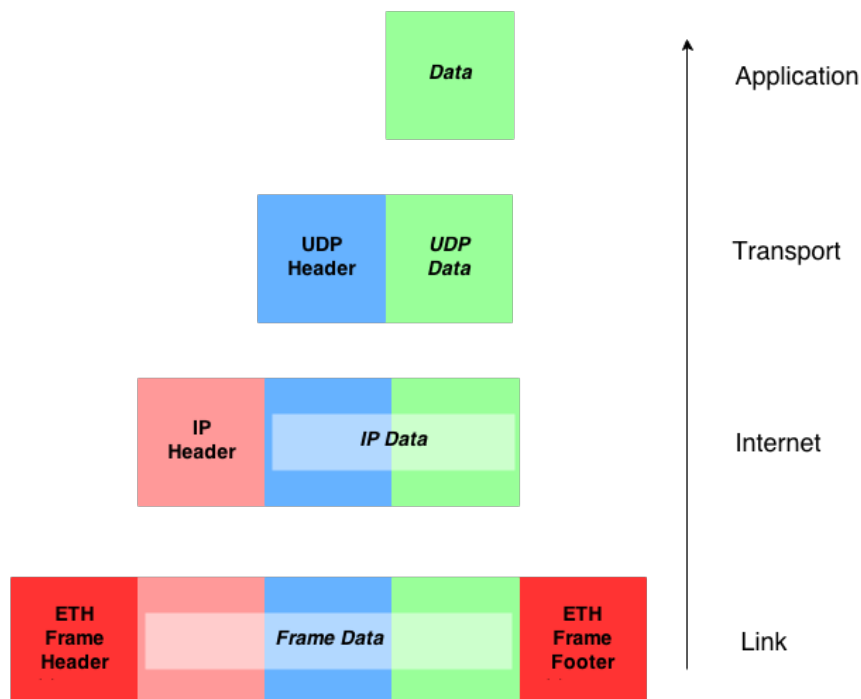


Figura 2.1. Dettaglio dei livelli dello stack di rete in Linux.

2.1.2 Lo stack di rete in Linux

Dall'interfaccia di rete i dati sono copiati, tramite accesso diretto alla memoria (*Direct Memory Access*, DMA), in un socket buffer.

Il driver di rete viene notificato della presenza del pacchetto tramite interrupt o polling (dipende dall'implementazione del driver) e prepara la struttura *sk_buff* (fig. 2.2), nel kernel space, che contiene il pacchetto dati e campi che puntano a specifici livelli dello stack di rete: link, internet, trasporto [9].

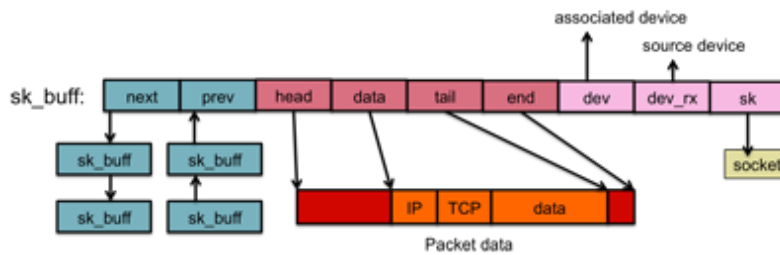


Figura 2.2. Il socket buffer.

L'elaborazione dei vari livelli avviene secondo lo stack di rete del kernel (vedi fig. 2.1):

- Link: ogni pacchetto ethernet ha codificato nell'header gli indirizzi MAC di origine e destinazione, lunghezza totale in bytes mentre il footer è costituito da un Controllo a Ridondanza Ciclica (CRC) di 32 bit;
- Network: secondo quanto codificato nell'header IP, il pacchetto verrà instradato verso un'altra destinazione, utilizzato localmente, oppure scartato.
- Transport: viene indicato il protocollo utilizzato (TCP/UDP o altro);
- Application: Il pacchetto viene reso disponibile dal socket buffer di ricezione allo spazio dell'applicazione.

Da notare che i dati contenuti nel pacchetto non vengono mai spostati durante i vari passaggi dello stack. Questo per ridurre la latenza. Alla fine risultano solo due operazioni di copia: dal dispositivo di rete nel socket buffer e da questo all'applicazione dell'utente.

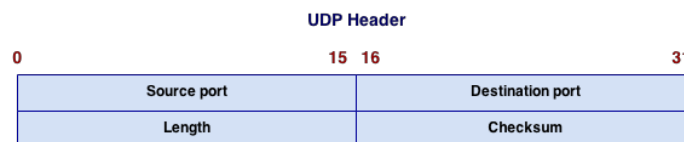


Figura 2.3. Header del pacchetto UDP.

2.1.3 Protocollo UDP

Questo protocollo di trasporto offre una serie di vantaggi: ridotto overhead, semplicità ed efficienza (vedi anche par. 1.3).

L'header è costituito da quattro campi di 2 bytes (fig. 2.3):

- Porta di destinazione (Destination Port): identifica univocamente l'applicazione client;
- Porta sorgente (Source Port): è la porta utilizzata dal processo server;
- Lunghezza totale in bytes del pacchetto UDP (UDP length);

- da notare che, nel caso di protocollo IPv4, la lunghezza massima dei dati contenuti nel datagramma UDP non può essere superiore a 65,507 bytes (65,535 – 8 byte UDP header – 20 byte IP header);
- Checksum: se il campo dati è abilitato, permette di verificare se ci sono stati errori nella trasmissione del pacchetto, altrimenti il campo è riempito da zeri.

2.2 GPU

Negli ultimi anni, i processori grafici stanno godendo di una crescente popolarità nell'ambito del calcolo ad alte prestazioni (High-Performance Computing, HPC) perché permettono di superare alcune limitazioni intrinseche all'architettura dei processori standard [10].

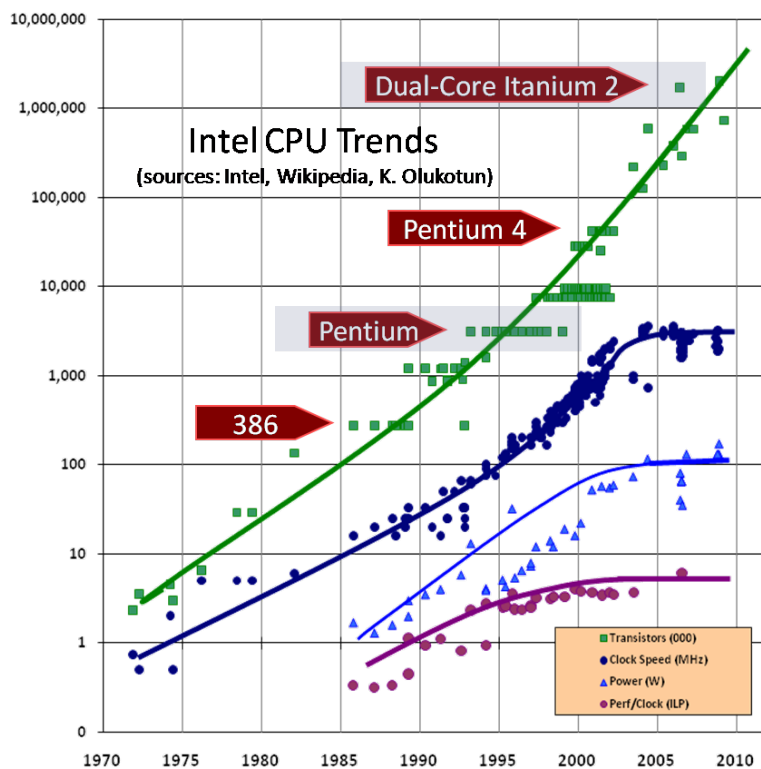


Figura 2.4. Andamenti delle caratteristiche salienti delle CPU nel corso degli ultimi anni.

2.2.1 Limiti delle CPU

Per oltre trent'anni, a partire dalla fine degli anni Settanta, l'aumento di prestazioni di una CPU si è ottenuta principalmente attraverso l'incremento della frequenza di funzionamento. Questo perché l'esecuzione dei programmi era pensata come strettamente sequenziale.

Come riferimento: il processore dell'Apple II (1977) lavorava a 1 MHz, nel 2001 i processori Intel hanno raggiunto i 2 GHz ed ora (2014) sfiorano i 4 GHz. C'è stato un evidente rallentamento nella corsa al clock, come si può vedere nella fig. 2.4.

I produttori hanno dovuto rivedere questa strategia trovandosi di fronte a problemi tecnici nella costruzione dei circuiti integrati: raggiungimento di limiti fisici nella realizzazione dei transistor (in particolare, non è possibile scendere al di sotto di una certa dimensione del *gate* poiché l'effetto tunnel vanificherebbe la sua funzione di controllo del flusso di elettroni tra *source* e *drain*), consumo di energia e dissipazione del calore.

Si è venuta a creare una situazione in cui la *Legge di Moore* [11], nella versione: “le prestazioni dei processori aumentano ogni 18 mesi”, non sarebbe più stata valida.

L'approccio dei principali produttori è stato allora quello di realizzare CPU con più core (da 2 a 12 al momento). Si sono aperte, in questa maniera, le porte alla diffusione del calcolo parallelo, prima di allora relegato alla nicchia dei supercomputer.

Ciononostante, questi processori multi-core mantengono la stessa impostazione di un single-core, cioè sono ottimizzati per l'esecuzione di compiti sequenziali e per minimizzare la latenza di esecuzione di un singolo thread.

Fondamentalmente le CPU sono ottimizzate per applicazioni dove la maggior parte del lavoro è a carico di un numero limitato di thread, sono presenti differenti tipologie di calcolo e un numero elevato di diramazioni condizionali.

Gli elementi essenziali del design di una CPU sono:

- **una logica di controllo sofisticata:** per gestire operazioni dello stesso thread effettuate in parallelo o al di fuori dell'ordine sequenziale;
- **ampia memoria cache:** per ridurre la latenza di accesso ai dati, evitando continui riferimenti alla memoria di sistema;
- **unità aritmetico-logiche** (arithmetic and logic unit, ALU): componenti dedicati all'esecuzione delle operazioni.

La porzione di chip dedicata al puro calcolo è quindi, per forza di cose limitata. Logica di controllo e cache occupano infatti spazio e consumano energia.

In fig. 2.5 il confronto con l'architettura di una GPU che verrà descritta nella prossima sezione.

2.2.2 Una diversa filosofia

Il design delle GPU deve soddisfare ovviamente le richieste del mercato per cui sono state sviluppate, cioè gestire l'enorme quantità di calcoli floating-point richiesti dai videogiochi o dalle applicazioni video in generale, il tutto a costi competitivi, e con un consumo energetico limitato.

Per questo motivo si è cercato di massimizzare (fig. 2.5):

- l'area del chip dedicata alle unità di calcolo;
- la quantità di istruzioni eseguite, attraverso l'utilizzo di un grande numero di thread (questo approccio è solitamente definito come “throughput-oriented”).

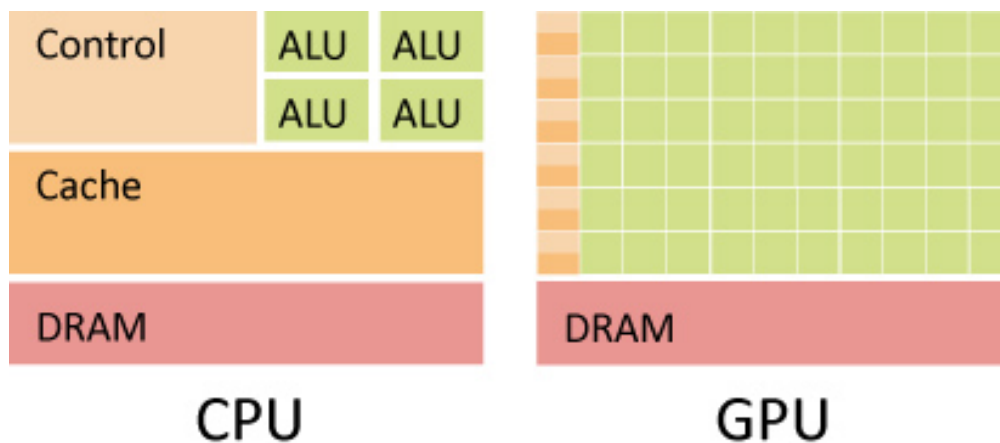


Figura 2.5. Confronto tra le due architetture.

Da ciò ne è derivata un'architettura generale così strutturata:

- unità di controllo molto semplificata¹;
- quantità limitate di cache, articolata su più livelli;
- un gran numero di ALU (rispetto a una CPU sono più semplici, con una maggiore latenza ma più efficienti dal punto di vista del consumo energetico).

Attualmente i principali produttori di GPU orientate al calcolo ad alte prestazioni sono AMD e NVIDIA.

Nel seguito ci riferiremo, tuttavia, solo a soluzioni di quest'ultimo, poichè rendono disponibili strumenti tali da consentire un migliore controllo dell'hardware. Inoltre, come già accennato nel par. 1.4.2, la scheda NaNet sfrutta la modalità di accesso diretto alla memoria del processore grafico attraverso (GPUdirect), disponibile per le schede NVIDIA.

2.2.3 L'architettura CUDA

Per venire incontro sia ai tradizionali compiti di un processore grafico che a quello che viene definito come GPGPU, cioè "general-purpose computing on graphics processing units", NVIDIA ha adottato un'architettura denominata CUDA (*Compute Unified Device Architecture*).

Essa espone il parallelismo attraverso una serie di "Streaming Multiprocessors" (SM), ciascuno dei quali può eseguire un migliaio di thread contemporaneamente.

Ogni SM è costituito da un elevato numero di core (detti CUDA core), variabile a seconda della generazione della GPU.

Ad esempio, per la classe FERMI sono 32/SM, per la KEPLER il numero sale a 192 (vedi par. 2.2.4).

¹Non sono disponibili, ad esempio, la predizione delle diramazioni (*branching prediction*) o l'esecuzione "out-of-order" delle istruzioni.

Single instruction multiple threads

Per gestire l'elevato numero di thread l'SM impiega un approccio detto SIMT (*single-instruction, multiple-thread*). Ogni thread viene mappato su un core e il multiprocessore li suddivide e gestisce in gruppi chiamati *warp*. Questa è una variante dell'architettura *single-instruction, multiple data* (SIMD).

Da notare che ogni warp è indipendente dagli altri e ciascuno di essi esegue un'istruzione alla volta. Quindi la massima efficienza si ottiene quando tutti i thread del warp seguono lo stesso schema di calcolo. Nel caso di condizioni che inducano una diramazione, esse verranno eseguite in maniera seriale, con grande degrado delle prestazioni.

Ogni thread fa parte di un *warp* (gruppo di 32 thread), appartenente a un *blocco*. I blocchi a loro volta sono organizzati in una griglia (*grid*). I thread in un blocco, i blocchi nella griglia e le griglie stesse sono elementi indicizzati come vettori tridimensionali (fig. 2.6).

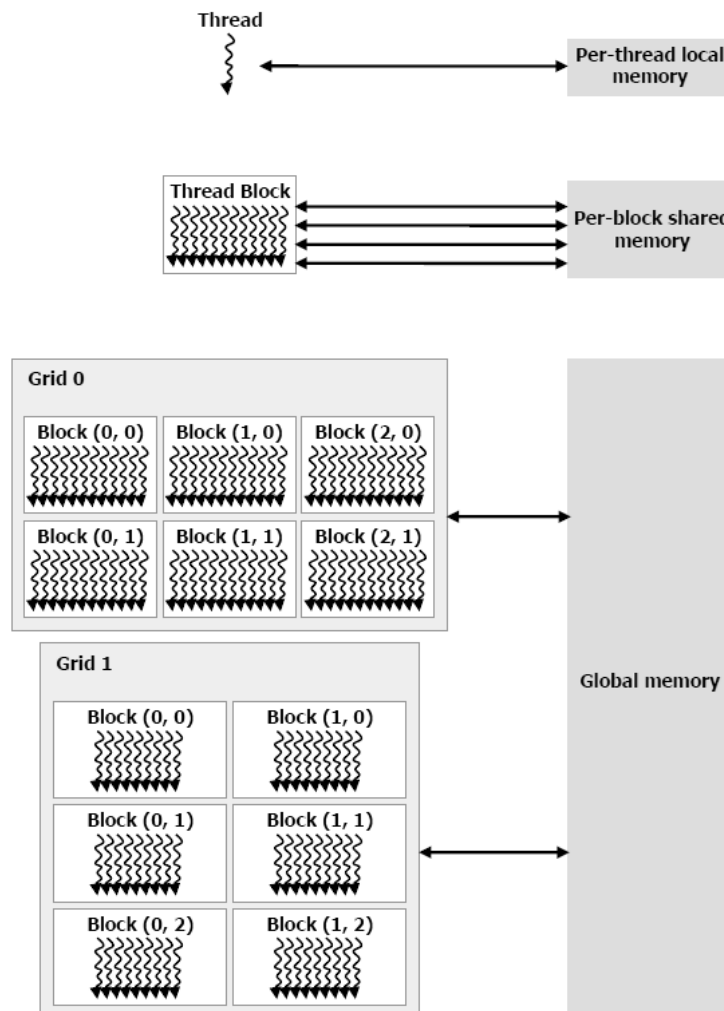


Figura 2.6. Organizzazione dei thread in blocchi e griglie. Sono indicati anche i tipi di memoria a cui possono accedere.

Organizzazione della memoria

Un multiprocessore ha a disposizione vari tipi di memorie (fig. 2.7), di seguito ordinate secondo la latenza di accesso da parte dei core di calcolo:

- un set di registri per core;
- una “shared memory” condivisa tra tutti i core di un SM;
- una “constant cache”, di sola lettura, condivisa da tutti i core per velocizzare le letture dalla “constant memory”, un’area di sola lettura presente nella memoria della GPU;
- una “texture cache”, di sola lettura, condivisa da tutti i core per velocizzare le letture dalla “texture memory”, un’area di sola lettura presente nella memoria della GPU;
- vi sono poi la memoria globale del dispositivo, condivisa da tutti gli SM e quella cosiddetta “locale” (parte di quella globale che può essere riservata dai singoli thread).

Da notare che il numero di blocchi che possono essere gestiti contemporaneamente da un SM dipende dal numero di registri per thread e da quanta shared memory per blocco sono richiesti dall’applicazione, poiché entrambi sono condivisi tra tutti i core del multiprocessore.

La memoria globale, essendo “off-chip”, è quella che presenta la latenza maggiore, da 200 a 800 cicli di clock a seconda dell’architettura del dispositivo (Fermi o Kepler) [12].

È utile sottolineare i confronti tra larghezza di banda relativa alla memoria e prestazioni di calcolo (in doppia precisione) nel caso di un sistema processore/RAM e in quello di un processore grafico:

- banda memoria globale GPU (K20x): ~250 GB/s
- banda RAM (DDR3@2133): ~17 GB/s;

e

- prestazioni GPU (K20x): ~1.22 TFlops;
- prestazioni Intel Xeon E5-2609@2.4GHz: ~77GFlops;

i due rapporti hanno un valore pari a ~15, evidenziando come l’aumentata disponibilità di risorse di calcolo abbia bisogno di una banda di accesso ai dati maggiore per mantenere un efficiente bilanciamento dell’architettura.

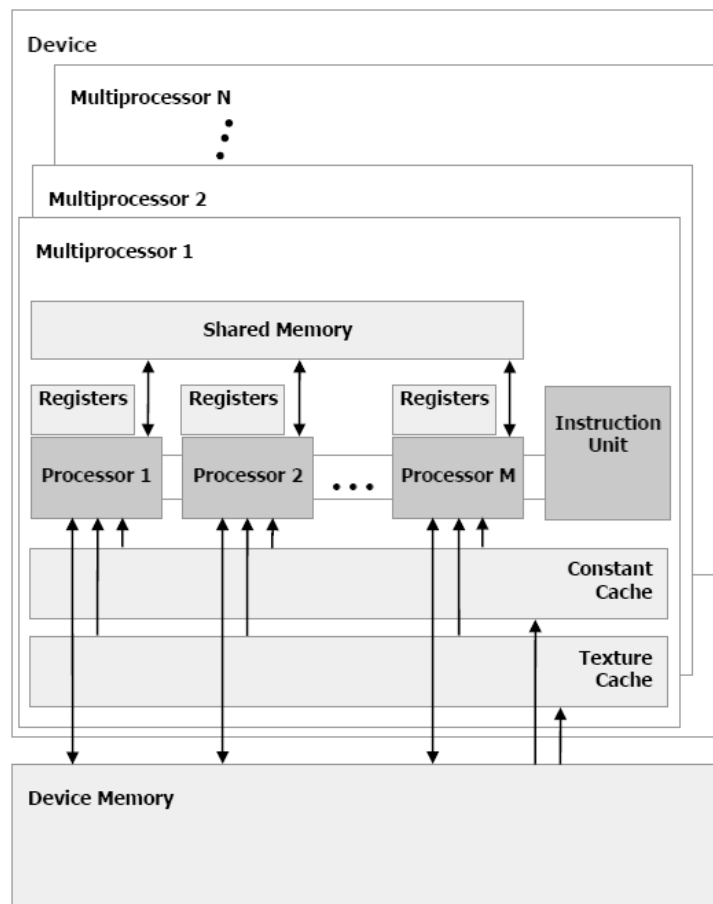


Figura 2.7. Schema generale della gerarchia della memoria in una GPU.

PCI express: collegamento tra host e device

Si definiscono, in un sistema eterogeneo GPGPU (*General-purpose computing on graphics processing units*):

- “Host” il sistema costituito da CPU e suo spazio di memoria;
- “Device” il o i processori grafici ad esso collegati e relative memorie.

Questi due elementi sono solitamente collegati attraverso un canale di comunicazione (“bus”) ad alta velocità detto PCI Express (PCIe).

Fisicamente la connessione è costituita da più piste (“lane”) indipendenti connesse serialmente. Ognuna di esse è formata da due coppie di fili (utilizzando una tecnologia differenziale), una per la trasmissione e l’altra per la ricezione.

Ci possono essere da 1 a 16 piste in un singolo slot (indicate come x1, x4, x8, x16).

Per lo standard PCIe gen. 2.0, ciascuna di esse permette di effettuare 5 GT/s (“Gigatrasfers” per secondo), con una codifica $8b/10b^2$ la banda reale è di 500 MB/s.

²Per trasferire 8 bit di dati, ne sono necessari 10. In questa maniera la banda teorica è ridotta del 20%.

Nel caso di PCIe gen. 3.0 ogni pista raggiunge gli 8 GT/s e, con una codifica $128b/130b^3$, si raggiunge una banda di 985 MB/s. Quindi nel caso di PCIe 3.0 x16, la banda totale è di circa ~16GB/s.

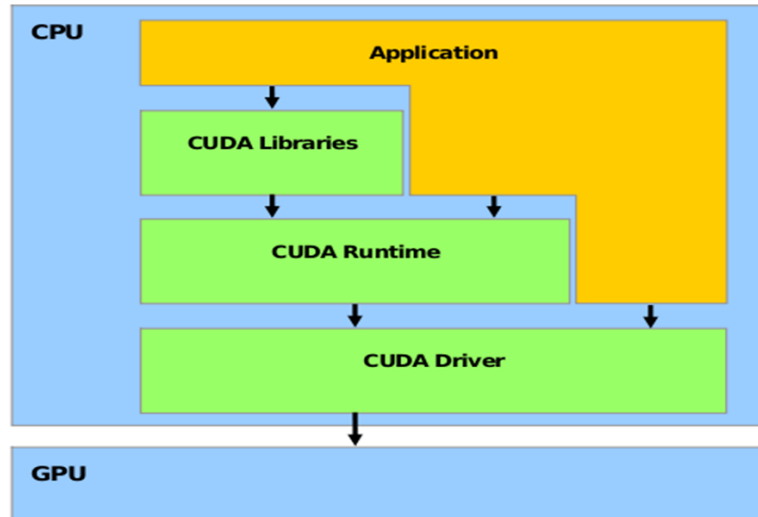


Figura 2.8. Schema implementativo del CUDA C.

Programmazione in un sistema eterogeneo: il CUDA C

Per facilitare la programmazione in un sistema eterogeneo, NVIDIA ha messo a disposizione un'estensione del linguaggio C/C++ detta CUDA C e relativo compilatore (*nvcc*).

Esso fornisce delle API (“Application programming interface”) driver e runtime, oltre a una serie di librerie (fig. 2.8).

L'API runtime è intermedia tra l'applicazione e il driver e facilita la programmazione mascherando alcuni dettagli. Mentre l'API driver, che si pone tra codice compilato e GPU, permette una maggiore ottimizzazione del codice tramite interventi diretti dell'utente.

Un programma CUDA avrà parte del codice che sarà eseguito sull'host e parte sul device.

Il compilatore *nvcc* le separerà per poi:

- compilare il codice in una forma “assembly” (PTX code) o binaria (“cubin object”);
- modificare il codice host in modo da implementare le chiamate a funzioni specifiche del CUDA runtime.

Alla fine, il codice C risultante verrà passato al compilatore standard dell'host che provvederà anche alla fase di link.

³Riduzione della banda dell'1,54%.

Kernel In CUDA C, è definito un nuovo tipo di funzione `__global__` detta “kernel” che viene eseguita N volte in parallelo da N differenti thread del device e ha le seguenti caratteristiche:

- è invocata nel codice host usando una sintassi “ad hoc”: `<<< ... >>>` (detta *chevron notation*);
- deve ritornare sempre un valore *void*;
- è di default asincrona, ovvero dopo il lancio del kernel il controllo è restituito all’host che può quindi proseguire nell’esecuzione dell’applicazione.

Nel listato 2.1, troviamo un esempio:

Listato 2.1. Codice CUDA con definizione e invocazione di un kernel

```
// Definizione del kernel
__global__ void test(...)
{
    ...
}

int main()
{
    ...
    // Chiamata della funzione kernel con N thread e 1 blocco
    test<<<1, N>>>(...);
    ...
}
```

Funzioni di base riguardanti la memoria In CUDA C, la memoria sul device è solitamente allocata tramite `cudaMalloc()` e liberata con `cudaFree()`. Per copiare i dati host ↔ device e device ↔ device viene utilizzato `cudaMemcpy()`.

Una particolare importanza ha la funzione `cudaMallocHost()` che permette di allocare sull’host una porzione di memoria “page-locked”, ovvero non può essere oggetto di swap da parte del Sistema Operativo, comunicandolo al driver del device e permettendogli un accesso tramite `cudaMemcpy` più veloce. Ciò può avvenire poiché ogni trasferimento di dati su PCIe avviene mediante DMA, a partire da memoria riservata (“pinned”). Quindi partendo da memoria allocata normalmente, dovrebbe essere copiata dalla CPU in un buffer apposito, per poi procedere alla transazione con il device. `CudaMallocHost` evita questo ulteriore passaggio, riducendo ovviamente la latenza.

2.2.4 Fermi - Kepler: differenze fra le architetture

I test di NaNet come elemento del LOTP (vedi Cap. 5) sono stati effettuati utilizzando due schede NVIDIA appartenenti alle classi Fermi e Kepler (la più recente).

È opportuno quindi evidenziare caratteristiche e differenze fra le due architetture.

Fermi (GF100)

Nel par. 2.2.3 abbiamo visto che gli elementi principali di una GPU sono i CUDA core, raggruppati in SM.

Una GPU Fermi ha 512 core, suddivisi tra 16 SM [13].

Ognuno di essi è composto da (fig. 2.9):

- 32 core, ognuno dei quali può effettuare calcoli interi e in virgola mobile;
- 16 unità “load-store” (LD/ST) per operazioni della memoria;
- 4 unità per il calcolo di funzioni speciali (Special Function Unit, SFU);
- 32K word da 32 bit dedicate ai registri;
- 64K di RAM suddivisibili tra cache e memoria locale.
- 2 “warp scheduler” e altrettante “dispatch unit” che gestiscono l’esecuzione delle istruzioni.

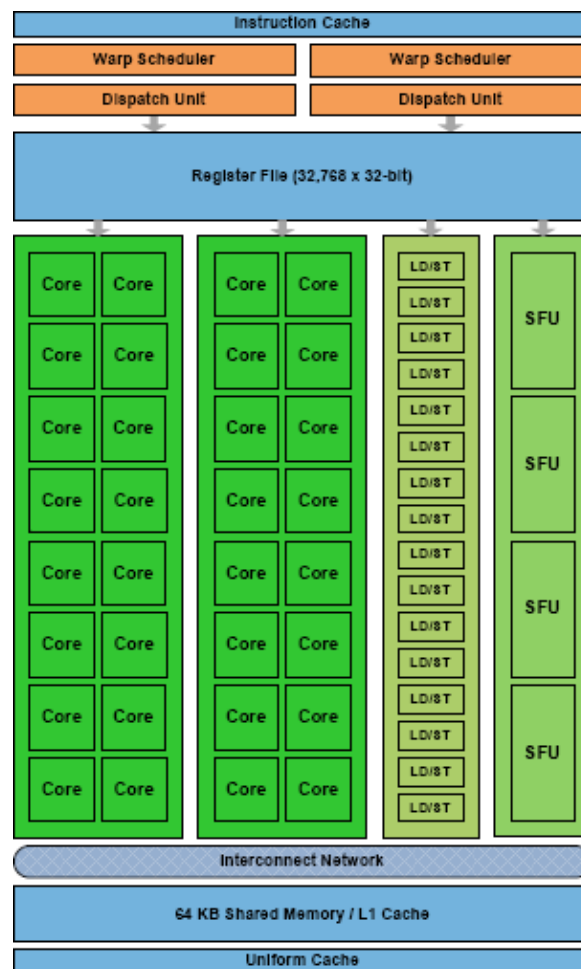


Figura 2.9. SM di una GPU classe Fermi.

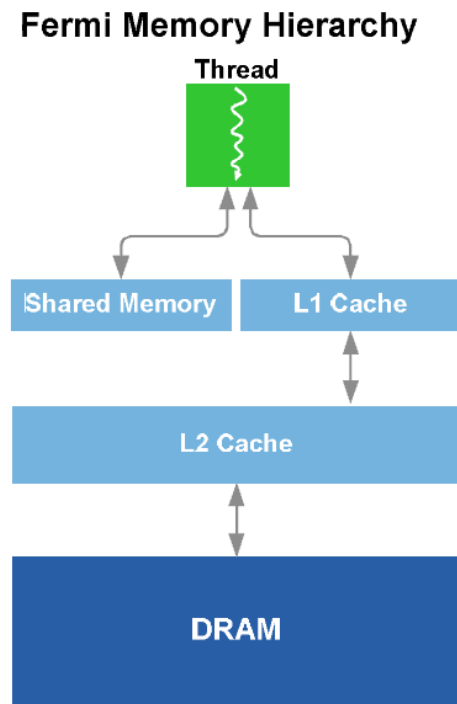


Figura 2.10. Nelle GPU Fermi la cache L1 è configurabile mentre quella L2 unificata.

Viene introdotta con questa architettura la possibilità di utilizzare parte della memoria locale, presente in ogni SM, come cache L1. Oltre che per le transazioni da e per la RAM del dispositivo può essere utilizzata per lo “spilling” dei registri, nel caso non fossero sufficienti per le operazioni svolte dai thread nel SM.

In particolare i 64K sono suddivisi tra la L1 cache appunto e la memoria shared. Quest’ultima svolge un ruolo molto importante, permettendo ai thread di uno stesso blocco di cooperare accedendo a un’area di memoria condivisa.

È presente anche una cache di secondo livello (L2), di 758KB comune questa volta a tutti i core del device, utile per condividere velocemente dati fra diversi SM.

Kepler (GK110)

Con questa nuova architettura, ottimizzata per il risparmio energetico, è stata rinnovata la struttura stessa del Multiprocessore, che ora viene chiamato SMX [14].

Ognuno di essi è costituito da (fig. 2.11):

- 192 CUDA core, ognuno dei quali può eseguire calcoli interi e in virgola mobile;
- 8 volte il numero di SFU presenti nelle Fermi;
- 4 “warp scheduler” e 8 “dispatch unit”.

La memoria è organizzata in maniera simile all’architettura Fermi (fig. 2.12).

La banda verso la memoria shared è però aumentata da 64b a 256b per ciclo di clock del core.

È stata aumentata a 1536KB la cache di secondo livello, con una larghezza di banda raddoppiata.

Vi è ora un'ulteriore livello di cache utilizzabile dai SMX (48KB di sola lettura) pensata per agevolare tutte le operazioni di load.

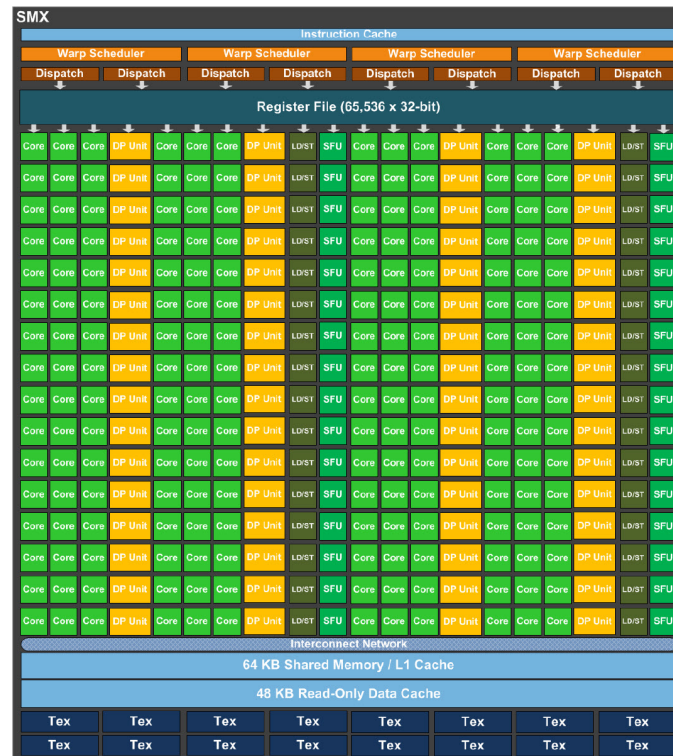


Figura 2.11. SM di una GPU classe Kepler.

2.3 Alcune considerazioni su SO real-time

Come accennato nel par. 1.4.2, è importante che il Sistema Operativo dell'host non introduca variazioni imprevedibili della latenza durante l'esecuzione delle applicazioni. Ovvero si richiede un comportamento il più possibile "deterministico".

Un sistema operativo in grado di svolgere le sue funzioni senza violare specifici vincoli di tempo e in modo che la durata di tali operazioni possa essere prevista a priori viene definito real-time.

Linux storicamente non soddisfa tali condizioni. Basti pensare, ad esempio, che i task del kernel non possono essere interrotti (modello "non-preemptive") e il passaggio di contesto potrà avvenire solo al termine del processo. Quindi il tempo di attesa per l'assegnazione della CPU ad un processo che si desidera abbia una priorità maggiore di quello in esecuzione è indeterminabile.

Negli ultimi anni è stato fatto un grande sforzo da parte degli sviluppatori di SO nel migliorare quelle che sono definite caratteristiche real-time⁴:

⁴<http://www.ibm.com/developerworks/library/l-real-time-linux/>

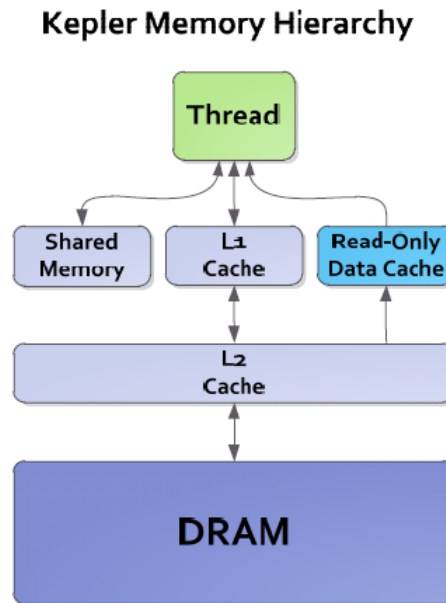


Figura 2.12. Kepler: schema gerarchico della memoria.

- predicibilità dei tempi di risposta;
- riduzione del jitter;
- accuratezza al μs ;
- granularità temporale.

Nella versione 2.6 del kernel sono stati apportati miglioramenti per implementare alcune caratteristiche real-time. I cambiamenti più importanti vengono, tuttavia, introdotti attraverso una specifica “RT patch”, il che comporta la necessità di utilizzare un kernel diverso da quello standard.

C’è da tenere comunque presente che l’utilizzo di un kernel real-time non è una panacea, dipendendo molto dalle applicazioni che si intendono utilizzare.

L’obiettivo di un determinismo, consistente e a bassa latenza, si ottiene, ad esempio, con dell’overhead addizionale dovuto principalmente alla gestione di interrupt hardware in thread separati. Questo aumento dell’overhead, in certe condizioni di lavoro, può risultare in una degradazione delle prestazioni.

Nel Cap. 3 verranno illustrati ad alcuni test di fattibilità nelle condizioni di lavoro del Trigger, confrontando un kernel “vanilla” e uno RT.

Capitolo 3

Configurazione in *loopback*

Prima di affrontare nei successivi capitoli il dettaglio dell'implementazione del trigger di livello 0 con la scheda NaNet, illustriamo alcuni test di latenza della comunicazione effettuati con schede ethernet standard e kernel Linux “vanilla” (cioè standard) e real-time.

Nel presente capitolo verrà descritta la configurazione di loopback utilizzata per le misure e le soluzioni ad alcune problematiche, relative al networking stack di Linux e al software *sockperf*, a essa connesse.

3.1 Perché in loopback?

Nel caso si debbano effettuare misure del tempo di invio e ricezione di pacchetti dati UDP, utilizzando due interfacce di rete posizionate su macchine diverse, il primo problema è che i clock interni non sono sincronizzati.

Per ovviare a ciò, in analogia con i test condotti su NaNet (vedi par. 5.2.4), si è adoperata una configurazione in *loopback*: i due NIC ethernet presenti sullo stesso server sono collegati con il relativo cavo.

In questa maniera il riferimento temporale è lo stesso¹.

La macchina utilizzata per le prove era così configurata:

- Supermicro SuperServer 6016GT-TF con X8DTG-DF motherboard (Intel 5520-Tylersburg chipset), dual Intel Xeon X5570 @2.93 GHz CPU, Intel 82576 GbE and NVIDIA Fermi M2070 GPU (per brevità verrà chiamata d'ora in poi *sistema M2070*).

3.1.1 Sockperf

I test sono stati effettuati con *sockperf* (versione 2.5.208), software open source diffusamente utilizzato per benchmark di latenza e throughput.

L'uso standard consiste nel lanciare, ad esempio, sulla macchina A (indirizzo IP_S) il programma con il flag *server*, in modo che l'applicazione rimanga in ascolto su ip indicato e porta scelta:

¹Nelle CPU Intel moderne (dall'architettura Nehalem in poi) il Time-Stamp Counter (TSC) è sincronizzato tra tutti i core [15, Cap. 17.13].

```
sockperf server -i <ip_s> -p <port_s>
```

e sulla macchina B la versione *client* con l'opzione ping-pong o throughput:

```
sockperf pp (tp) -i <ip_s> -p <port_s> -t <time> -m <message>
```

Dove:

- pp: modalità ping-pong
 - i pacchetti vengono inviati al server e rimandati indietro;
 - il client calcola la latenza di comunicazione come la metà del tempo di andata e ritorno (Round Trip Time, RTT);
- tp: modalità throughput
 - bytes ricevuti nell'intervallo temporale fissato (numero totale di pacchetti per loro dimensione);
- ip_s: indirizzo ip a cui inviare i pacchetti;
- port_s: porta a cui vanno indirizzati;
- time: per quanti secondi proseguire a inviare pacchetti al server;
- message: lunghezza degli stessi in bytes.

Di default sockperf utilizza il protocollo UDP.

Nella nostra configurazione, i due NIC ethernet (eth1 ed eth2) sono, come già detto, presenti nella stessa macchina.

Per comodità assegniamo:

- client: eth1 → 10.0.0.1
- server: eth2 → 10.0.0.2

Loopback: il problema dei pacchetti *local*

Dalle prime prove di latenza si sono ottenuti valori di pochi microsecondi. Tale risultato è impossibile per questo genere di NIC con l'uso dello stack software standard.

Quello che avviene è che il Sistema Operativo riconosce entrambe le interfacce di rete come “locali” e di conseguenza i dati inviati dall'una all'altra in realtà non passano attraverso il cavo di collegamento ma semplicemente vengono effettuate copie direttamente in memoria.

Per aggirare questa impostazione di default, è stato necessario utilizzare un kernel versione 2.6.33 o superiore, poiché a partire da essa è possibile modificare alcuni parametri del kernel stesso utili allo scopo.

Si è dovuto poi intervenire sulle regole di instradamento (“routing”) e anche sul socket del client implementando una modifica al sorgente di sockperf.

Vediamo qui di seguito quali sono stati i passi necessari.

Innanzitutto per consentire che le interfacce di rete accettino pacchetti provenienti da indirizzi locali, si rende necessario modificare alcune impostazioni (“accept_local”) nel file di configurazione sysctl.conf.

Nota: il campo rp_filter (Reverse Path Forwarding) deve essere diverso da 0 affinché queste impostazioni abbiano effetto².

Devono essere, inoltre, impostate nuove regole per il routing, in modo che obbligatoriamente pacchetti inviati tramite l’interfaccia eth1 e destinazione 10.0.0.2 vengano trasmessi dall’interfaccia selezionata e ricevuti da eth2. In questa maniera i dati effettivamente viaggeranno lungo il cavo di collegamento. Processo non scontato poiché, come detto prima, entrambe le schede sono viste come interfacce locali.

Modifiche al file /etc/sysctl.conf

(sysctl -p per rendere operative le nuove impostazioni)

```
net.ipv4.conf.default.rp_filter = 1
net.ipv4.conf.default.accept_local = 1
net.ipv4.conf.eth0.accept_local = 1
net.ipv4.conf.eth1.accept_local = 1
net.ipv4.conf.all.accept_local = 1
net.ipv4.conf.lo.accept_local = 0
```

Modifiche alle tabelle di routing

```
ip rule flush
ip rule add pref 500 lookup local

ip rule add pref 10 iif eth1 lookup local
ip rule add pref 11 iif eth2 lookup local

ip rule add pref 100 to 10.0.0.1 lookup 100
ip rule add pref 101 to 10.0.0.2 lookup 101

ip route flush table 100;ip route add default dev
eth2 table 100
ip route flush table 101;ip route add default dev
eth1 table 101
```

In questo modo tutto il traffico che arriva dall’interfaccia indicata (*incoming interface*, iif) utilizzerà le tabelle selezionate seguendo la priorità delle regole di routing opportunamente impostata.

Associare socket del client e NIC

²<https://www.kernel.org/doc/Documentation/networking/ip-sysctl.txt>

Perché sia possibile inviare i pacchetti in configurazione loopback attraverso il cavo, è necessario inoltre che il socket creato dal client sia associato (“bound”) a una specifica interfaccia di rete (in questo caso eth1).

Le proprietà del socket si possono modificare attraverso la funzione `setsockopt` come accennato nel par. 2.1.1.

La versione standard di `sockperf` non fornisce questa opzione. Si è allora aggiunto nel file sorgente `SockPerf.cpp`:

```
if (!rc && (s_user_params.mode == MODE_CLIENT))
{
    if (setsockopt(fd, SOL_SOCKET, SO_BINDTODEVICE, "
        eth1", 4) < 0)
        printf("Not binding to Network Device");
    else {
        printf("Binding to Device");
    }
}
```

In questa maniera tutti pacchetti relativi al socket del client, in entrata e in uscita, passeranno per il NIC indicato.

Da notare che la funzione `bind()` non è adatta allo scopo poiché essa associa un particolare indirizzo IP a un socket, così che questo riceva e trasmetta pacchetti che riporteranno nel loro header quell’indirizzo come sorgente. Anche se questo fosse associato a eth1, il routing del kernel potrebbe decidere che la strada migliore per raggiungere l’IP di destinazione passa in realtà per eth2. Così il pacchetto, riportante l’IP sorgente associato a eth1, in realtà viene trasmesso da eth2.

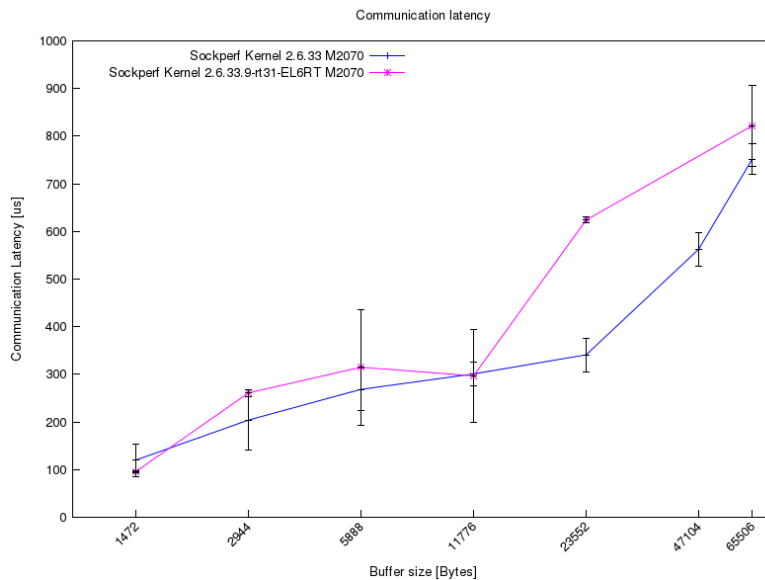


Figura 3.1. Latenze misurate in configurazione di loopback con kernel vanilla e real-time.

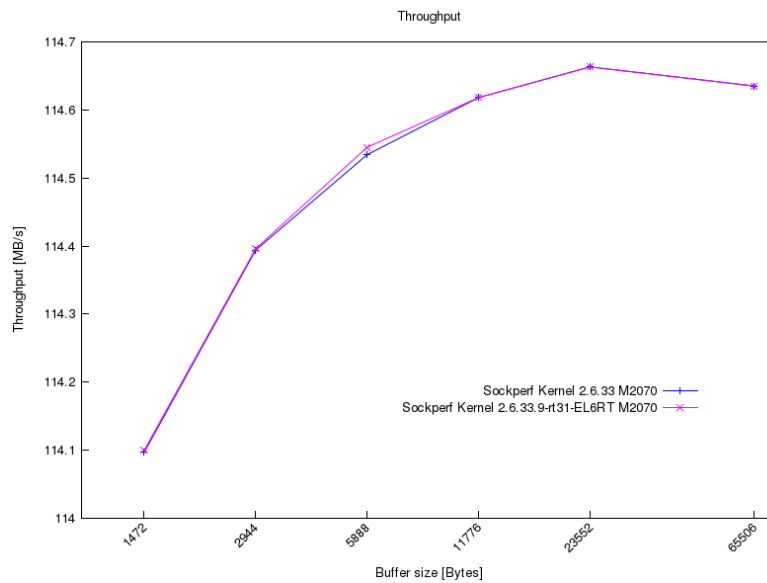


Figura 3.2. Valori del throughput in configurazione di loopback con kernel vanilla e real-time.

3.2 Test con due NIC Ethernet

Una volta correttamente configurato il loopback, si è proceduto ad effettuare i test di latenza e throughput.

Sul sistema M2070 è stata installata una versione di Linux CentOS 6.3, con la possibilità di scegliere due versioni del kernel:

- “vanilla” 2.6.33;
- “real-time” 2.6.33.9-rt31-EL6RT.

I test prevedono l’invio di pacchetti di dimensione crescente, multipli di 1472 bytes. La dimensione di un pacchetto è limitata dalla Maximum Transmission Unit (MTU) del protocollo ethernet utilizzato. Tale MTU è di 1500 bytes, a cui vanno sottratti 20 bytes per l’header IPv4 e 8 per quello UDP lasciandone 1472 per il payload.

Nelle figg. 3.1 e 3.2 sono illustrate rispettivamente throughput e latenze del sistema con kernel vanilla e real-time.

Con lo scopo di evitare ulteriori fonti di fluttuazioni, sono stati disabilitati due processi normalmente attivi sul SO Linux: `CPUspeed` che si occupa di gestire la frequenza dei processori in base al carico di lavoro e `IRQbalance` che ha il compito di distribuire gli interrupt in maniera dinamica fra le CPU del sistema.

Ciò che si osserva è una riduzione delle fluttuazioni ma nel contempo un aumento della latenza di comunicazione.

Da questo punto di vista è un risultato non positivo in quanto, come detto in precedenza, si ha a disposizione un budget temporale limitato.

La distribuzione delle latenze dei pacchetti può essere apprezzata negli istogrammi delle figg. 3.3 e 3.4.

Nel par. 5.1 vedremo come le variazioni delle latenze di comunicazione qui ottenute sono molto maggiori del tempo di elaborazione dell'algoritmo utilizzato nel LOTP (e delle sue fluttuazioni), non soddisfacendo così la richiesta di un comportamento real-time.

Le latenze elevate e il loro essere non deterministiche portano a concludere che una soluzione basata su NIC ethernet con device driver standard nel contesto di un Trigger di Livello 0 non possa essere facilmente implementata.

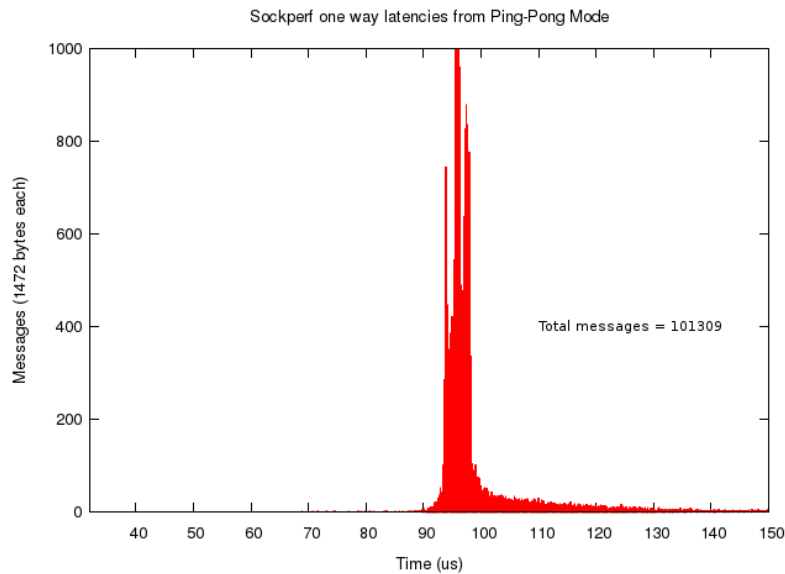


Figura 3.3. Distribuzione delle latenze dei pacchetti (kernel 2.6.33).

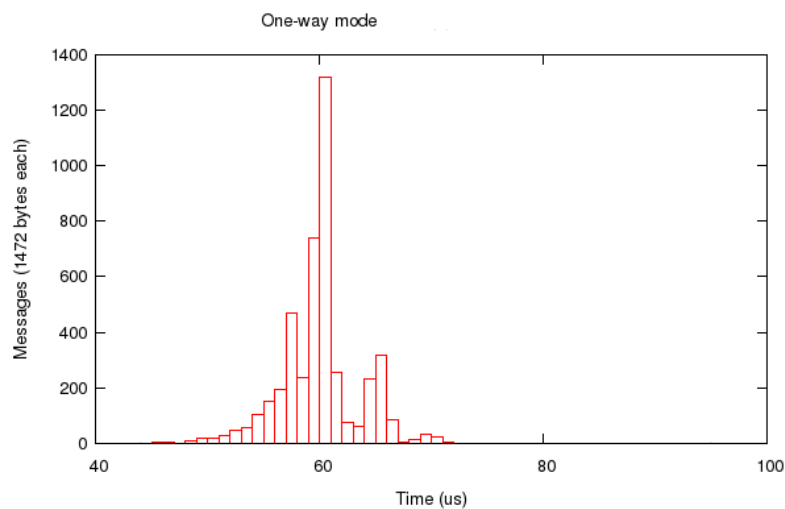


Figura 3.4. Distribuzione delle latenze dei pacchetti (kernel 2.6.33.9-rt31-EL6RT).

Capitolo 4

Architettura della scheda di rete NaNet

Al fine di realizzare un'interfaccia di acquisizione dati, con supporto alle GPU, per esperimenti di fisica delle alte energie, la scheda di comunicazione NaNet integra alcune delle caratteristiche del progetto APEnet+, sviluppato dall'“Istituto Nazionale di Fisica Nucleare” (INFN), per l'interconnessione a bassa latenza dei nodi di un cluster ibrido CPU/GPU a topologia toroidale 3D [16].

In questo capitolo verranno presentati i blocchi logici che costituiscono NaNet, gli elementi distintivi e le prestazioni, che la rendono candidata all'utilizzo nella catena di trigger dell'esperimento NA62.

Verranno, inoltre, presentati alcuni esempi di come la flessibilità del suo design ne consenta l'utilizzo in ambiti diversi.

4.1 La logica della scheda

L'idea alla base dello sviluppo di NaNet è quella di evitare che l'host sia coinvolto nelle comunicazioni tra rete e GPU, garantendo una bassa latenza e un throughput sufficiente per le applicazioni.

Altro aspetto importante è la disponibilità di una *custom logic* che permette di effettuare un preprocessamento dei dati.

Dal punto di vista hardware, la scheda è basata sulla FPGA Stratix IV GX FPGA Dev Kit e utilizza una connessione PCIe gen. 2.0 (fig. 4.1).

Lo schema dei blocchi logici che ne costituiscono l'architettura è in fig. 4.2.

Questa impostazione garantisce che NaNet sia modulare e altamente configurabile a seconda delle necessità di utilizzo.

Esamineremo qui di seguito i compiti dei singoli blocchi.

4.1.1 *Interfaccia I/O*

In NaNet il link fisico (Physical Link Coding) è implementato attraverso un modulo Altera Triple Speed Ethernet Megacore (TSE MAC), che permette di supportare un canale ethernet da 10/100/1000 Mbps. Questo tipo di connessione si affianca a



Figura 4.1. La scheda NaNet in funzione.

quella “standard” con APElink (canali usati da APENet+ con banda teorica di 34 Gbps ciascuno).

La gestione del protocollo ethernet è necessaria poiché nell’ambito del TDAQ di NA62 il trasferimento in rete dei dati avviene mediante connessioni di questo tipo e utilizzando datagrammi UDP (vedi par. 1.3).

Tale protocollo viene gestito in NaNet da un “UDP offloader” (*Protocol Manager*) che riceve i dati provenienti dal TSE MAC e offre un canale a 32 bit con una banda di 6.4 Gbps (sei volte le richieste del link GbE).

In questa maniera il Micro controller sulla FPGA (un Nios II a 200 MHz) viene liberato dalla suddetta incombenza.

Il flusso dati proveniente dall’UDP offloader è indirizzato al “NaNet Controller” (NaNet CTRL) che si occupa di incapsularlo secondo il protocollo standard ereditato da APENet+.

Questo è basato su *word* da 128bit e il pacchetto dati è formato da header, footer e un payload che ha una dimensione massima di 4096 bytes.

I pacchetti dati poi vengono inviati dal NaNet CTRL alla Network Interface (par. 4.1.3).

4.1.2 Router

Questo blocco si occupa dell’instradamento dei pacchetti, svolgendo una funzione di switch tra quelli provenienti dai link e la Network interface.

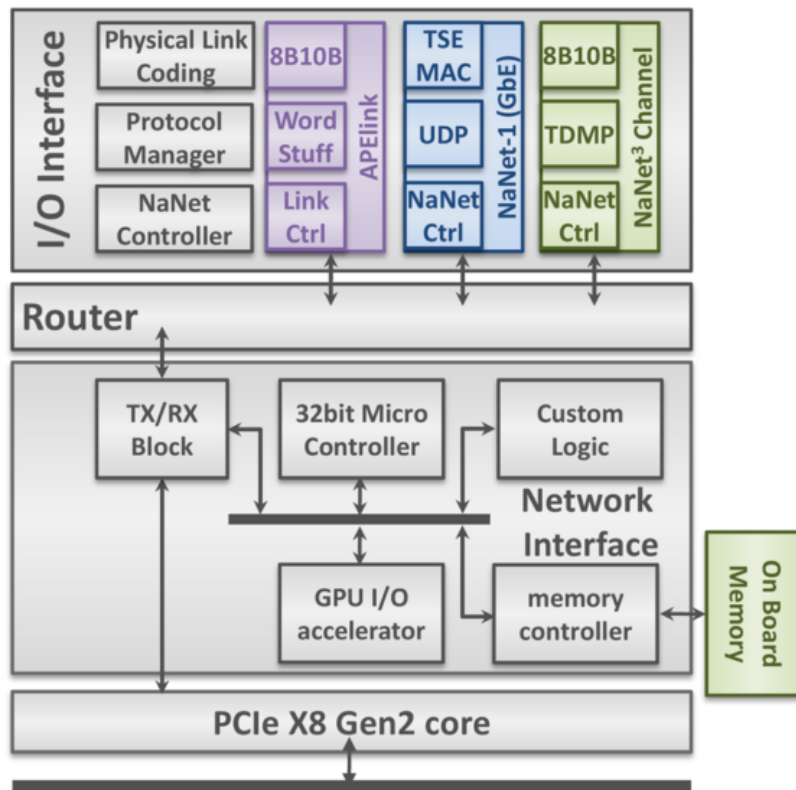


Figura 4.2. Schema logico della scheda di comunicazione NaNet.

4.1.3 Network interface

Sono due i principali compiti svolti da questo elemento dell'architettura:

- dal punto di vista della trasmissione, inoltra i dati provenienti dalla connessione PCIe (host e GPU) alle porte di destinazione;
- dal punto di vista della ricezione, fornisce supporto hardware al protocollo RDMA (Remote Direct Memory Access) permettendo la trasmissione di dati attraverso la rete senza coinvolgimento del processore.

Quando sono presenti GPU di classe Fermi o Kepler, la Network interface permette di accedere direttamente alla memoria dei device, sfruttando la modalità di comunicazione peer-to-peer (GPUDirect v2) attraverso il "GPU I/O Accelerator" (fig. 4.2).

4.1.4 Micro controller

Un'implementazione dell'RDMA di tipo *zero-copy* (ovvero senza spostamento di dati dallo spazio utente al kernel e viceversa) e che bypassi CPU e RAM dell'host richiede che la scheda di comunicazione possa intervenire in maniera autonoma sui buffer utilizzati dall'applicazione.

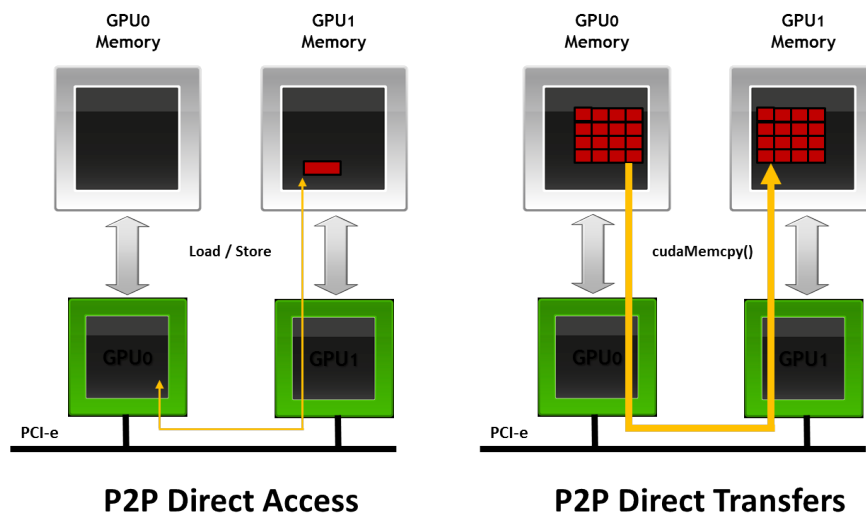


Figura 4.3. Comunicazione GPUDirect V2 peer-to-peer tra processori grafici sullo stesso root PCIe root complex.

In questo modo calcolo e comunicazione risultano due operazioni completamente separate.

Oltre a questo, va tenuto in considerazione che nel SO Linux un processo utente non può accedere direttamente alla memoria fisica ed è necessario utilizzare indirizzi di memoria virtuale.

Il Micro controller (μC) sulla FPGA, a cui sono demandati questi compiti, implementa in un firmware:

- una *Look-Up Table* (LUT) con indirizzi di memoria e dimensioni dei buffer registrati dall'applicazione (possibili destinazioni di trasferimenti dati da nodi remoti);
- una *Page Table* (PT) per suddividere i buffer in pagine di memoria (imitando la struttura dati di Linux) ed effettuare la conversione degli indirizzi da virtuali a fisici.

All'inizializzazione di un'applicazione, le LUT e PT sono allocate dal μC e aggiornate dal kernel driver di NaNet durante la successiva esecuzione.

Sempre il Micro controller provvede alla loro gestione durante la fase di invio e ricezione dei messaggi.

4.1.5 GPU I/O Accelerator

Abbiamo visto nel par. 2.2 che una GPU è un device a sé stante, con unità di calcolo, memoria e relativa logica di gestione separata dall'host.

A causa di questa impostazione architetturale, la copia di dati provenienti dalla rete nella memoria del processore grafico risulta essere un collo di bottiglia per applicazioni che necessitino di bassa latenza, essendo richiesto il passaggio intermedio per l'host con tutte le conseguenze del caso (fig 4.4): $NIC \rightarrow CPU - RAM \rightarrow GPU$

Dall'architettura NVIDIA Fermi è disponibile un protocollo proprietario (GPUDirect V2 *peer-to-peer*) che permette di scambiare dati direttamente tra GPU connesse allo stesso *PCIe root complex* senza mediazione dell'host (fig. 4.3).

Si rende così possibile per una GPU leggere/scrivere la memoria di un altro processore grafico, a condizione che la topologia PCIe sia rispettata.

Il *GPU I/O Accelerator* consente alla scheda NaNet di prendere parte alla transazione *peer-to-peer* con schede di classe Fermi o Kepler.

Questa è una delle caratteristiche ereditate da APENet+ che, come accennato nel par. 2.2.2, è stato il primo dispositivo non-NVIDIA a sfruttare questa possibilità per trasferimento dati su una rete, permettendo comunicazioni tra GPU su nodi diversi con latenze estremamente ridotte (figg. 4.4 e 4.5) [17].

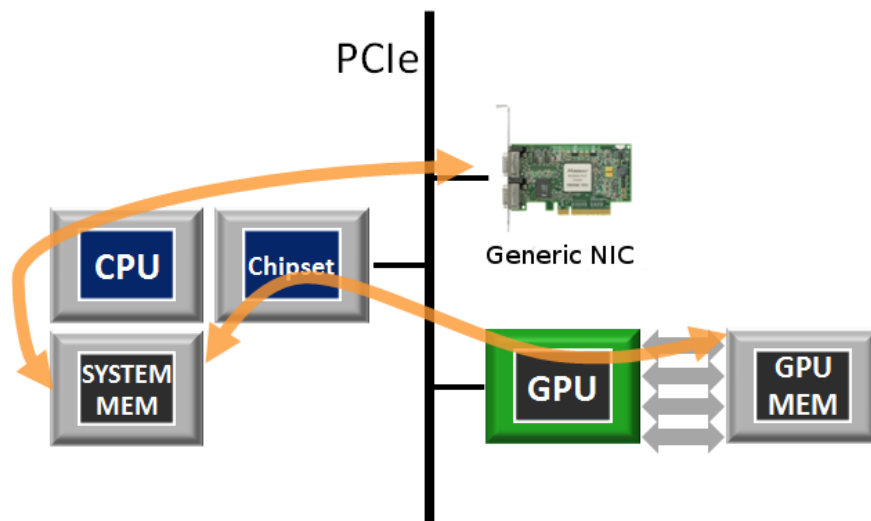


Figura 4.4. Metodo standard per inviare un pacchetto dati proveniente dalla rete alla memoria GPU.

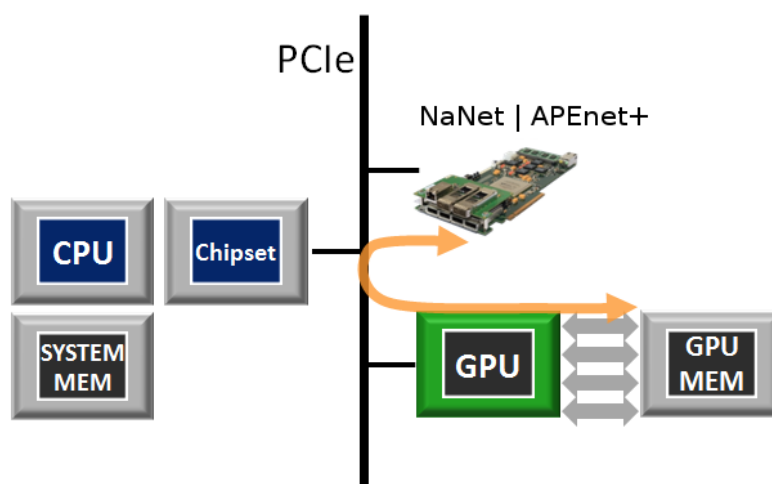


Figura 4.5. Comunicazione *peer-to-peer* tra le schede NaNet/APENet+ e la GPU.

4.1.6 La *Custom logic*

Questo elemento permette di intervenire direttamente sui dati, eseguendo operazioni su di essi, riordinandoli, ecc. Ovviamente in questo modo si garantisce una grande adattabilità agli ambiti di utilizzo.

Nel caso specifico di NA62, verrà effettuato un opportuno padding dei dati ricevuti dal RO board prima del trasferimento DMA per rispettare l'allineamento richiesto dalla memoria della GPU.

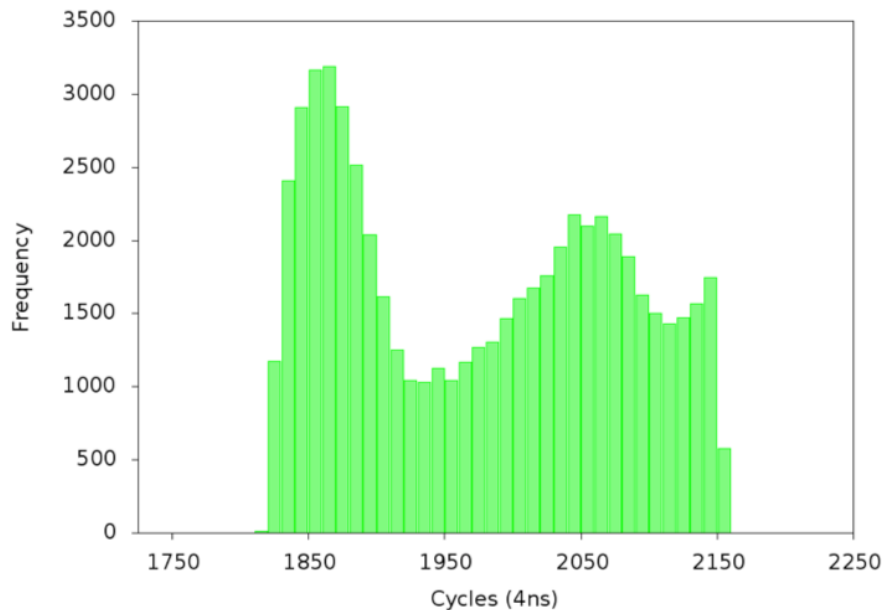


Figura 4.6. Distribuzione in frequenza delle latenze di attraversamento dei sottosistemi di nanet (per 60000 pacchetti UDP da 1472 bytes).

4.1.7 Performance

Di seguito verranno descritti i risultati delle misure riguardanti latenza e banda della scheda.

Latenza di attraversamento

Avendo descritto i vari sottosistemi che compongono NaNet, è utile qui dare una descrizione della latenza a cui va incontro un pacchetto dati durante il loro attraversamento (in particolare UDP offloader, Micro Controller e blocco Rx della Network interface).

Ogni fase introduce ovviamente un ritardo che deve essere ovviamente il più ridotto e determinato possibile.

Tali misure sono state effettuate utilizzando contatori (con risoluzione di 4 ns) [18] inseriti in un footer (appositamente disegnato) durante i vari stadi del processo a cui è soggetto un pacchetto dati: dall'ingresso nel NaNet CTRL al completamento della

transazione DMA su PCIe. I risultati sono in fig. 4.6 e riguardano 60000 pacchetti UDP.

La variabilità è dovuta al Nios II μ C impegnato nella gestione degli indirizzi di memoria e della loro traduzione da virtuali a fisici e viceversa. Tale comportamento, ovviamente non è desiderabile. Per ridurre il carico di lavoro derivante da queste procedure, è in fase di completamento l'implementazione di un blocco logico ulteriore che svolga la funzione di Translation Lookaside Buffer (TLB). Questo mantiene in una *Content addressable memory* (CAM) quante più associazioni indirizzo virtuale/fisico possibili, fino a 4 porte, ciascuna da 32 ingressi, rendendo più veloce quindi gli accessi. Solo nel caso in cui un certo indirizzo non fosse disponibile nella CAM, verrà consultata la Page Table.

Latenza della comunicazione e banda

I test sono stati effettuati in configurazione di loopback: i pacchetti vengono inviati dalla scheda ethernet dell'host e ricevuti da NaNet. In questa maniera il riferimento temporale dato dal registro TSC del processore è unico, come descritto nel par. 3.1.

In particolare:

- per la latenza, viene misurato l'intervallo temporale tra quando viene segnalato all'applicazione il riempimento del buffer di ricezione (multiplo della dimensione del payload del pacchetto UDP) e l'istante in cui è stato inviato il primo pacchetto del gruppo necessario a riempirlo;
- per la banda, fissato il numero di pacchetti da inviare (multiplo intero della dimensione dei buffer di ricezione), è il rapporto tra il totale dei bytes trasmesso e l'intervallo temporale intercorso tra l'istante in cui viene segnalato il riempimento dell'ultimo buffer di ricezione e l'invio del primo pacchetto UDP.

Le due macchine utilizzate sono dotate di GPU appartenenti a generazioni diverse (vedi par. 2.2.4):

- Supermicro SuperServer 6016GT-TF con X8DTG-DF motherboard (Intel 5520-Tylersburg chipset), dual Intel Xeon X5570 @2.93 GHz CPU, Intel 82576 GbE and NVIDIA Fermi M2070 GPU (*sistema M2070*);
- Supermicro SuperServer 7047GR-TPRF con X9DRG-QF motherboard (Intel C602-Patsburg chipset), dual Intel Xeon E5-2609 @2,40 Ghz CPU, Intel i350 GbE and NVIDIA Fermi K20Xm GPU (*sistema K20Xm*).

I grafici delle figg. 4.7 e 4.8 riportano i risultati delle misure di latenza e banda al variare della dimensione dei buffer di ricezione.

Risulta evidente come entrambe le caratteristiche siano soggette a fluttuazioni estremamente ridotte.

Non ci sono, inoltre, sostanziali differenze tra i sistema M2070 e K20Xm per quanto riguarda i valori misurati.

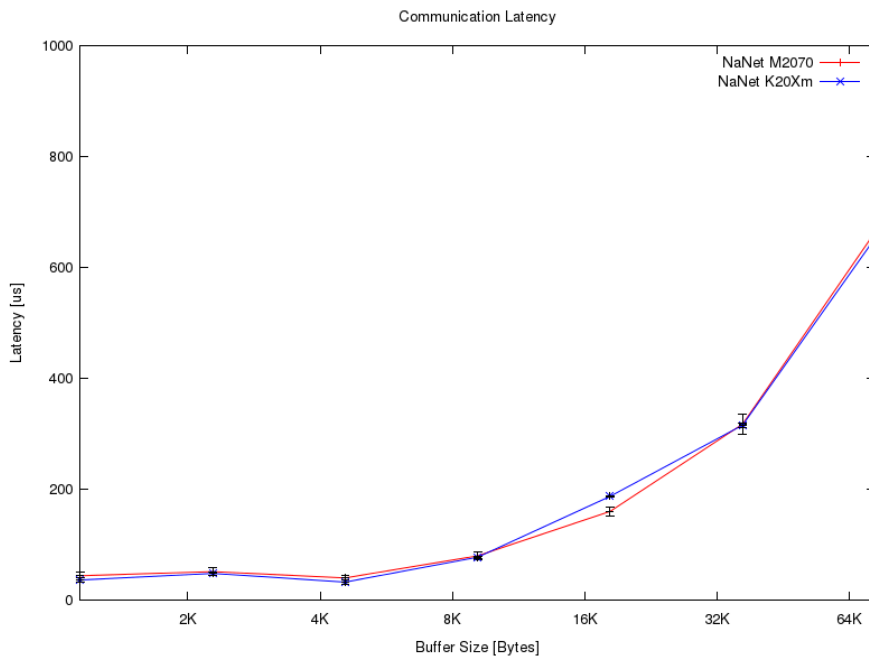


Figura 4.7. Test di latenza al variare della dimensione del buffer di ricezione.

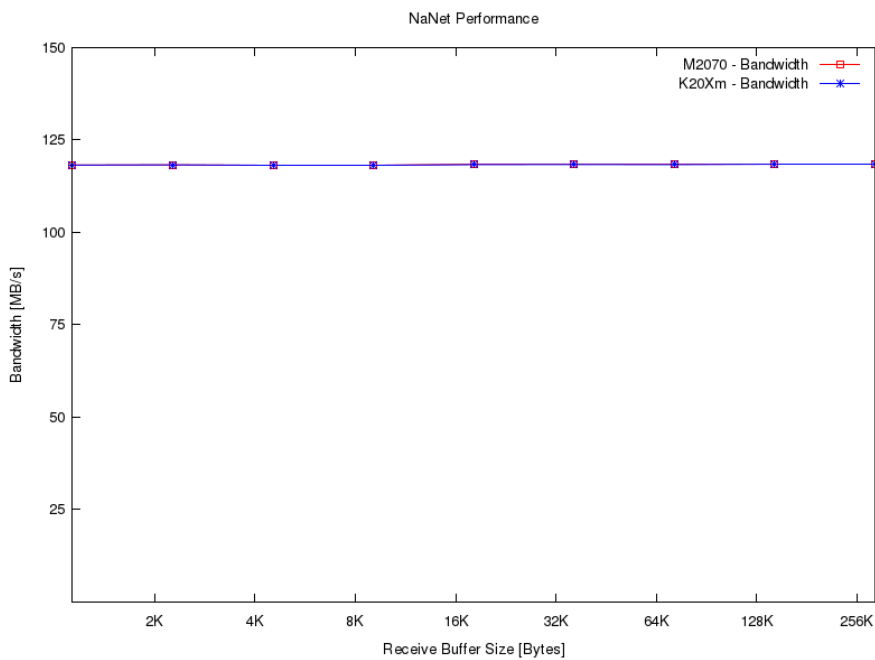


Figura 4.8. Banda di NaNet in funzione della dimensione del buffer di ricezione.

4.1.8 Componenti software

Organizzazione dei buffer di ricezione sulla GPU in una lista circolare

Per interagire con la GPU è essenziale la creazione di buffer di ricezione a cui NaNet possa accedere per trasferire il flusso di pacchetti provenienti dalla rete, durante tutto il tempo in cui l'applicazione è attiva.

Questi sono organizzati in una lista circolare e chiamati *CLOP* (Circular List Of Persistent buffers).

L'utente può definire il numero degli elementi della lista e la dimensione del singolo elemento come un multiplo delle dimensioni del datagramma UDP che verrà utilizzato dall'applicazione (fig. 4.9).

Essendo una lista circolare, se N sono gli elementi CLOP (ciascuno contenente M datagrammi UDP), dopo $M * N$ pacchetti si comincerà a riscriverla partendo dal primo buffer della lista.

Cambiando M è possibile, fissata la frequenza dei dati in ingresso, stabilire ogni quanto verrà invocato il kernel CUDA che consumerà il contenuto del buffer.

Al variare di N è possibile calibrare il funzionamento del sistema evitando che i dati di un buffer vengano sovrascritti prima che questi siano stati effettivamente utilizzati per il calcolo. Come si vedrà nel par. 5.2.2 il tempo di esecuzione del kernel di interesse è assai stabile, rendendo così agevole stabilire quale sia il numero di buffer adatto alle condizioni del trigger.

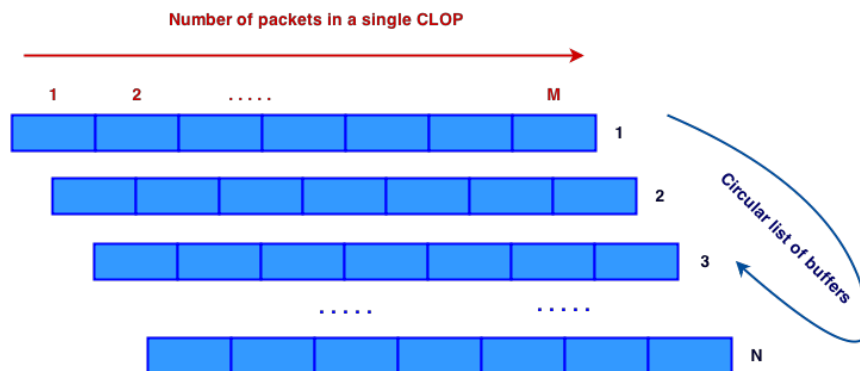


Figura 4.9. Lista circolare di buffer registrati sulla GPU.

Applicazioni su host e NaNet

Per il funzionamento di NaNet è necessaria l'esecuzione di applicazioni distinte su host e μC .

Nel primo caso sono presenti:

- kernel driver per controllare NaNet e GPU;
- una API (*Application programming interface*), costituita da una serie di librerie che permettono l'apertura/chiusura della scheda, la gestione dei CLOP

(allocazione e deallocazione, “pinning”, passaggio dell’indirizzo virtuale all’applicazione utente) e la segnalazione di eventi di ricezione nei buffer registrati in modo che l’applicazione possa utilizzare i dati (ad esempio lanciare un kernel CUDA).

Nel Nios, il software dovrà occuparsi della configurazione del NIC, della generazione degli indirizzi virtuali di destinazione all’interno dei CLOP per i pacchetti in arrivo e della risoluzione di quelli fisici per consentire le transazioni DMA verso la memoria della GPU.

4.1.9 NaNet: una famiglia di schede

Come abbiamo visto nel par. 4.1.1, l’interfaccia di I/O di NaNet supporta connessioni APElink e GbE. Per la natura modulare della scheda, però, risulta agevole implementare altri link e funzionalità, in base alle necessità.

Questa flessibilità ha portato allo sviluppo di una famiglia di schede:

- NaNet-1: la versione utilizzata per i test di questa Tesi e descritta nel presente capitolo;
- NaNet-10: utilizzerà una scheda Stratix V, con supporto al PCIe gen. 3.0. Per venire incontro alle maggiori richieste di banda della configurazione finale del trigger, metterà a disposizione due canali 10GbE full duplex attraverso una scheda Terasic Dual XAUI To SFP+, installata sul connettore HSMC (*High Speed Mezzanine Card*). Inoltre, la custom logic verrà sfruttata maggiormente poiché i dati dei fotomoltiplicatori del RICH saranno suddivisi tra quattro diversi pacchetti, ciascuno proveniente da una TEL62 (par. 1.3), e i link GbE verranno poi aggregati da uno switch in un unico canale 10GbE che avrà come destinazione NaNet. Sarà compito, quindi, della custom logic ricostruire l’evento, intervenendo sui pacchetti distinti, prima che questo possa essere trasmesso alla GPU per l’esecuzione dell’algoritmo di trigger.
- NaNet³: sviluppata nell’ambito dell’esperimento KM3 (telescopio per neutrini astrofisici) [19], avrà il compito di ricevere dati dai fotomoltiplicatori attraverso connessioni ottiche e di provvedere alla sincronizzazione dell’elettronica a mare con quella di riva [20].

Capitolo 5

NaNet come elemento del Trigger di Livello 0

Nel paragrafo 1.4 erano state introdotte le varie fasi del trigger di livello 0 del RICH dell'esperimento NA62 basato su GPU, entriamo qui nei dettagli del software utilizzato per i test in loopback e del kernel CUDA che implementa l'algoritmo MATH. Verranno presentati i risultati dei test effettuati sia con un NIC ethernet standard che con NaNet. La scheda si dimostrerà adatta ad essere utilizzata nella catena di trigger.

5.1 L0TP con NIC ethernet standard

Nel Cap. 3, è stata esaminata la connessione tra due schede ethernet standard. Riportiamo qui di seguito i risultati ottenuti da sockperf relativo alla fig. 3.3, con pacchetti UDP da 1472 bytes.

```
sockperf: Summary: Latency is 99.129 usec <---
sockperf: Total 100816 observations; each percentile contains
          1008.16 observations
sockperf: ---> <MAX> observation = 657.743 <---
sockperf: ---> percentile 99.99 = 474.758
sockperf: ---> percentile 99.90 = 201.321
sockperf: ---> percentile 99.50 = 163.819
sockperf: ---> percentile 99.00 = 149.694 <---
sockperf: ---> percentile 95.00 = 116.730
sockperf: ---> percentile 90.00 = 105.027
sockperf: ---> percentile 75.00 = 97.578
sockperf: ---> percentile 50.00 = 96.023
sockperf: ---> percentile 25.00 = 95.775
sockperf: ---> <MIN> observation = 64.141 <---
```

Osserviamo come l'estrema variabilità dei risultati non consente di soddisfare le esigenze di una latenza stabile nel tempo, come richiesto dal trigger di livello 0.

In fig. 5.1 i valori corrispondenti al minimo, al 99 percentile e alla media, vengono confrontati con il tempo di esecuzione di un kernel che implementi l'algoritmo MATH (la variabilità del tempo di esecuzione del kernel non è apprezzabile alla scala del

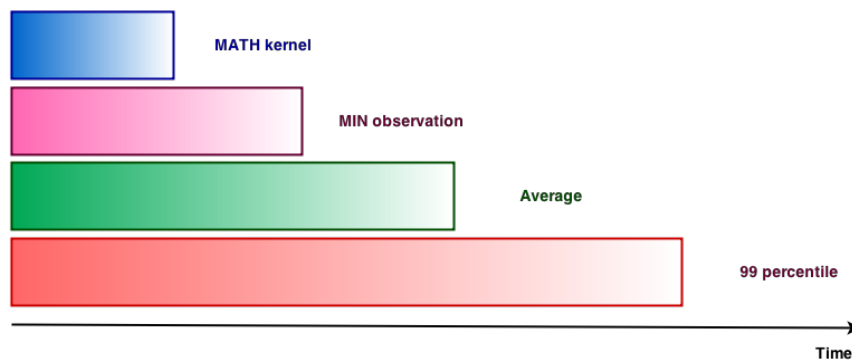


Figura 5.1. Confronto del tempo di esecuzione dell'algoritmo MATH rispetto alla variabilità della latenza di comunicazione in un sistema che utilizzi un NIC ethernet standard.

grafico), per un pacchetto dati di dimensione confrontabile (vedi paragrafi 1.2.1 e 5.2.4). Le latenze di comunicazione variano tra due e cinque volte questo valore.

La proporzione tra le varie fasi della comunicazione del trigger in questa configurazione, come evidenziato nel grafico di fig. 5.2, è sbilanciata verso i tempi di comunicazione. Risulta quindi essenziale ridurli per essere sicuri di rientrare nella latenza richiesta dall'esperimento. Inoltre, è da ricordare che nel caso dell'utilizzo di NaNet non è necessario effettuare copie dei dati tra host e device, il che comporta un ulteriore risparmio di tempo.

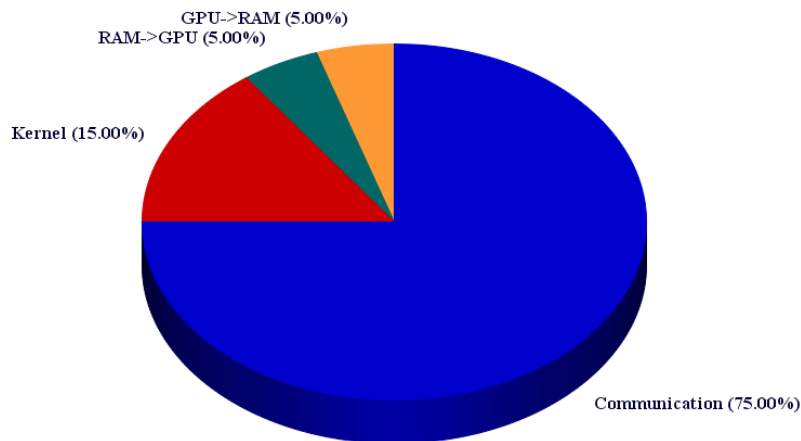


Figura 5.2. Ripartizione latenze fra le varie fasi del trigger.

5.2 L0TP con NaNet

I test in loopback sono stati effettuati inviando a NaNet pacchetti UDP il cui payload, generato tramite Monte Carlo, è conforme al protocollo di Read-Out del RICH di NA62.

- registra i CLOP sulla GPU, specificando dimensione del datagramma UDP (*size*), dimensioni del singolo buffer (multiple della *size* in bytes del pacchetto, ad esempio $M * size$) e numero di buffer nella lista circolare (N).

Si entra poi nel loop principale (configurato in modo che le ripetizioni corrispondano al numero di volte che si desidera venga lanciato il kernel CUDA sulla GPU):

- viene letto il registro TSC (T_{send});
- si inviano M pacchetti UDP (in modo da riempire un CLOP) dalla scheda ethernet il cui datagramma contiene i dati generati per il RICH nel formato sopra descritto (in particolare di *size* pari a 1168 bytes);
- attende il segnale della ricezione dei dati in un buffer CLOP;
- viene letto il registro TSC (T_{comp});
- viene lanciato il kernel CUDA con l'algoritmo MATH in maniera sincrona con l'host;
- quando è completata l'esecuzione del calcolo sulla GPU, viene letto nuovamente il registro TSC (T_{kernel});
- vengono calcolate e registrate in un array le latenze $L_{comm} = T_{comp} - T_{send}$ e $L_{calc} = T_{kernel} - T_{comp}$.

La fase finale comprende chiusura del NIC, deregistrazione dei buffer CLOP e deallocazione dei buffer di appoggio sull'host.

È importante segnalare che il kernel CUDA è lanciato in maniera sincrona, cioè in modo che l'applicazione non possa proseguire finché la GPU non abbia terminato l'elaborazione e restituito il controllo all'host.

Questa metodologia è ovviamente utile per le misure di latenza ma è penalizzante in condizioni operative poiché il lancio e l'uscita dal kernel hanno un costo in termini di tempo.

Nelle condizioni operative del trigger, calcolo e comunicazione si sovrapporranno, grazie al motore RDMA di NaNet.

Kernel CUDA MATH

L'algoritmo di Crawford per la determinazione di raggio e centro delle circonferenze implementato dal kernel MATH è stato illustrato nel par. 1.2.1.

Ricordiamo che nella programmazione CUDA ogni warp (32 threads) è indipendente dagli altri e ciascuno di essi esegue un'istruzione alla volta (par. 2.2.3).

Tenendo conto del particolare formato dati utilizzato, la soluzione più pratica è quella di mappare un intero evento, costituito da 32 hits, su un warp.

In questo modo, più warp (eventi) possono essere gestiti in maniera naturale da ogni singolo blocco (nella fig. 5.4 ogni hit, indicato con un numero intero all'interno del quadrato, è assegnato a un singolo thread), che risulterà quindi bidimensionale.

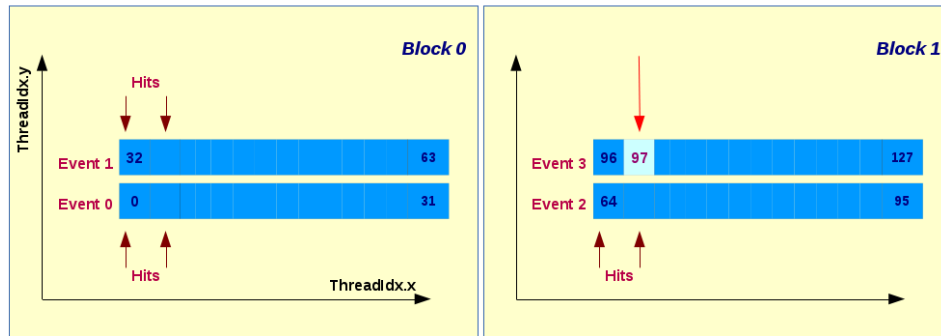


Figura 5.4. Nello schema, esempio di come i dati del RICH vengono ripartiti fra thread e block del kernel CUDA.

In fig. 5.5 vediamo come i tempi di calcolo del kernel MATH risentano delle differenze architetturali tra le due generazioni di GPU, M2070 e K20Xm.

All'aumentare della dimensione del CLOP, e quindi del numero di eventi da elaborare contemporaneamente, la latenza per la scheda di classe Fermi cresce in maniera più decisa rispetto alla Kepler.

È un risultato che era lecito aspettarsi in quanto le modifiche apportate all'architettura Kepler (maggiore numero di unità di calcolo, scheduler capaci di gestire un numero di warp e istruzioni per ciclo superiore alla classe Fermi, banda maggiore in ogni livello della memoria) la rendono capace di migliori prestazioni rispetto alla generazione precedente.

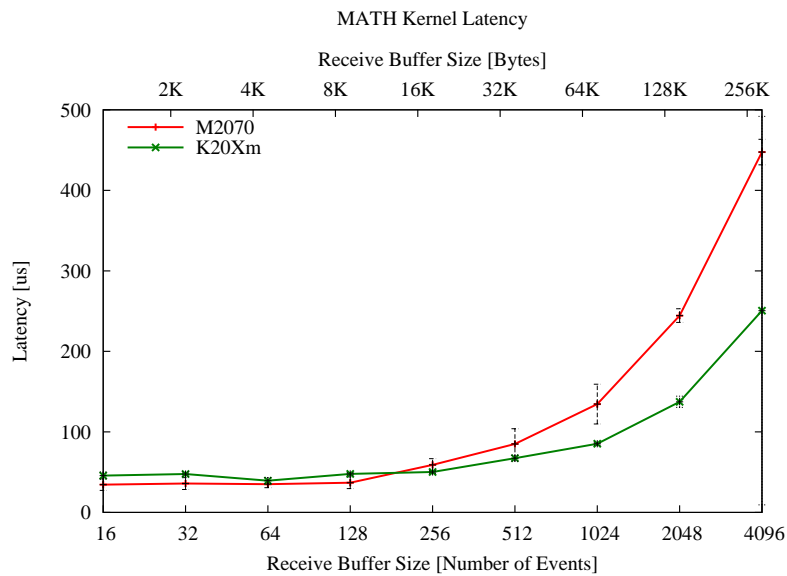


Figura 5.5. Latenze relative all'esecuzione del kernel MATH, sulle GPU NVIDIA M2070 e K20Xm, al variare del numero di eventi da elaborare.

5.2.3 Misure con oscilloscopio

Per verificare la completa integrazione di NaNet nel trigger di livello 0, si è proceduto, tra l'altro, ad effettuare una serie di test collegando la scheda stessa e una TEL62 a un oscilloscopio.

Sono stati quindi inviati dalla TEL62 gruppi di 32 pacchetti UDP verso la scheda, configurata con dei CLOP (vedi par. 4.1.8) la cui dimensione è 8 volte quella del datagramma contenente i dati.

In fig. 5.6 si possono apprezzare i segnali corrispondenti a ogni invio da parte della RO board (linea rossa) e i quattro completamenti della transazione PCIe da parte di NaNet (linea gialla), che corrispondono al completamento della trasmissione dei dati degli 8 pacchetti nel buffer di ricezione in memoria GPU.

Le misure così ottenute sono in buon accordo con quelle effettuate senza oscilloscopio e riportate nel paragrafo successivo.



Figura 5.6. Test effettuato collegando NaNet a un oscilloscopio.

5.2.4 Misure nella configurazione standard del trigger

In precedenza abbiamo definito quale sia la configurazione del trigger in cui NaNet andrà ad operare. Descriviamo qui di seguito i risultati dei test necessari volti a valutare le prestazioni di banda e latenza.

In fig. 5.7 vengono riportate le misure di banda e throughput (considerando cioè anche il tempo di calcolo del kernel CUDA) ottenute con i due sistemi M2070 e K20Xm.

Si può osservare come, in entrambi, la banda venga saturata già a partire da buffer di ricezione contenenti 128 eventi, ovvero 8 datagrammi per ogni buffer CLOP,

che avrà quindi una dimensione di circa 8K. Vengono così processati 1.6 Meventi/s in maniera costante.

Per quanto riguarda i tempi relativi a comunicazione e calcolo, nei grafici delle figg. 5.8 e 5.9 possono essere confrontati sia nel caso di GPU classe Fermi che Kepler.

La variabilità delle latenze risulta estremamente ridotta e costante nell'intervallo di dimensioni dei buffer considerati.

Osserviamo, inoltre, che in entrambi i casi i requisiti:

- latenza totale inferiore 1 ms (vedi figg. 5.8 e 5.9);
- massimizzazione del numero di eventi processati dalla GPU;

sono soddisfatti da buffer di dimensione pari a 1024 eventi (CLOP contenenti 64 datagrammi).

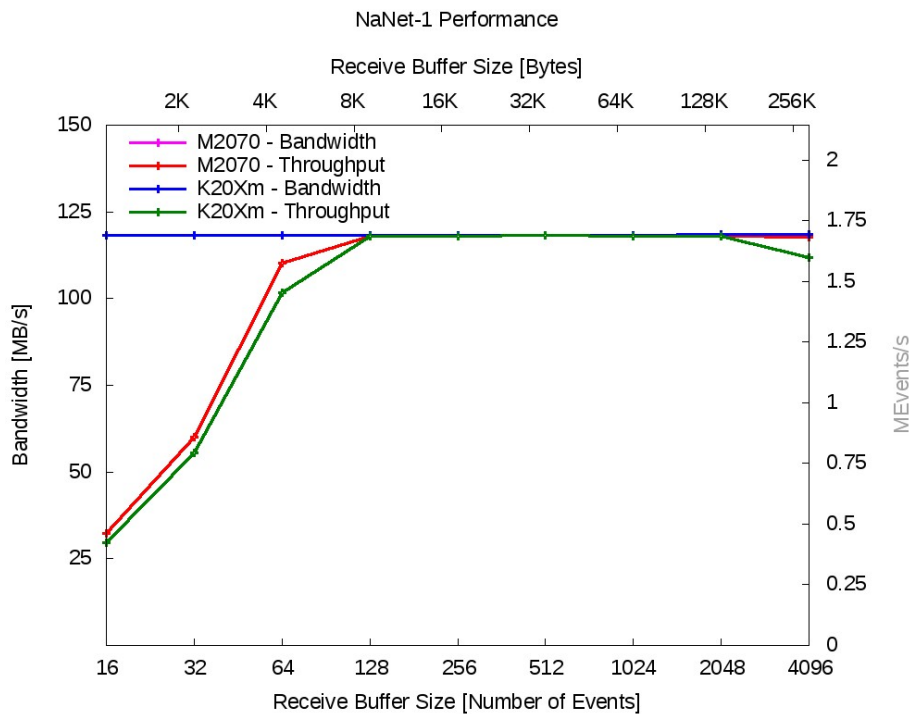


Figura 5.7. Misure di banda e throughput in funzione del numero di eventi (dimensione dei CLOP).

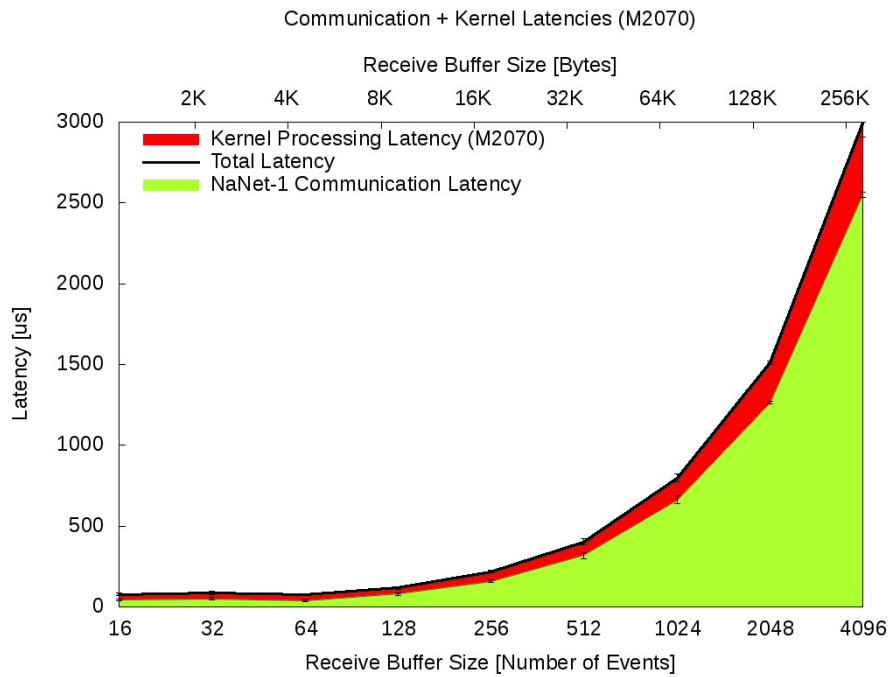


Figura 5.8. Latenze di comunicazione e calcolo per il sistema M2070.

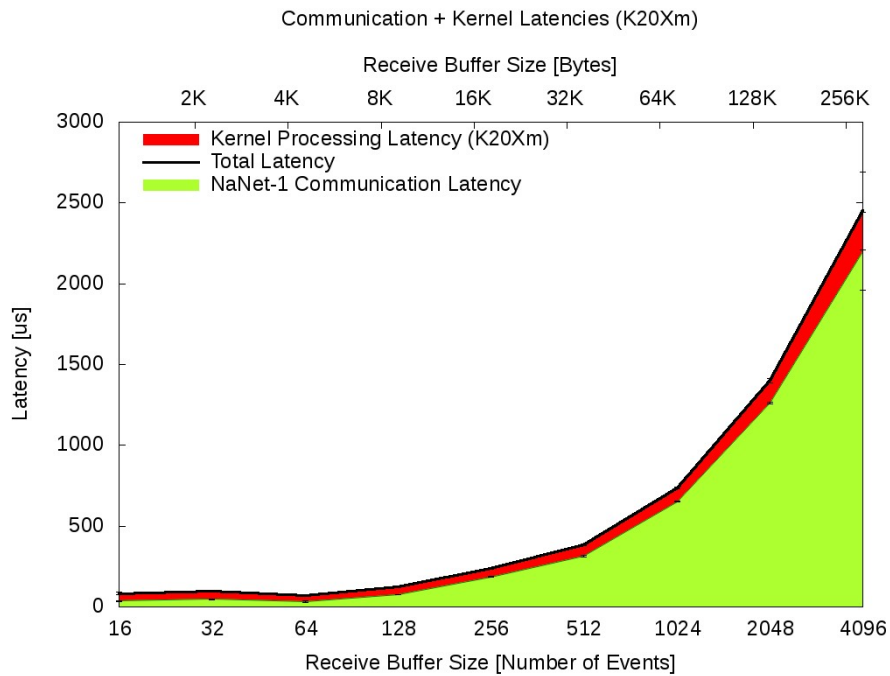


Figura 5.9. Latenze di comunicazione e calcolo per il sistema K20Xm.

Conclusioni

Il progetto di ricerca relativo all'utilizzo delle GPU nel trigger di livello 0 dell'esperimento di NA62 è il primo di questo tipo.

Le novità tecnologiche introdotte nell'ambito dei processori grafici negli ultimi anni quali: comunicazione peer-to-peer, elevata potenza di calcolo, facilità di programmazione, hanno aperto la strada al loro utilizzo nell'ambito dei trigger di basso livello degli esperimenti di fisica delle alte energie.

Non è, però, sufficiente affidare la comunicazione tra il Read-out del RICH e il trigger a una scheda ethernet standard. Come abbiamo visto, tale componente non è adatto a soddisfare le condizioni di latenza richieste dalle specifiche dell'esperimento.

Sfruttando il protocollo GPUDirect V2, la scheda di interconnessione NaNet sviluppata dall'“Istituto Nazionale di Fisica Nucleare” permette di trasferire direttamente in memoria GPU i pacchetti ricevuti dalla rete, senza intervento da parte della CPU o della RAM dell'host, riducendo grandemente le latenze associate alle comunicazioni e le loro fluttuazioni.

La caratterizzazione effettuata nella configurazione di loopback, su sistemi con GPU di classe NVIDIA Fermi e Kepler, conferma che è possibile per un sistema così concepito avere tempi di risposta deterministici inferiori a 1 ms. La banda relativa alla comunicazione, inoltre, viene saturata anche con buffer di dimensioni contenute. Si riescono così a processare in maniera stabile fino a ~ 1.6 Meventi/s.

Sono stati condotti con successo test di interoperabilità con la RO board TEL62 dimostrando così la possibilità di integrare NaNet nella catena di acquisizione e trigger dell'esperimento NA62.

Un'importante considerazione da fare è che, per come è stata condotta la misura, tali risultati positivi sono stati ottenuti utilizzando una configurazione penalizzata dal punto di vista delle prestazioni poiché risultano serializzate le operazioni di ricezione dei buffer e invocazione del kernel CUDA per l'esecuzione dei calcoli.

Nelle normali condizioni di funzionamento del trigger, grazie al motore RDMA della scheda NaNet, il trasferimento dati si sovrappone all'esecuzione del kernel GPU con prestazioni migliori di quelle effettivamente misurate.

Come ulteriore sviluppo futuro, NaNet (vedi 4.1.9) evolverà verso NaNet-10 che implementerà la possibilità di utilizzare link a 10GbE e la connessione PCIe gen. 3.0, per supportare la maggiore banda richiesta a regime dall'esperimento NA62.

Verrà, inoltre, sfruttata la custom logic per ricostruire gli eventi (i cui hit saranno suddivisi in quattro diversi pacchetti provenienti dalle TEL62) ed effettuare il padding necessario per l'allineamento richiesto dalla memoria GPU per minimizzare i tempi di accesso ai dati da parte dei thread.

Bibliografia

- [1] G. Collazuol, V. Innocente, G. Lamanna, and F. Pantaleo, “Real-time use of gpus in na62 experiment,” in *Cellular Nanoscale Networks and Their Applications (CNNA), 2012 13th International Workshop on*, IEEE, 2012.
- [2] A. J. Buras, S. Uhlig, and F. Schwab, “Waiting for precise measurements of $K^+ \rightarrow \pi^+ \nu \bar{\nu}$ and $K_L \rightarrow \pi^0 \nu \bar{\nu}$,” *Rev. Mod. Phys.*, vol. 80, pp. 965–1007, Aug 2008.
- [3] F. Mescia and C. Smith, “Improved estimates of rare K decay matrix-elements from Kl3 decays,” *Phys. Rev. D*, vol. 76, 2007.
- [4] G. Anzivino, C. Biino, A. Bizzeti, F. Bucci, P. Cenci, R. Ciaranfi, G. Collazuol, V. Falaleev, S. Giudici, E. Iacopini, E. Imbergamo, M. Lenti, M. Pepe, R. Piandani, M. Piccini, M. Raggi, A. Sergi, and M. Veltri, “Construction and test of a {RICH} prototype for the {NA62} experiment,” *Nuclear Instruments and Methods in Physics Research Section A: Accelerators, Spectrometers, Detectors and Associated Equipment*, vol. 593, no. 3, pp. 314 – 318, 2008.
- [5] “Technical Design Document.” <http://na62.web.cern.ch/na62/Documents/TechnicalDesign.html>, december 2010.
- [6] B. Angelucci, G. Anzivino, C. Avanzini, C. Biino, A. Bizzeti, F. Bucci, A. Cassese, P. Cenci, R. Ciaranfi, G. Collazuol, V. Falaleev, S. Galeotti, S. Giudici, E. Iacopini, G. Lamanna, M. Lenti, G. Magazzù, E. M. Marinova, M. Pepe, R. Piandani, M. Piccini, G. Ruggiero, A. Sergi, M. Sozzi, and M. Veltri, “Pion-Muon separation with a RICH prototype for the NA62 experiment,” *Nuclear Instruments and Methods in Physics Research A*, vol. 621, pp. 205–211, Sept. 2010.
- [7] J. Crawford, “A Noniterative method for fitting circular arcs to measured points,” *Nucl.Instrum.Meth.*, vol. 211, pp. 223–225, 1983.
- [8] R. Ammendola *et al.*, “APEnet+: a 3D Torus network optimized for GPU-based HPC systems,” *Journal of Physics: Conference Series*, vol. 396, no. 4, p. 042059, 2012.
- [9] J. Corbet, A. Rubini, and G. Kroah-Hartman, *Linux Device Drivers, 3rd Edition*, ch. 17. O’Reilly Media, 2005.
- [10] H. Sutter, “The free lunch is over. A fundamental turn toward concurrency in software.” <http://www.gotw.ca/publications/concurrency-ddj.htm>, 2009.

- [11] G. Moore, "Cramming more components onto integrated circuits," *Proceedings of the IEEE*, vol. 86, pp. 82–85, Jan. 1998.
- [12] "CUDA C Programming Guide v.5.5." http://docs.nvidia.com/cuda/pdf/CUDA_C_Programming_Guide.pdf. Accessed: 2014-03-07.
- [13] "White Paper NVIDIA Fermi." http://www.nvidia.it/content/PDF/fermi_white_papers/NVIDIA_Fermi_Compute_Architecture_Whitepaper.pdf. Accessed: 2014-02-25.
- [14] "White Paper NVIDIA Kepler GK110." <http://www.nvidia.it/content/PDF/kepler/NVIDIA-Kepler-GK110-Architecture-Whitepaper.pdf>. Accessed: 2014-02-25.
- [15] "Intel® 64 and IA-32 Architectures Software Developer's Manual." <http://www.intel.com/content/www/us/en/processors/architectures-software-developer-manuals.html>. Accessed: 2014-02-26.
- [16] R. Ammendola *et al.*, "QUonG: A GPU-based HPC system dedicated to LQCD computing," in *Application Accelerators in High-Performance Computing (SAAHPC), 2011 Symposium on*, pp. 113–122, 2011.
- [17] R. Ammendola, M. Bernaschi, A. Biagioni, M. Bisson, M. Fatica, O. Frezza, F. Lo Cicero, A. Lonardo, E. Mastrostefano, P. S. Paolucci, D. Rossetti, F. Simula, L. Tosoratto, and P. Vicini, "GPU Peer-to-Peer Techniques Applied to a Cluster Interconnect," in *Parallel and Distributed Processing Symposium Workshops PhD Forum (IPDPSW), 2013 IEEE 27th International*, pp. 806–815, 2013.
- [18] G. Lamanna *et al.*, "GPU for real time processing in HEP trigger systems," in *Proceedings of 15th International Workshop on Advanced Computing and Analysis techniques in Physics (ACAT) 2013*, 2013.
- [19] M. Ageron *et al.*, "Technical Design Report for a Deep-Sea Research Infrastructure in the Mediterranean Sea Incorporating a Very Large Volume Neutrino Telescope," Tech. Rep. ISBN 978-90-6488-033-9, 2011.
- [20] R. Ammendola *et al.*, "Design and implementation of a modular, low latency, fault-aware, fpga-based network interface.," in *Track on Interconnect architectures for reconfigurable computing systems, held at Reconfigurable Computing and FPGAs (ReConFig), 2013 International Conference on*, 2013. to be published.
- [21] "NA62 online software and TDAQ interface." <http://na62.web.cern.ch/na62/Documents/NotesDoc/NoteNA62-11-02.pdf>. Accessed: 2014-03-07.

Ringraziamenti

Desidero ringraziare le persone care che mi hanno aiutato in ogni modo possibile: tutta la mia famiglia, insieme a Barbara, Letizia, Lucio, Tommaso. Senza di loro non sarei arrivato a questa pagina.

Un ringraziamento particolare va al mio relatore Alessandro Lonardo. Per avermi seguito costantemente, insegnato molto, supportato e sopportato sempre.

Grazie a tutti gli altri “apisti”: Andrea, Elena, Francesca, Francesco, Laura, Michele, Ottorino, Pier, Piero, Roberto. Mi hanno accolto e fatto sentire parte di un gruppo molto affiatato. Ringrazio Andrea Maiorano per la gentilezza e per avermi fatto divertire con la fisica degli strumenti musicali.

Voglio ringraziare gli amici con cui condivido le partite di GDR (e non solo!), cercando di salvare il Multiverso lanciando secchiate di D20. Mi sono stati sempre vicini.

Infine, ricordo con affetto Roberto, che non c'è più.