

HARD COMPUTING APPLICATO A QUERY SQL

DOTT. STEFANO GHIO

DISI – DIPARTIMENTO DI INFORMATICA E SCIENZE
DELL'INFORMAZIONE, GENOVA

RELAZIONE PER IMPLEMENTAZIONE DI LINGUAGGI
DOCENTE PROF. MASSIMO ANCONA

Indice

1 - Introduzione.....	3
2 – Accelerare operazioni SQL su GPU con CUDA.....	4
2.1 – Cos'è CUDA?.....	4
2.2 – Migliorare le prestazioni delle query SQLite.....	6
3 – Query compiler per FPGA.....	11
4 - Conclusioni.....	16
5 – Bibliografia.....	17

Abstract

La continua ricerca di metodi di programmazione performante in ambiti a task limitati spinge sempre più programmatori a scegliere soluzioni *hard computing* migrando dalle tradizionali CPU verso GPU ed FPGA. Vediamo ragioni e risultati di questa scelta analizzando il problema della rapidità di esecuzione di varie query SQL.

1 - Introduzione

Con il termine “hard computing” vado ad indicare tutte quelle soluzioni di programmazione che comprendono l'utilizzo di GPU (*Graphic Processing Unit*) o FPGA (*Field Programmable Gate Array*) a fianco od al posto delle tradizionali CPU.

“Hard” proprio per sottolineare che è richiesta al programmatore una certa conoscenza dell'hardware su cui lavora.

Il motivo che recentemente ha spinto sempre di più verso la ricerca di soluzioni di questo tipo è la scarsa performance dei processori odierni nel momento in cui si richiedono elaborazioni limitate ad un numero ristretto di compiti. Questo accade proprio a causa della loro natura *multi-purpose* che non consente di accentuare pesantemente il parallelismo di esecuzione o di evitare l'overhead di gestione della memoria tipica dei sistemi con architettura di Von Neumann[1].

In seguito analizzeremo due differenti implementazioni di un sistema per ottimizzare query SQL generiche:

- GPU Nvidia con codice CUDA per SQLite nell'ambito di query richiedenti l'analisi di valori floating point
- FPGA Xilinx per la realizzazione di un hardware query compiler

In entrambi gli esempi il sistema risultante realizza un coprocessore da affiancare ad una normale CPU (od un sistema più ampio), ma nel primo caso l'ottimizzazione riguarda uno specifico DBMS.

2 – Accelerare operazioni SQL su GPU con CUDA

2.1 – Cos'è CUDA?

Prima di poter analizzare il lavoro in sé è necessario dare una breve introduzione su cosa sia CUDA e come si possa utilizzare per programmare su GPU.

CUDA[2] è l'acronimo di *Compute Unified Device Architecture*, un'architettura di calcolo parallelo sviluppata da Nvidia. Il linguaggio che i programmatori devono impiegare per avvantaggiarsi di questa tecnologia è "C for CUDA", un'estensione Nvidia del linguaggio C standard; sono inoltre disponibili estensioni di terze parti per rendere utilizzabili anche linguaggi quali Python, Fortran, Java e MatLab.

Le estensioni apportate al classico C danno accesso ad istruzioni native ed alla memoria delle GPU compatibili con CUDA, rendendo così direttamente disponibile al programmatore l'architettura a più core (**NON** *multicore*) delle schede video attuali, in cui ogni singolo core è in grado di mantenere migliaia di thread attivi contemporaneamente.

L'approccio alla programmazione così realizzato viene definito comunemente GPGPU (*General Purpose computing on GPU*). Si potrebbe dire che finalmente qualcuno si sia accorto che le schede video odierne dispongono di una potenza di calcolo enorme – e spesso superiore a quella del sistema CPU+RAM su cui vengono installate - da poter impiegare per qualcosa di meglio che far muovere un paio di fiamme e detriti sullo schermo.

Rispetto ad un più anziano approccio GPGPU mediante API grafiche, CUDA presenta dei vantaggi:

- Letture sparse, il codice può accedere in lettura indirizzi arbitrari della memoria
- Memoria condivisa, una porzione di memoria (16KB) viene condivisa tra i thread. Può essere impiegata come cache controllata dall'utente, garantendo un'ampiezza di banda maggiore che rispetto all'utilizzo di texture lookup.
- Rapida trasmissione dati a e dalla GPU
- Supporto completo per operazioni tra interi e bit a bit, inclusi texture lookup

Sono presenti anche alcune limitazioni generali:

- L'estensione del C per CUDA non prevede ricorsione o puntatori a funzione ed un processo singolo deve girare sparso su più spazi di memoria disgiunti
- Non è supportato il rendering di texture, sebbene per alcuni questo non sia per nulla uno svantaggio
- La doppia precisione per numeri in virgola mobile è supportata solo su alcuni modelli di schede video più recenti e non è esattamente conforme allo standard IEEE 754[3]
- L'ampiezza di banda e la latenza del bus tra la CPU e la GPU può rivelarsi un collo di bottiglia
- Il modello di esecuzione dei thread è SIMD (*Single Instruction Multiple Data*) ed è consigliato di raggruppare i thread in gruppi di almeno 32 fino al centinaio per avere le migliori performance. Anche se il codice presentasse percorsi di esecuzione ramificati, questo non sarebbe un problema, ammesso che tutti i thread seguano lo stesso percorso

La figura seguente mostra una schematica di esecuzione di un processo su CUDA.

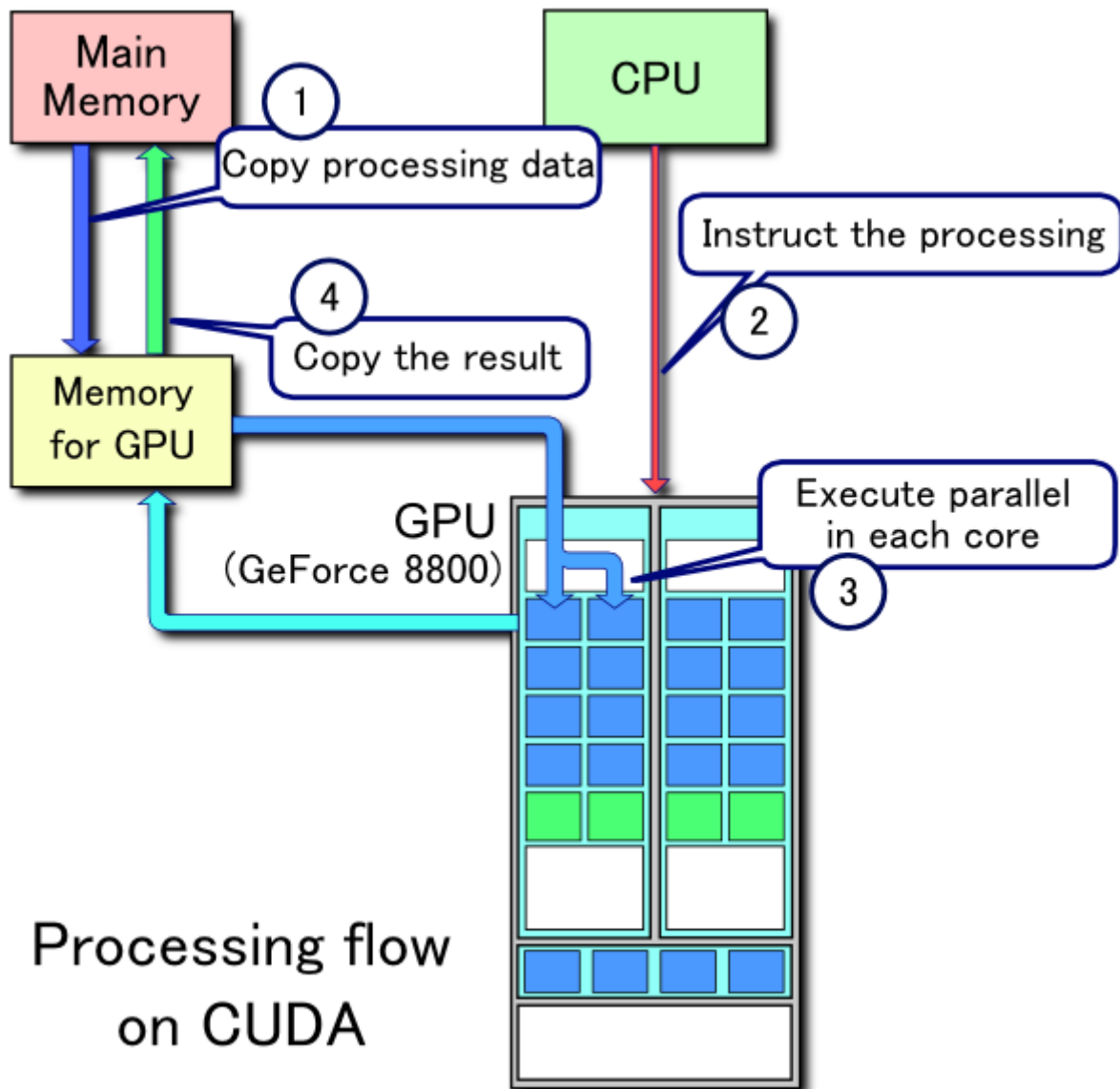


Figura 1: Schematica di esecuzione di un processo su CUDA.

E' possibile notare come la memoria di sistema e la memoria disponibile sulla scheda video restino separate per l'intero processo, comunicando solamente in fase di recupero dati e restituzione del risultato; inoltre la CPU, una volta "istruita" la GPU sul lavoro da svolgere può continuare la sua computazione senza restare bloccata in attesa.

2.2 – Migliorare le prestazioni delle query SQLite

Descrizione dell'esperimento

Per l'esperimento[4] è stato scelto di utilizzare il DBMS SQLite[5] e di focalizzare l'attenzione sulle query di tipo SELECT che dovessero venire richiamate più volte sullo stesso set di dati. La specifica del comando SELECT indica che la GPU verrà impiegata su dati di sola lettura, consentendo di massimizzare l'ampiezza di banda utilizzata e di memorizzare le tuple del database nella forma riga-colonna. Inoltre, i dati da analizzare risiedono già in memoria della scheda, evitando il rallentamento del caricamento degli stessi dalla memoria principale e, affinché l'esperimento mostri risultati positivi, è necessario che le query effettuate siano molte e fitte piuttosto che poche e sporadiche, in modo tale da giustificare quantomeno la messa in piedi di un sistema simile (1-5 milioni di tuple). Infine, le query prese in considerazione lavorano sia su dati di tipo intero che floating point a 32 o 64 bit in precisione singola o doppia.

Tutte queste limitazioni derivano principalmente dal fatto che un'esperienza più estesa richiederebbe molto tempo per essere affrontata correttamente e completamente, ma soprattutto dal concetto di impiego di queste soluzioni alternative che non è sostituzione globale delle CPU per ogni elaborazione – che non sarebbe possibile in modo efficiente al momento –, quanto piuttosto il rimpiazzo del processore per un numero limitato di operazioni critiche che risultano fortemente ottimizzate.

Sistema

Il sistema utilizzato per l'esperimento è di fascia piuttosto alta: Intel Xeon X5550[6] (2,66GHz quadcore con supporto a 8 hardware thread e banda massima 32GB/s) su macchina con kernel Linux 2.6.24 e 5GB di memoria. La GPU è una Nvidia TESLA C1060[7] (240 streaming multiprocessors, 4GB memoria e banda massima di 102GB/s) su cui gira una versione di SQLite ottimizzata in modo da mantenere ogni informazione utile in memoria per tutta la durata del processo.

Dati

Il data set impiegato è stato generato con il generatore di numeri casuali della libreria scientifica GNU. Una colonna per ogni tipo di dato ha una distribuzione uniforme nell'intervallo $[-99.0, 99.0]$, una colonna ha una distribuzione normale con una sigma di 5 ed una ha distribuzione normale con sigma 20. Sono state generate 5 milioni di tuple contenenti un id, tre interi e tre floating point.

Nel seguito indicheremo con *uniformT* e *normalTN* tali colonne dove *T* è *i* oppure *f* per indicare se si tratta di dato intero o floating point ed *N* è 5 o 20 a seconda della sigma.

Per i test sono state scritte 13 query SELECT differenti, 5 delle quali fanno operazioni su valori interi, 5 su valori floating point e 3 impiegano funzioni di aggregazione:

1. `SELECT id, uniformi, normali5 FROM test WHERE uniformi > 60 AND normali5 < 0`
2. `SELECT id, uniformf, normalf5 FROM test WHERE uniformf > 60 AND normalf5 < 0`
3. `SELECT id, uniformi, normali5 FROM test WHERE uniformi > -60 AND normali5 < 5`
4. `SELECT id, uniformf, normalf5 FROM test WHERE uniformf > -60 AND normalf5 < 5`
5. `SELECT id, normali5, normali20 FROM test WHERE (normali20 + 40) > (uniformi - 10)`
6. `SELECT id, normalf5, normalf20 FROM test WHERE (normalf20 + 40) > (uniformf - 10)`
7. `SELECT id, normali5, normali20 FROM test WHERE normali5 * normali20 BETWEEN -5 AND 5`
8. `SELECT id, normalf5, normalf20 FROM test WHERE normalf5 * normalf20 BETWEEN -5 AND 5`
9. `SELECT id, uniformi, normali5, normali20 FROM test WHERE NOT uniformi OR NOT normali5 OR NOT normali20`
10. `SELECT id, uniformf, normalf5, normalf20 FROM test WHERE NOT uniformf OR NOT normalf5 OR NOT normalf20`
11. `SELECT SUM(normalf20) FROM test`
12. `SELECT AVG(uniformi) FROM test WHERE uniformi > 0`
13. `SELECT MAX(normali5), MIN(normali5) FROM test`

Figura 2: Query impiegate per i test

SQLite

Analizzare il modo in cui SQLite processa le query aiuta a capire quali operazioni è preferibile parallelizzare per aumentare le prestazioni totali. In particolare, ogni query viene trasformata in un programma *opcode*, molto simile all'assembly. Andando ad ispezionare tale programma è possibile rimuovere operazioni non necessarie e scegliere quali inviare alla GPU per l'elaborazione.

Ad esempio, la query numero 1 di figura 2, verrà tradotta come:

0:	Trace	0	0	0
1:	Integer	60	1	0
2:	Integer	0	2	0
3:	Goto	0	17	0
4:	OpenRead	0	2	0
5:	Rewind	0	15	0
6:	Column	0	1	3
7:	Le	1	14	3
8:	Column	0	2	3
9:	Ge	2	14	3
10:	Column	0	0	5
11:	Column	0	1	6
12:	Column	0	2	7
13:	ResultRow	5	3	0
14:	Next	0	6	0
15:	Close	0	0	0
16:	Halt	0	0	0
17:	Transaction	0	0	0
18:	VerifyCookie	0	1	0
19:	TableLock	0	2	0
20:	Goto	0	4	0

Figura 3: Query 1 di figura 2, dopo il parsing di SQLite, a fianco del nome delle operazioni sono gli argomenti passati

Una macchina virtuale esegue questo opcode iterando sequenzialmente sull'intera tabella per generare le tuple risultato. Siccome tutti i dati sono salvati in memoria, alcune di queste operazioni non sono necessarie e non verranno implementate su CUDA. Inoltre, gli opcode manipolano l'esecuzione che non risulta sequenziale, l'operazione NEXT (riga 14 figura 3) ad esempio, avanza da una tupla alla seconda e salta successivamente al valore del secondo argomento. Analizzando questo opcode appare evidente che le operazioni dalla 6 alla 14 vengono eseguite per ogni riga della tabella e sono ottimizzabili assegnandone ognuna ad un thread CUDA separato e riunendo il tutto prima dell'operazione NEXT.

Riscrivendo la macchina virtuale SQLite con CUDA è possibile migliorare le prestazioni parallelizzando l'esecuzione di alcune istruzioni.

Risultati

Considerando i dati come risidenti in memoria, rispetto alla sola elaborazione con CPU e codice SQLite ottimizzato per la macchina, ma senza coprocessore CUDA, si è ottenuto un miglioramento in rapidità di esecuzione nell'ordine delle 50 volte, che diventano 36 se si aggiunge il trasferimento dei dati dalla memoria primaria a quella secondaria.

Queries	Speedup	Speedup w/ Transfer	CPU time (s)	GPU time (s)	Transfer Time (s)	Rows Returned
Int	42.11	28.89	2.3843	0.0566	0.0259148	1950104.4
Float	59.16	43.68	3.5273	0.0596	0.0211238	1951015.8
Aggregation	36.22	36.19	1.0569	0.0292	0.0000237	1
All	50.85	36.20	2.2737	0.0447	0.0180920	1500431.08

Figura 4: Query elaborate su CPU e GPU con relativi tempi di esecuzione

La figura sopra mostra i risultati ottenuti per le varie query prese in analisi nel caso in cui i dati stessero interamente in memoria (prima colonna) o vi dovessero essere trasferiti (seconda colonna).

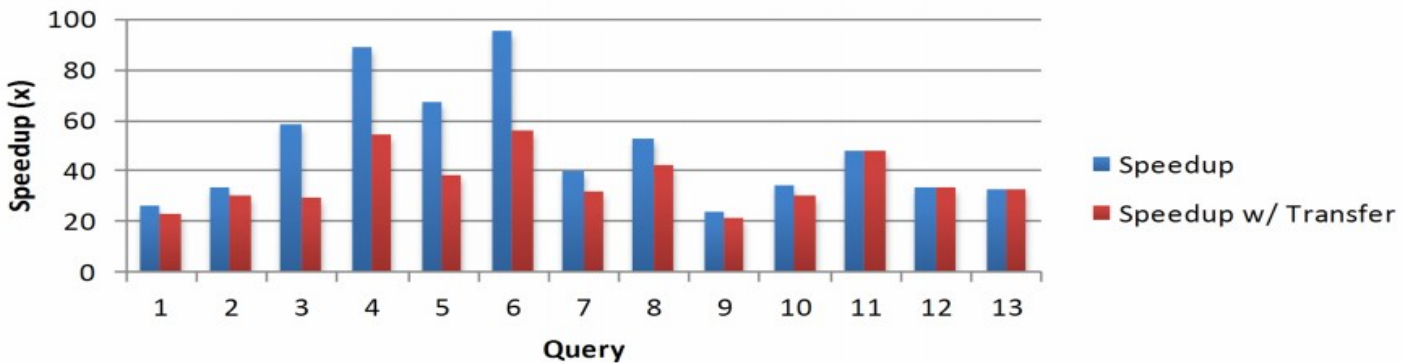


Figura 5: Panoramica dei risultati. Query dispari lavorano su numeri interi, query pari lavorano su floating point e le ultime tre (11-13) sono le operazioni di aggregazione

Nei casi in cui fosse necessario trasferire dati da una memoria all'altra o non sia possibile parallelizzare le operazioni, come nel caso della restituzione dei risultati, si nota come i due tipi di elaborazione (con e senza caricamento dati dalla memoria) non si discostino poi tanto.

Limitazioni della tecnologia CUDA

Analizzando i risultati si possono notare come alcune delle limitazioni illustrate in precedenza pesino negativamente sulle prestazioni.

Innanzitutto, jump indiretti non sono supportati. Siccome gli opcode SQLite implementati nella macchina virtuale sono realizzati in un blocco switch e tale blocco è utilizzato da ogni thread per ogni opcode, paragonare l'argomento dello switch al valore dell'opcode crea un overhead non trascurabile. Se fossero possibili i jump indiretti, siccome ogni opcode è unico, sarebbe possibile effettuare immediatamente in salto senza dover analizzare il blocco switch ad ogni richiesta.

Un'altra limitazione deriva dal fatto che gli array acceduti dinamicamente vengono memorizzati nella memoria locale piuttosto che nei registri da CUDA. Con memoria locale si intende la memoria come viene vista da un singolo thread che risiede in memoria globale. Avendo la stessa latenza

della memoria globale, questa risulta essere più lenta di un registro per un fattore 100 circa.

Infine, operazioni atomiche in CUDA sono definite solo per valori interi, rendendo quindi meno efficiente l'elaborazione per altri tipi di dato quando è richiesta la comunicazione tra i vari thread creati.

La limitazione maggiore si rivela tuttavia essere la memoria. Sebbene 4GB siano molti e consentano di immagazzinare milioni di tuple, database molto estesi possono arrivare ad avere Terabyte di dati che non trovano posto sulla scheda.

Futuro

Con la prossima generazione di schede video "Fermi" alcune di queste limitazioni dovrebbero scomparire in quanto si sta cercando di inserire il supporto CUDA all'interno del C++, cosa che consentirebbe di risolvere alcuni problemi critici come l'allocazione dinamica della memoria.

3 – Query compiler per FPGA

Come nel caso precedente, andiamo a definire cosa si cerca di ottenere con l'esperimento[8] prima di procedere con la sua discussione.

Abbiamo appena visto cosa succede se scegliamo di usare un coprocessore GPU per velocizzare alcune query. Osserviamo cosa succede se inseriamo al suo posto una FPGA programmata in modo tale da tradurre in hardware le query SQL che vogliamo effettuare ottenendo così un query compiler, ossia un oggetto che, presa una query SQL, la “instrada” verso il circuito sintetizzato appropriato per velocizzarne l'esecuzione.

Descrizione dell'esperimento

Si impiega Glacier[9], una component library e compilatore per processare data stream su FPGA.

L'applicazione che sarà creata mira a supportare la ricezione e l'elaborazione di un ampio numero (3 milioni circa) di pacchetti di dati, trasmessi via UDP sotto forma di stream continuo in ambito di gestione bancaria delle informazioni. La motivazione principale che spinge a scegliere questa soluzione è che le normali tecniche note ad oggi non consentono un'elaborazione così rapida da poter gestire una mole di dati simile.

Sistema

Viene impiegata una FPGA Xilinx XC5VLX110T[10]

Dati

L'idea è quella di gestire molti dati di piccole dimensioni, la cui tabella può essere descritta come segue:

```
CREATE INPUT STREAM Trades (  
  Seqnr int,          -- sequence number  
  Symbol string(4),  -- valor symbol  
  Price int,         -- stock price  
  Volume int)        -- trade volume
```

Figura 6: Tabella formato dati in input

In questo caso ci si limita ad uno stream di dati unico e non si consente l'uso di comandi di join.

Vengono analizzate le seguenti query:

```
SELECT Price, Volume
  FROM Trades
 WHERE Symbol = "UBSN"
 INTO UBSTrades (Q1)

SELECT Price, Volume
  FROM Trades
 WHERE Symbol = "UBSN" AND Volume > 100000
 INTO LargeUBSTrades (Q2)

SELECT count () AS Number
  FROM Trades [SIZE 600 ADVANCE 60 TIME]
 WHERE Symbol = "UBSN"
 INTO NumUBSTrades (Q3)

SELECT wsum (Price, [.5, .25, .125, .125]) AS Wprice
  FROM (SELECT * FROM Trades
        WHERE Symbol = "UBSN")
        [SIZE 4 ADVANCE 1 TUPLES]
 INTO WeightedUBSTrades (Q4)

SELECT Symbol, avg (Price) AS AvgPrice
  FROM Trades [SIZE 600 ADVANCE 60 TIME]
 GROUP BY Symbol
 INTO PriceAverages (Q5)
```

Figura 7: Query analizzate per l'esperimento

Siccome si parla di gestione di dati finanziari, alcune query comportano una semplice estrazione dati (*Q1,Q2*) mentre altre richiedono un'analisi dei dati su un periodo di tempo prolungato (*Q3,Q4,Q5*).

Saranno inoltre supportate le seguenti operazioni:

$\pi_{a_1, \dots, a_n}(q)$	projection
$\sigma_a(q)$	select tuples where field a contains true
$\odot_{a:(b_1, b_2)}(q)$	arithmetic/Boolean operation $a = b_1 \star b_2$
$q_1 \cup q_2$	union
$agg_{b:a}(q)$	aggregate agg using input field a , $agg \in \{\text{avg}, \text{count}, \text{max}, \text{min}, \text{sum}\}$
$q_1 \text{ grp}_{x c} q_2(x)$	group output of q_1 by field c , then invoke q_2 with x substituted by the group
$q_1 \boxplus_{x k,l}^t q_2(x)$	sliding window with size k , advance by l ; apply q_2 with x substituted on each wind.;
$q_1 \boxdot q_2$	concatenation; position-based field join

Figura 8: Operazioni aritmetiche supportate

Ogni operazione SQL può essere riscritta in forma algebrica, le query proposte in figura 7, secondo lo schema di figura 8, diventano perciò:

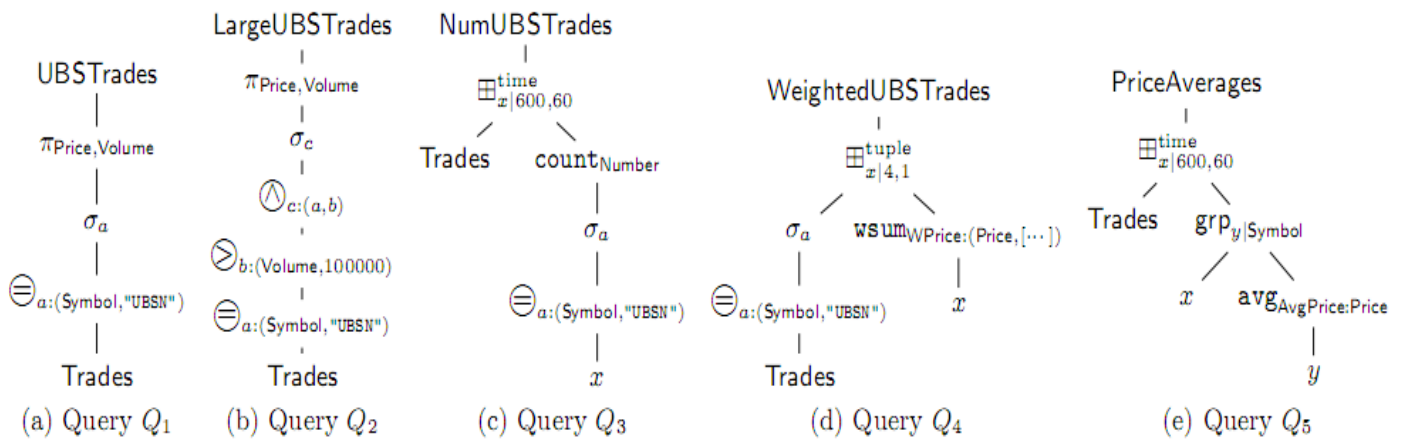


Figura 9: Query di figura 7 in forma algebrica seconda la segnatura di figura 8

Una volta rappresentate con questa schematica, è possibile iniziare a visualizzare le query sotto forma di stream dati attraverso un percorso preciso, che poi sarà sintetizzato come circuito sulla FPGA.

La query $Q1$, diventerà quindi:

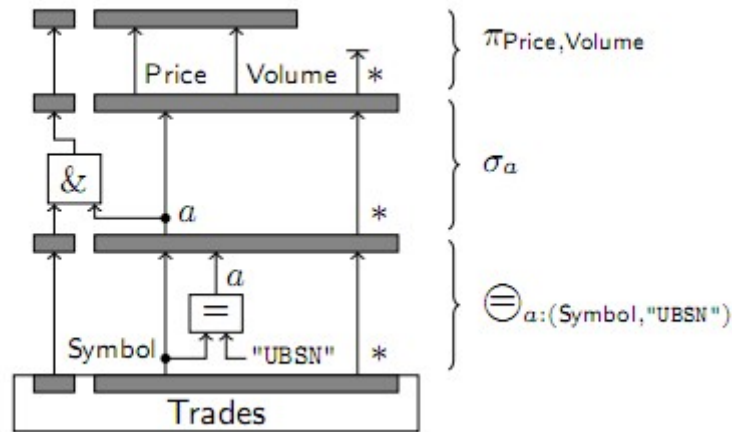


Figura 10: Query Q1 rappresentata sotto forma di circuito

Mentre la finestra entro la quale si vogliono analizzare i dati per le query $Q3, Q4$ e $Q5$, potrà essere rappresentata come:

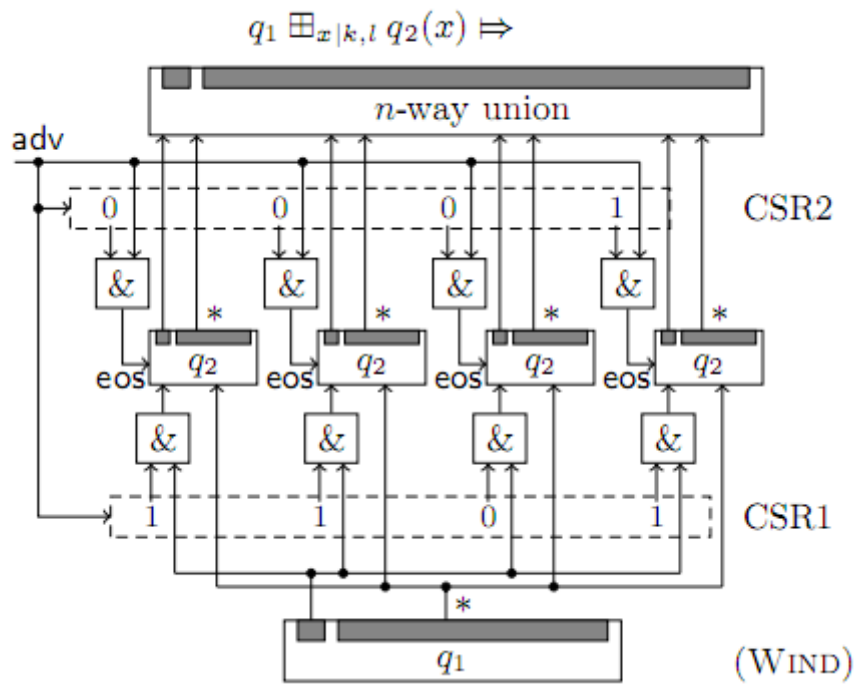


Figura 11: Finestra di analisi dati per le query Q3, Q4 e Q5

Da cui è possibile visualizzare il parallelismo di esecuzione che sarà realizzato come sintesi di circuito virtuale sulla scheda.

Glacier

Un ruolo fondamentale in questo esperimento è svolto dalla libreria Glacier, che consente di realizzare una sintesi hardware per le query analizzate in precedenza, integrando anche un certo grado di parallelismo ulteriormente configurabile successivamente dall'utente.

Si osserva inoltre che si ottengono risultati migliori quando le frequenze di clock non sono troppo elevate, questo per consentire la sincronizzazione di tutti i flip-flop coinvolti nelle operazioni e di memorizzare il corretto risultato prima di continuare con la computazione delle altre operazioni descritte nella query.

E' inoltre possibile forzare al massimo il parallelismo rimuovendo alcuni registri intermedi ed inserire più operazioni differenti all'interno dello stesso ciclo di clock, sebbene questa soluzione si scontri spesso con il problema dell'attesa del dato corretto descritta poco sopra.

Risultati

Con il sistema appena descritto si è riusciti ad arrivare a processare 100 milioni di tuple in input al secondo con una frequenza di clock di appena 100MHz.

Confrontando i risultati ottenuti con questa implementazione hardware con una software su CPU classica su una macchina con kernel Linux 2.6 si nota che, anche se il numero di pacchetti in input è relativamente basso, l'implementazione software non riesce a gestire l'intero stream in ingresso.

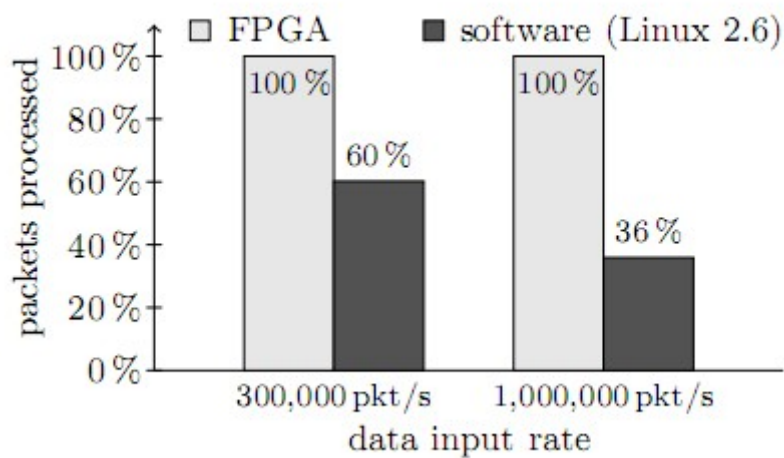


Figura 12: Risultati per percentuale di pacchetti processati rispetto al numero di input

4 - Conclusioni

Dai precedenti esperimenti si nota come, in ambito specifico con compiti precisi e ridotti, scegliere una soluzione alternativa ad un processore tradizionale possa garantire un enorme miglioramento delle prestazioni proprio grazie alla differente natura che caratterizza le CPU tradizionali (non volte al parallelismo quanto piuttosto alla multifunzionalità), le moderne GPU (con grande potenza di calcolo parallelo, memoria e ampiezza di banda) e le FPGA (grande flessibilità e rapidità di esecuzione anche a basse frequenze di lavoro).

5 – Bibliografia

- [1] Schema di architettura Von Neumann:
http://en.wikipedia.org/wiki/Von_Neumann_architecture
- [2] Pagina principale del progetto CUDA: http://www.nvidia.com/object/cuda_home_new.html
- [3] Bozza (gratuita) dello standard IEEE 754 per il calcolo aritmetico di numeri in virgola mobile:
<http://www.validlab.com/754R/nonabelian.com/754/comments/Q754.129.pdf>
- [4] Peter Bakkum – Kevin Skadron, Accelerating SQL database operations on a GPU with CUDA:
http://www.cs.virginia.edu/~skadron/Papers/bakkum_sqlite_gpgpu10.pdf
- [5] Pagina principale del progetto SQLite: <http://www.sqlite.org/>
- [6] Panoramica dell'Intel Xeon X5550: <http://ark.intel.com/Product.aspx?id=37106>
- [7] Panoramica della Nvidia TESLA C1060:
http://www.nvidia.com/object/product_tesla_c1060_us.html
- [8] Rene Mueller – Jens Teubner – Gustavo Alonso, Streams on wires – A query compiler for FPGAs:
<http://www.vldb.org/pvldb/2/vldb09-622.pdf>
- [9] Rene Mueller – Jens Teubner – Gustavo Alonso, Glacier: A Query-to-hardware compiler:
<http://people.inf.ethz.ch/jteubner/publications/glacier-demo/glacier-demo.pdf>
- [10] Specifiche della famiglia Xilinx XC5VLX110T:
http://www.xilinx.com/support/documentation/data_sheets/ds100.pdf