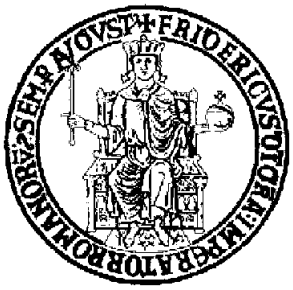
DOTTORATO DI RICERCA

in

SCIENZE COMPUTAZIONALI E INFORMATICHE

XXIII ciclo



Consorzio tra Università di Catania, Università di Napoli Federico II,
Seconda Università di Napoli, Università di Palermo, Università di Salerno

SEDE AMMINISTRATIVA: UNIVERSITÀ DI NAPOLI FEDERICO II

Valutazione delle prestazioni di algoritmi paralleli fino ad ambienti GPU

Valeria Mele

TESI DI DOTTORATO DI RICERCA

A mio zio Rosario
A Monica, a Roberta, a Ferruccio, a zia Mariella
Perché forse mai prima di questa tesi è stato scritto un lavoro scientifico
pensando così tanto a qualcuno

Ringraziamenti

Napoli, 30 Novembre 2011

Negli anni in cui ho studiato e lavorato per raggiungere questo momento, sono successe tante cose, dal punto di vista della mia vita personale, ma anche della mia formazione. Tante sono state le occasioni e le esperienze, che mi hanno piano piano resa un po' più adulta e un po' più ricercatrice.

Quello che ho imparato e adesso sto cercando di mettere a frutto con i primi lavori scientifici, tra cui questa tesi, non sarebbe stato possibile senza il Prof. Murli che è senza dubbio una guida scientifica irrinunciabile e, come gli dico spesso, a volte sembra proprio 'crederci' più di me. Meno male che c'è lui, che ha sempre la soluzione e soprattuto sa molto bene come riportarmi sulla strada quando ce n'è bisogno.
Un altro contributo fondamentale alla mia formazione è venuto e continua a venire dalla Prof. D'Amore, che in questi anni ha dedicato molto del suo tempo a spiegarmi, correggermi, indirizzarmi, e anche a perdonare i miei ritardi e le mie facce mute e perplesse di fronte ad argomenti ancora non metabolizzati, sempre con l'elegante franchezza che la contraddistingue: come non dedicarle un ringraziamento particolare.
In questa tesi la loro partecipazine è stata fondamentale.

Ma in realtà è tutto il gruppo che devo ringraziare: il Prof. Laccetti, che accompagno anche nella didattica e che mi è venuto incontro con grande disponibilità nei momenti più stressanti, il Prof. Lapegna che mi ha fatto da tutor, Rosanna Campagna e Rossella Arcucci delle quali proprio non saprei

fare a meno, Diego Romano che mi soccorre sempre anche all'ultimo minuto, Livia Marcellino e la principessa Alice, Maria Gelli che è un punto di riferimento, e tutti gli altri. Loro sono la mia squadra. E la ricerca senza squadra forse è pure possibile, ma sicuramente non è ugualmente interessante e divertente.

Un ringraziamento doveroso va al Prof. Paul Messina, per la gentilezza e la disponibilità con cui mi ha ascoltato e mi ha risposto, nelle occasioni in cui, ospite del Prof. Murli, ho potuto incontrarlo: è stato un importante incoraggiamento. Spero di saper mettere a frutto le sue osservazioni e le sue indicazioni nella mia ricerca.

Ringrazio poi con tutto il cuore i miei genitori che qualsiasi cosa sia accaduta sono stati la mia unica certezza, e sapere che continueranno sempre ad essere una certezza è ciò che mi permette di non farmi sottomettere definitivamente da timori ed insicurezze, e di continuare a cercare ciò che voglio veramente, in un momento in cui questo si può considerare un lusso di pochi.

Ringrazio gli Amici che ci sono stati ogni volta che ne ho avuto bisogno, e che ho trascurato spesso per dedicarmi allo studio, soprattutto ultimamente, senza che se la siano mai presa.

Ma soprattutto ringrazio chi era lontano e nonostante questo è riuscito a starmi vicino, a soccorrere il mio morale ferito e a raccogliere i miei sorrisi e le mie piccole soddisfazioni, con la pazienza e la comprensione che sanno avere solo le persone speciali.

Questa tesi è un po' di tutti loro.
Il mondo è un posto affascinante e difficile e il tempo non è dei migliori.
Bisogna rimboccarsi le maniche, ma...
Io sono molto fortunata.

Valeria

Indice

Prefazione

Attraverso l'esperienza acquisita nello sviluppo e l'analisi di algoritmi per diversi sistemi ed architetture paralleli [39, 40, 41], l'attenzione del mio studio si è focalizzata sulle prestazioni degli algoritmi, le relazioni esistenti tra i diversi fattori che intervengono nella loro progettazione e i limiti imposti dall'ambiente di calcolo.
Per quanto riguarda le architetture classificabili secondo Flynn, negli anni si è consolidato ormai un sistema di parametri che ha mostrato la sua efficacia nella previsione e nella valutazione della prestazione ottenibile da un algoritmo, sequenziale e parallelo.

Attualmente, si può dire che non esista un calcolatore che si possa definire 'non parallelo', anzi, decisamente negli ultimi anni l'aumento delle prestazioni è stato sempre dovuto al massiccio incremento del parallelismo e la stessa legge di Moore è stata generalmente rivista per adattarne il concetto di base a questo fenomeno: non si tiene più conto del numero di transistor presenti in un processore, ma del grado di parallelismo delle macchine.
Tuttavia, i sistemi più moderni sono tutti di tipo ibrido (secondo Flynn) e più si complica la loro struttura ed aumenta la loro eterogeneità, più diventa difficile utilizzare i classici parametri di valutazione. In quest'ambito, infatti, spesso lo studio si limita solo all'osservazione dei tempi d'esecuzione.

In questa tesi, si ripercorrono le caratteristiche delle architetture parallele fino a prendere in considerazione i calcolatori muniti di Graphic Processing Unit (GPU) e si propone un modello per la valutazione delle prestazioni a partire dai risultati classici, calandoli nella realtà della macchina in questione

ed individuando, attraverso lo studio dei dettagli tecnici e gli esperimenti, parametri che permettono di capire i limiti e interpretare la variazioni nei guadagni ottenuti.

'Io sono un filosofo, confondo tutti!
Uccelli, bestie, uomini e... No! Le donne no.'

Lord Byron, 'Don Juan, Canto sesto', 1818
(citato da C. Babbage nella sua autobiografia)

'...dimenticare questo mondo e tutti i suoi guai
e, se è possibile,
tutti i suoi numerosi Ciarlatani,
tutto, subito,
tranne la mia Incantatrice di Numeri.'

Charles Babbage ad Ada Lovelace, 1843

Capitolo 1

Introduzione

Questo lavoro di tesi si inserisce nell'ambito del Calcolo Parallelo. Il **Calcolo Parallelo** [43], è una delle principali metodologie per ottenere alte prestazioni. Chiaramente il passaggio da un algoritmo sequenziale ad uno parallelo non è indolore: comporta uno sforzo in termini di risorse e di lavoro non sempre trascurabile, che però è generalmente ben compensato dal vantaggio che si ottiene nelle performance. In ogni caso, il raggiungimento di prestazioni al limite di ciò che è permesso dall'attuale tecnologia comporta sempre un'approfondita analisi ed uno studio delle possibilità e delle modalità, da cui non si può prescindere.

Si tratta dunque di una disciplina sviluppatasi al fine di sfruttare al massimo le architetture capaci di elaborazione concorrente che si sono evolute a partire già dai primi anni della storia dell'architettura dei calcolatori (**Capitolo 2**).
Quando l'avvento dei circuiti integrati diede la spinta definitiva all'industria dell'informatica, mentre si sperimentavano nuovi progetti architetturali e Gordon Moore fondava l'Intel e rifletteva sulla densità dei transistor sul materiale semiconduttore, Michael J. Flynn, progettista all'IBM, cominciando ad interessarsi della valutazione delle performance delle diverse architetture, chiariva il concetto di *concorrenza* e in base a quello inquadrava le diverse possibilità architetturali all'interno di una *tassonomia* passata alla storia

con il suo nome, e che ancora oggi possiamo usare come riferimento per la comprensione funzionale dei calcolatori paralleli (**Capitolo 3**), perché nei concetti che riassume non ha mai smesso di essere attuale.

Successivamente la possibilità di confrontare, spiegare e prevedere le prestazioni di un sistema di calcolo è diventata sempre più importante e gli sforzi in questo senso si sono moltiplicati, dando vita a molti modelli che, oltre a dare agli sviluppatori la consapevolezza delle possibilità offerte, hanno creato la giusta base di comprensione per la nascita di nuove idee e nuovi progetti. Nel **Capitolo 4** riprenderemo i concetti fondamentali relativi alla valutazione delle prestazioni di algoritmi paralleli con particolare attenzione alla funzione *overhead*.
Il classico approccio alla valutazione delle performance però diventa sempre più difficile a mano a mano che le architetture si complicano. Le macchine attuali mescolano principi diversi e tentano di trarre beneficio da ogni sviluppo passato, e a volte è impossibile ricondurle ad una categoria precisa di cui esista un modello consolidato. I progetti più recenti e certamente quelli futuri sono ormai basati sull'eterogeneità: accostamento e stratificazione di tipi di parallelismo diversi. Su questa linea si sta infatti lavorando allo sviluppo dei prossimi sistemi **exascale**, dei sistemi operativi ad essi destinati e del software che potrà sfruttarli.

Un esempio di architettura parallela che mette insieme più livelli di parallelismo per ottenere l'accelerazione di un algoritmo, sono le GPU (Graphics Processing Unit), dispositivi da affiancare al microprocessore ormai affermate non solo nell'ambito della resa grafica ma anche di applicazioni cosiddette *general purpose*.
Nel **Capitolo 5** si definirà la struttura dell'architettura generica di una *GPU* (Grafics Processing Unit) [42], inquadrandola nell'ambito della *tassonomia di Flynn*. Sarà poi sviluppata un'analisi che porterà ad individuare alcune caratteristiche rilevanti di tale tipo di architettura, che ci permetteranno di descrivere un possibile criterio di ottimalità per prevedere l'*operating point* di un algoritmo, ovvero il numero di thread con cui ottenere le prestazioni

migliori.
Tale criterio verrà poi validato nel **Capitolo 6**, attraverso gli esperimenti riportati nella recente letteratura sull'argomento.

Capitolo 2

Storia delle architetture

Il parallelismo è un concetto che ha accompagnato lo sviluppo delle architetture dei calcolatori lungo tutta la storia. Appena si è posto il problema dell'aumento delle possibilità e quindi delle prestazioni dei sistemi di calcolo è apparsa chiara l'opportunità di inserire concorreza tra le diverse componenti (vedi Capitolo 3). La performance dei calcolatori nel tempo è stata incrementata, infatti, da una parte migliorando la tecnologia fisica e quindi il tempo di un'operazione, dall'altra aumentando il numero di operazioni eseguibili contemporaneamente.

2.1 Background storico

Volendo ripercorrere le idee che hanno portato fino ad immaginare e poi progettare macchine da calcolo dovremmo andare molto molto dietro nel tempo. Tuttavia quello che ci interessa davvero è la storia del **calcolatore elettronico**, che ha rivoluzionato la tecnologia del XX secolo, caratterizzando un'epoca.
In ogni caso, forse non ci si sarebbe arrivati senza lo studio e i tentativi di alcune personalità notevoli, che precorrendo i tempi avevano cominciato a progettare e costruire macchine calcolatrici già nei secoli precedenti. Tra questi, scegliamo di citare Blaise Pascal (1623-1662, francese), che nella Francia della metà del XVII secolo [9, 7] progettò, costruì e vendette la prima macchi-

na calcolatrice funzionante, interamente meccanica (azionata a manovella), in grado di compiere egregiamente somme e sottrazioni. La sua macchina fu ripresa ed ampliata da Leibniz (1646-1716, tedesco) in una che potesse eseguire anche moltiplicazioni e divisioni [7]. Liebniz espresse chiaramente l'opportunità di far compiere ad una macchina calcoli che comportavano solo una noiosa perdita di tempo per gli studiosi[1], e questo tema fu mantenuto e portato avanti anche nell'opera di un altra grande personalità quale l'inglese Charles Babbage (1792-1871) che progettò dettagliatamente la *Macchina Analitica* (Analytical Engine) [8, 10, 7] e a cui si deve probabilmente anche il primo accordo tra uno scienziato ed un governo: egli riuscì ad ottenere supporto economico per il proprio progetto in Inghilterra, ma per il susseguirsi di difficoltà impreviste e per l'inadeguatezza dei tempi, la macchina non fu mai realizzata. Il suo progetto però prevedeva un'unità aritmetica, la capacità di realizzare salti condizionati e cicli, e una memoria integrata. Sarebbe inoltre stata programmabile: Ada Lovelace [10] scrisse per questa macchina un 'programma' in grado di calcolare i numeri di Bernoulli. L'insieme dei principi alla base della macchina di Babbage è a tutti gli effetti vicinissimo a quello che ha ispirato successivamente il progetto dei veri e propri calcolatori funzionanti e non prescindeva da concetti di **parallelismo**, prevedendo che le diverse componenti potessero lavorare in maniera concorrente per velocizzare l'operazione globale.

La tecnologia divenne matura per l'interesse verso i calcolatori, sia in Europa che negli Stati Uniti, solo dopo l'inizio del XX secolo.

Già nei primi anni '40 l'americano John Atanasoff[2] (1903-1995) scriveva del progetto di una macchina elettronica ad aritmetica binaria per la risoluzione di un sistema di equazioni lineari, ma le sue idee non ebbero la fortuna meritata a causa dell'inadeguatezza della tecnologia del suo tempo.

Il vero stimolo per lo sviluppo che portò ai calcolatori elettronici venne dalla

[1] "...is unworthy of escellent men to lose hours like slaver in the labor of calculation which could safely be relegated to anyone else if machines were used", riportato in [7].

[2] Atanasoff fu autore tra l'altro di una delle prime comunicazioni in cui si evidenzia l'importanza dei grandi sistemi d'equazioni lineari per diverse applicazioni, comunicazione che Goldstine e Von Neumann consideravano il primo articolo di *analisi numerica* mai scritto.

Seconda Guerra Mondiale: primi tra tutti, gli inglesi si impegnarono nella costruzione del segretissimo computer COLOSSUS (1943) per interpretare i messaggi tedeschi cifrati da ENIGMA, con anche il contributo (1912-1954) del grande matematico Alan Turing.
Il primo vero computer di uso generale realizzato e funzionante, si deve però ad Howard Aiken e al suo gruppo di lavoro della Harvard University: il Mark I (progetto iniziato nel 1939 e terminato nel 1943). Era un dispositivo digitale elettromeccanico, basato su relè (interruttori azionati da un elettromagnete), composto da 72 parole di 23 cifre decimali, con un tempo di istruzione di 6 secondi e un nastro di carta per l'input e l'output.
Questo calcolatore rimase a disposizione della Marina Militare degli Stati Uniti per tutta la seconda guerra mondiale ed ebbe un alto impatto mediatico, permettendo la nascita e lo sviluppo del laboratorio universitario di Harvard.
Il Mark I e i suoi successori ebbero un breve successo, presto messi a confronto con i *calcolatori elettronici* che sfruttavano valvole termoioniche.

2.2 Generazione I - Calcolatori a valvole

Durante la guerra, negli Stati Uniti l'esercito stabilì di finanziare la costruzione dell'ENIAC (*Electronic Numerical Integrator And Computer*) da parte di John Mauchley, che aveva studiato anche il lavoro di Atanasoff.
L'ENIAC era costituito da 18000 valvole termoioniche – componenti elettronici in grado di funzionare come interruttori elettronici comandati da segnali elettrici, cioè triodi) e 1500 relè – pesava 30 tonnellate e consumava 140 KW di energia. Era dotata di 20 registri, ciascuno in grado di memorizzare un numero decimale a 10 cifre. Veniva programmato regolando 6000 interruttori multi-posizione e connettendo una moltitudine di prese con una vera e propria foresta di cavi (figura 2.1). Non fu usato per scopi bellici perché terminato troppo tardi, ma, presentato a una scuola estiva, scatenò l'interesse generale verso la costruzione di grandi computer digitali.
Nel 1944 anni John von Neumann (1903-1957), geniale scienziato paragonato a Leonardo da Vinci per le sue abilità, che era già allora uno dei più

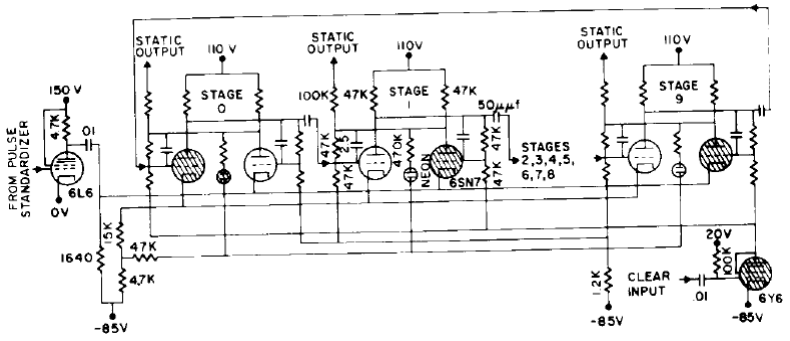


Figura 2.1: Schema del contatore decimale ad anello dell'ENIAC.

importanti matematici al mondo, fu invitato a vedere l'ENIAC[3] e collaborò poi all'evoluzione del progetto in EDVAC (*Electronic Discrete Variable Automatic COmputer*) [1] all'università della Pennsylvania. Quando il progetto dell'EDVAC si fermò, Von Neumann ne sviluppò, con il collaboratore Goldstine (1913-2004), la propria versione a Princeton: la macchina IAS (1952), che ha costituito la base per tutti i computer digitali a venire. Le idee innovative introdotte erano la rappresentazione dei programmi, come i dati, in forma numerica all'interno della memoria del computer, e la sostituzione dell'aritmetica decimale con una binaria, riprendendo il funzionamento della macchina di Atanasoff.

Il progetto che Von Neumann descrisse è rimasto noto come **macchina di Von Neumann** [1], schematizzabile secondo la Fig.2.3 .

In quel momento le macchine calcolatrici concrete diventavano facilmente programmabili, e ne veniva aperto l'utilizzo ad infinite possibilità. In pratica nasceva il concetto di calcolatore così come ancora oggi viene inteso.

Il primo calcolatore commerciale a valvole termoioniche fu l'UNIVAC, prodotto dalla compagnia fondata da Mauchley ed utilizzato nel 1952 per i primi exit-poll delle elezioni presidenziali statunitensi: anticipò correttamente la vincita di Eisenhower. Tra le ultime macchine a valvole ricordiamo invece

[3]Goldstine [7] racconta chee Eckert, scienziato partecipante al progetto dell'ENIAC, sapendo dell'imminente visita di Von Neumann, preannunciò che avrebbe capito se costui fosse un genio come si diceva o meno dalla prima domanda che avrebbe fatto: la prima domanda di Von Neumann fu riguardo la struttura logica della macchina, e questo bastò ad Eckert per stabilire che si trattava davvero di un genio.

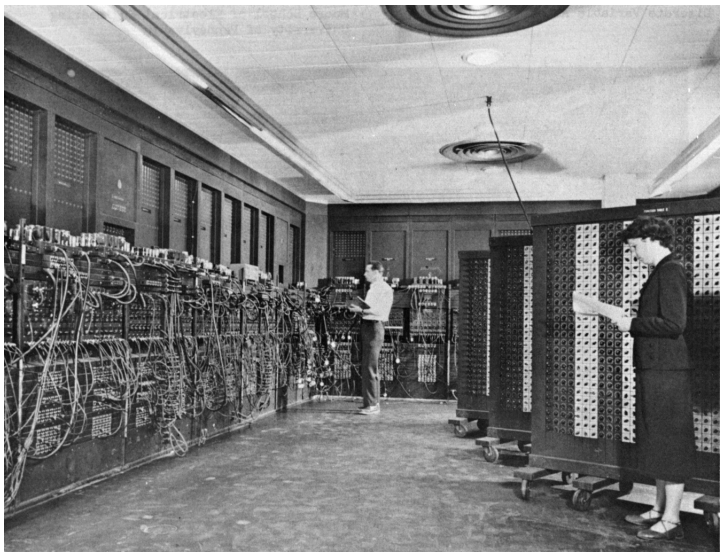


Figura 2.2: Eniac al lavoro.

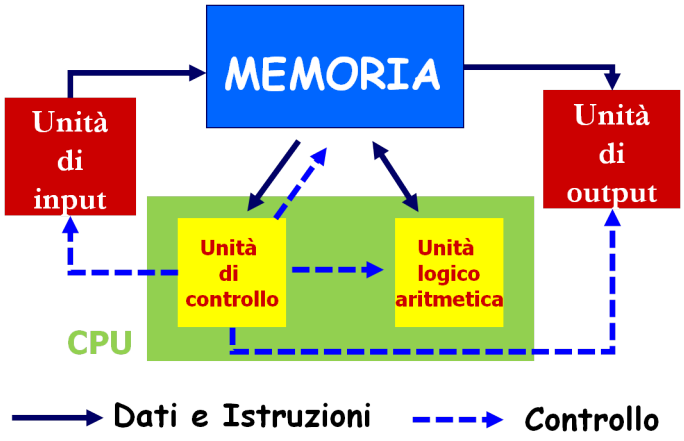


Figura 2.3: Schema generale della Macchina di Von Neumann.

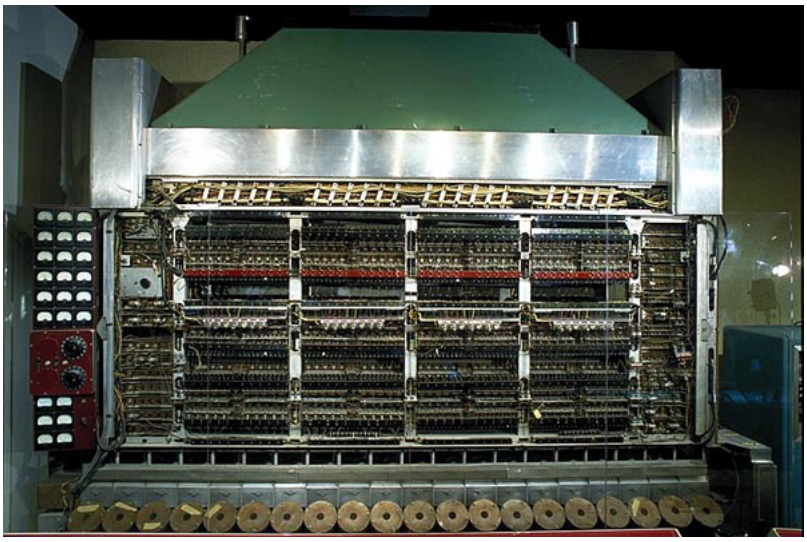


Figura 2.4: IAS, Institute of Advanced Studies of Princeton, la prima macchina a programma memorizzato.

quelle che seguivano il modello 709 dell'IBM (1958), società nata per il commercio di schede perforate che da pochi anni si interessava ai computer veri e propri.

2.3 Generazione II - Calcolatori a transistor

Nel 1948 i Bell Labs inventarono il transistor, un componente elettronico a stato solido realizzato con un cristallo di germanio che poteva essere utilizzato come 'interruttore elettronico' comandato da segnali elettrici e quindi poteva sostituire la valvola termoionica: questa invenzione valse il premio Nobel per la fisica nel 1956 ai ricercatori coinvolti. Il transistor rivoluzionò i computer al punto che nei tardi anni '50 resero obsolete le valvole [12].
Nel 1954 il germanio viene sostituito con un altro semiconduttore, il silicio, molto più diffuso ed economico.
Il primo calcolatore a transistor fu costruito presso i Lincoln Laboratory del M.I.T., era a 16 bit e fu chiamato TX-0 (*Transistorized eXperimental computer 0*), da cui seguì il più evoluto TX-2, precursore della macchina messa in commercio solo diversi anni dopo (1961) dalla nascente Digital Equipment

Figura 2.5: UNIVAC della Eckert-Mauchley Computer Corporationn, primo calcolatore commerciale.

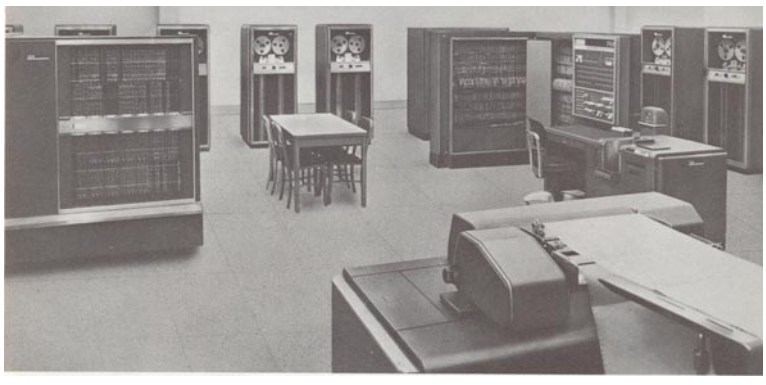


Figura 2.6: IBM modello 709.

Corporation (DEC), la PDP-1 (*Programmed Data Processor 1*), osteggiata dagli stessi finanziatori che non credevano potesse esserci mercato per i computer.
Intanto IBM reagiva costruendo un calcolatore a transistor sul modello 709, l'IBM 7090[4], con prestazioni doppie rispetto al PDP-1, ma che costava diversi milioni di dollari, contro i 120000 dell'altra. Furono quindi vendute decine di PDP.
Il PDP-1, di cui fu fornito anche il M.I.T, aveva uno schermo e la capacità di disegnare punti in qualsiasi zona di questo schermo: è stato il primo computer con monitor e tastiera. Gli studenti del M.I.T. lo programmarono per giocare a *spacewar* di fatto inventando il primo videogioco della storia.
DEC con il PDP-8 (considerato il primo 'minicomputer' per la piccola dimensione relativamente ai suoi contemporanei) introdusse il bus unico di connessione, e di questa macchina vendette decine di migliaia di esemplari.
L'IBM, che pure con la 7094 (macchina ad aritmetica binaria parallela su registri a 36 bit) si imponeva sul mercato, cominciò a fare veri profitti vendendo una 'piccola' macchina per le aziende, chiamata 1401 (senza aritmetica decimale senza registri), in grado di leggere e scrivere nastri magnetici, leggere e perforare schede e stampare output alla stessa velocità del 7094 ma ad una frazione del prezzo.
Nel 1963 la Bourroughs Corporation decideva di non soffermarsi sull'hardware ma dedicarsi al software: aggiungere all'hardware supporto per il compilatore e programmarla in Algol60, quindi con un linguaggio ad alto livello. Nacque così la Bourroughs B5000.

Supercalcolatori Nel 1964 una sconosciuta società, la Control Data Corporation (CDC) produsse il modello 6600, disegnato da Seymour Cray (1925-1996, figura rimasta legendaria come Von Neumann), con prestazioni di un ordine di grandezza maggiori del 7094, grazie all'**alto parallelismo** e fu subito amato dai matematici numerici. Era una macchina dotata di diverse unità funzionali che potevano lavorare contemporaneamente e necessitava di

[4]tra i progettisti del prototipo dell'IBM 7090 c'era il giovane promettente Ph.D. Michael J. Flynn.

Figura 2.7: DEC PDP-1, primo calcolatore commerciale a transistor, primo calcolatore con monitor e tastiera.

Figura 2.8: DEC PDP-8, primo minicalcolatore commerciale.

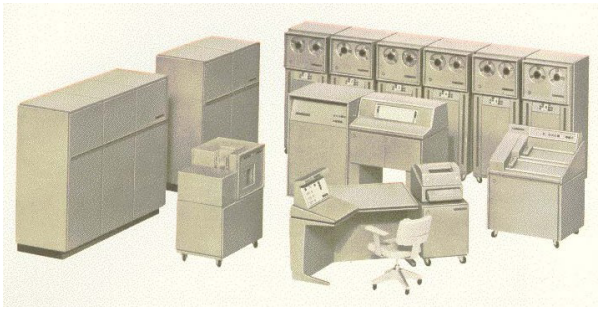


Figura 2.9: Bourroughs B5000, il primo calcolatore per un linguaggio ad alto livello.

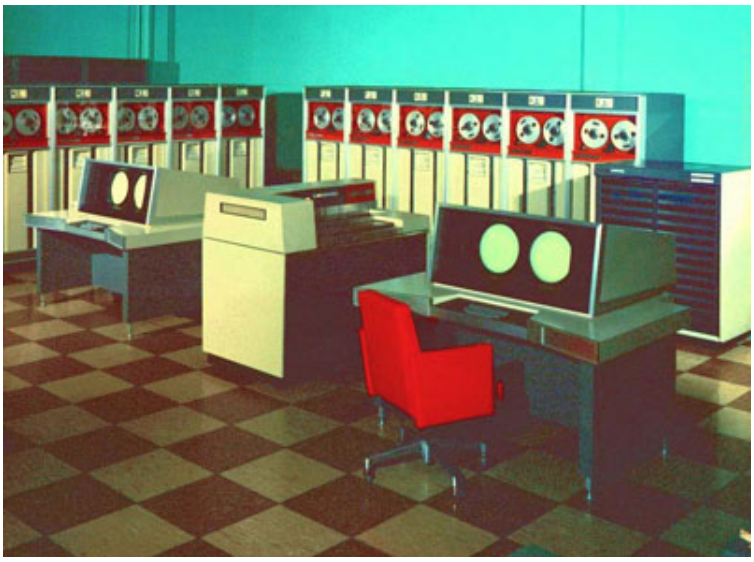


Figura 2.10: CDC modello 6600, il primo supercalcolatore.

un'attenta programmazione. Inoltre la CPU veniva dedicata interamente ai calcoli veri e propri, mentre tutti gli altri dettagli di gestione del programma e dell'input/output venivano affidati a CPU secondarie. In pratica, questa macchina conteneva idee chiave per i calcolatori prodotti decine di anni dopo ed è stato il primo *supercalcolatore* così chiamato.

2.4 Generazione III - Circuiti integrati o Chip

Con la miniaturizzazione dei componenti elettronici, venne l'idea di combinare diversi componenti (transistor, diodi, resistenze, ecc.) su una piastrina di semiconduttore di dimensioni più piccole di un francobollo: era il 1958 e Robert Noyce inventava il circuito integrato. I computer potevano essere più piccoli, più veloci e più economici.
IBM, ormai leader nel mondo dei computer, introduce il System/360 [13], basato su circuiti integrati e pensato sia per i calcoli scientifici sia per quelli commerciali: da questo progetto produce una famiglia di computer tutti dotati dello stesso linguaggio assemblativo ma di dimensione e potenza diverse. Questo colmava la distanza che c'era tra le sue due produzioni precedenti (7094 e 1401) e permetteva di creare software che avrebbe funzionato su tutte le macchine della famiglia (a meno di problemi di quantità di memoria), semplificando il lavoro e gli investimenti dei clienti e creando un trend poi seguito da molti altri produttori. Il 360 era multiprogrammabile (poteva contenere più programmi in memoria) e poteva anche 'emulare' altri computer, in particolare i modelli precedenti di IBM, realizzando la retrocompatibilità che ne velocizzò la diffusione. Aveva registri a 32 bit per l'aritmetica binaria ma anche una memoria orientata al byte, molto grande (2^{24} byte) per un'epoca in cui ogni byte costava alcuni dollari.
Dopo questa l'IBM produsse diverse macchine compatibili, ma quando si trattò di indirizzare una memoria di 2^{32} byte fu costretta a fare sostanziali cambiamenti per introdurre il nuovo tipo di indirizzamento.
Intanto i *microcomputer* fecero passi avanti: il DEC PDP-11 era una sorta di versione 'piccola' dell'IBM 360, e anch'esso ebbe la sua 'famiglia' e il suo enorme successo soprattutto nelle università.

Legge di Moore L'aumento delle prestazioni è anche la principale chiave per l'interpretazione della crescita tecnologica dei sistemi di calcolo: Gordon Moore (1929-vivente), cofondatore della Intel, nel 1965 [16] fece una predizione (figura 2.12), leggermente modificata da lui stesso dieci anni dopo [16], secondo cui

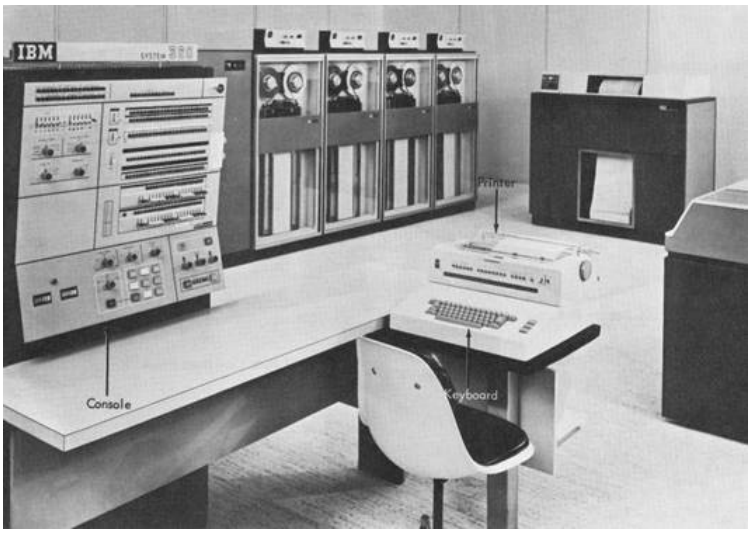


Figura 2.11: IBM System/360, la prima *famiglia* di calcolatori.

La complessità dei componenti a costo minimo raddoppierà all'incirca ogni due anni..

Qualche tempo dopo un altro ingegnere della Intel, David House [16], reinterpretava questa osservazione/previsione nel modo in cui è oggi ben nota:

La potenza di calcolo dei microprocessori raddoppierà ogni 18 mesi..

Tale legge ha trovato riscontro nella pratica per oltre 40 anni, anzi, si può dire che ha instaurato un ciclo virtuoso, spingendo gli avanzamenti tecnologici verso prodotti migliori e più economici, che a loro volta spingono la creazione di nuove applicazioni, che di nuovo incentivano l'avanzamento tecnologico, e così via.

Tassonomia di Flynn Le soluzioni che vengono progettate necessitano di una sistematizzazione che ne permetta la comprensione e l'evoluzione. Michael Flynn, ingegnere dell'IBM che vi lavorava da prima ancora di laurearsi, scrive un noto articolo (vedi Capitolo 3) in cui stabilisce le possibili componenti di un sistema di calcolo, evidenzia i fattori su cui intervenire per velocizzare l'esecuzione, e, prescindendo dalla realtà tecnologica del momento, classifica le possibilità architetturali secondo le scelte di concorrenza: tale

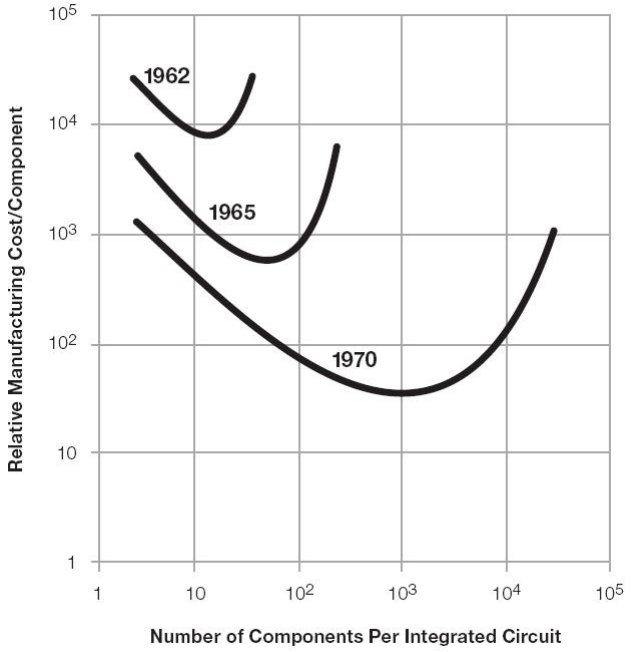


Figura 2.12: Grafico inserito nell'articolo di Moore del 1965.

classificazione era destinata a restare un punto di riferimento per gli anni a venire.

2.4.1 VLSI

Negli anni '80 si parlava ormai di tecnologia VLSI (*Very Large Scale Integration*) con cui si inserivano sullo stesso chip da migliaia a milioni di transistor. E i computer diventarono sempre più piccoli e più veloci. Nelle università si poteva fornire di computer ogni dipartimento, e con il crollo dei prezzi, anche i privati potevano permettersene uno: nascevano i *personal computer*, destinati ad usi nuovi, come l'elaborazione di testi, l'amministrazione domestica o videogiochi, e venduti in 'scatole di montaggio', senza software.
Nel 1973 Gary Kildall (1942-1994) aveva scritto il sistema operativo CP/M, scritto per il processore Intel 8080, i cui concetti fondamentali sarebbero stati la base per MS-DOS.
Steve Jobs (1955-2011) e Steve Wozniak (1950-vivente) nel 1976 progettaro-

no l'Apple, su cui funzionava CP/M e che poteva addirittura essere collegato ad una televisione da usare come schermo, e nel 1977 l'Apple II, che aggiungeva al testo scritto anche grafica e colori, veniva venduto già assemblato e con il sistema operativo AppleDOS. Questo personal computer ebbe grande diffusione tra privati e scuole, e, con la forza della loro semplicità di utilizzo, resero immediatamente la Apple (società fondata per l'occasione) un pericoloso concorrente per le altre compagnie.
Intel cominciò a distribuire il processore Intel 8080 con il CP/M su floppy disk, con tanto di file system e interprete di comandi scritti da tastiera (*shell*). Ma dopo il successo ottenuto dalla Apple, anche l'IBM si diede ai personal computer e nel 1981, IBM introdusse sul mercato, con tanto di libro degli schemi e dei diagrammi dei circuiti, il suo PC IBM, con sistema operativo Microsoft MS-DOS, che divenne subito il computer più venduto (e più clonato) della storia.
Intanto Apple produceva prima l'Apple Lisa, per la prima volta dotato di interfaccia grafica, e poi nel 1984 il più fortunato Macintosh. La Microsoft di Bill Gates cominciò a sviluppare, seguendo le idee della Apple, l'interfaccia grafica per MS-DOS chiamata Windows che divenne poi un vero e proprio sistema operativo evolutosi negli anni. In questo modo la concorrenza di IBM e Microsoft alla Apple era spietata, nonostante non potessero competere in termini tecnologici, perché i loro prodotti risultavano più economici. Il Macintosh sopravvisse a stento a questa guerra, e la società si è ripresa davvero soltanto a metà degli anni '90.
Del 1981 è anche la comparsa del primo calcolatore portatile, Osborne-1 della Osborne Computer Corporation, società che fallì poco dopo, incapace di battere la concorrenza.
Intel realizzò diversi processori sull'onda dell'8080, di cui ebbe particolare fortuna il 386, con tutti i suoi 'figli', compresi i Pentium. Microsoft proseguì i propri progetti affiancando le produzioni Intel, di fatto detronizzando IBM. Nel 1992 DEC presentò Alpha, una macchina RISC a 64 bit, dalle prestazioni nettamente superiori a tutti i precedenti, ma le macchine a 64 bit hanno cominciato a diffondersi solo molto dopo.

Figura 2.13: Apple II, il primo calcolatore della Apple a grande diffusione.

Figura 2.14: IBM PC 5150, il primo personal computer di IBM, il calcolatore più venduto e più clonato della storia

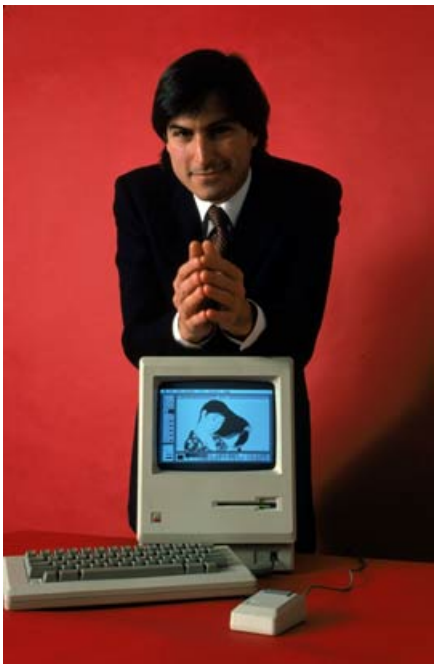


Figura 2.15: Steve Jobs con un Apple Macintosh 128K

Figura 2.16: Osborn 1, il primo computer portatile della storia

Supercalcolatori Nella seconda metà degli anni '70 l'introduzione dei sistemi vettoriali (che sfruttavano massicciamente concetti di **concorrenza** tra le operazioni, rif. Capitolo 3) ha segnato l'inizio del moderno supercalcolo: ricordiamo il Cray-1, progettato nel 1976 da un team di progettisti guidati da Seymour Cray per la Cray Research. I *supercalcolatori* offrivano un vantaggio in termini di performance di almeno un ordine di grandezza rispetto ai sistemi convenzionali dell'epoca [4].
Aziende, università e nazioni fanno a gara per produrne di sempre più potenti, spinti dalle necessità di applicazioni sempre più avanzate per risolvere problemi in tutti i campi. Per facilitare le statistiche sui computer ad alte prestazioni, nacque la *Top500* [18]: elenco dei siti che possiedono i 500 più potenti sistemi di calcolo del mondo, aggiornato due volte all'anno dal 1993[5].
Alla fine degli anni '80 l'interesse si concentrò sui sistemi di calcolo a memoria distribuita, in grado di superare il limite alla scalabilità dato dalla memoria condivisa. Tuttavia, con la nascita delle architetture RISC ed il conseguente incremento delle prestazioni dei microprocessori, che restavano relativamente economici, il rapporto costo-prestazioni del parallelismo a larga scala dei supercomputer cominciava ad apparire sconveniente, al punto che si cominciò a parlare di 'attacco dei micro-killer' [6]: il risultato fu che negli stessi supercalcolatori cominciarono ad essere utilizzati microprocessori commerciali invece di multiprocessori ad hoc nei sistemi massicciamente paralleli e così già nei primi anni '90 apparirono sul mercato nuovi **processori massicciamente paralleli** (MPP, rif. Capitolo 3), con prestazioni uguali o migliori rispetto ai precedenti, ma meno impegnativi dal punto di vista della costruzione [4].

[5]La classifica viene stilata secondo la prestazione ottenuta sul LINPACK Benchmark (vedi Capitolo 4), ovvero secondo il parametro **rmax**, ma per ogni macchina presente si riportano alcune altre informazioni: il produttore e/o venditore, il tipo di macchina secondo quanto riportato dal produttore/venditore, il sito di installazione, l'anno di istallazione o di più recente aggiornamento, il campo di applicazione in cui viene utilizzata, il numero di processori/core, la peak performance teorica, la dimensione del problema per cui si è ottenuto rmax, e la dimensione del problema su cui si ottiene la metà di rmax.

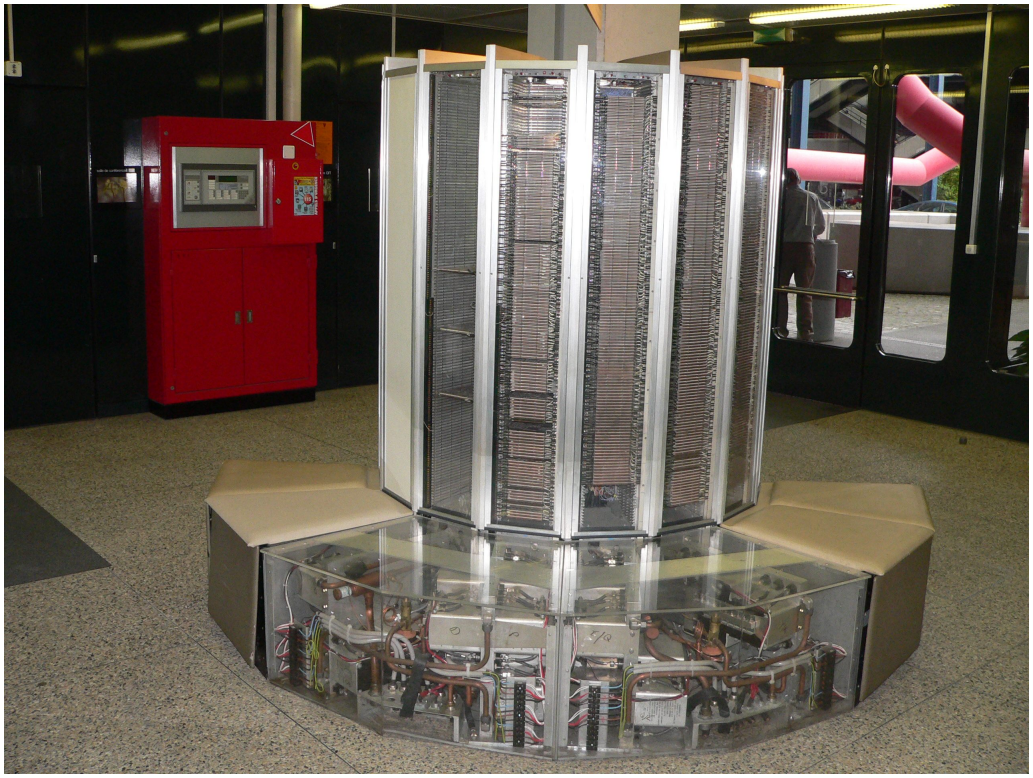


Figura 2.17: Il supercalcolatore Cray-1

2.4.2 Gli invisibili

Alla fine degli anni '90 l'industria ha cominciato a produrre e vendere **multiprocessori simmetrici** (SMP, rif. Capitolo 3) nelle stesse workstation. Ma, dal punto di vista di supercomputer e personal computer quello che è accaduto a partire dagli anni '90 non ha visto cambiamenti tali da poter definire una nuova generazione, benché società giapponesi abbiano tentato progetti visionari.
La tendenza principale è quella di rimpicciolire i computer, moltissimo, si parla di ULSI (*Ultra Large Scale Integration*): vengono anche prodotti negli anni '90 i primi PDA (Personal Digital Assistants, o anche *computer palmari*) con interfaccia grafica ed estremamente maneggevoli. Quello che però possiamo considerare come caratterizzante degli ultimi anni dal punto di vista tecnologico è la diffusione capillare di computer con hardware e software coprogettati, integrati in altri oggetti (elettrodomestici, orologi, carte di credito, telefoni...), computer sempre più piccoli, fino a poter essere considerati invisibili perché la loro presenza non viene percepita dall'utente se non per le loro funzionalità.
Quello che ci si aspetta dal futuro è che i computer vengano integrati in ogni cosa, appartenendo ad una moltitudine di gesti quotidiani di ogni persona, senza che se ne renda conto e senza che ne possa fare a meno: *ubiquitous computing*. Questo sè, rappresenta un grosso cambiamento per il mondo.

2.5 Considerazioni sulla legge di Moore

Come abbiamo detto, la legge di Moore non ha mai smesso di trovare applicazione nell'andamento della produzione industriale, ed è stata a sua volta un incentivo per lo sviluppo tecnologico. Osserviamo che valendo la relazione:

$$P = f * p$$

dove

- P è la prestazione del calcolatore, espressa in operazioni floating point al secondo (flops)
- f è la frequenza di clock, ovvero il numero di cicli in un secondo, espressa in Mhz, e dipende dal numero e dalla densità dei transistor sul chip,
- p è un fattore che dipende dall'organizzazione dei transistor sul chip,

ma non sempre si può aumentare f fissando p, e viceversa.
Dunque non necessariamente una maggiore frequenza di clock significa un maggior numero di flops, né una minore frequenza di clock significa un minor numero di flops. Infatti [4] il miglioramento tecnologico del chip e l'aumento del numero di processori fino ad un certo punto hanno contribuito nella stessa proporzione alla crescita annuale delle performance di calcolatori e supercalcolatori, ma è sempre stato chiaro che il miglioramento delle potenzialità del chip ha dei limiti: non può proseguire per sempre e comunque non con regolarità. Questo vuol dire che secondo la formulazione originale di Moore, tale legge è destinata a non valere per sempre, perché prima o poi (ma ormai più prima che poi) non avrà più senso rimpicciolire ulteriormente ed ammassare i transistor sul semiconduttore. Infatti, come si può vedere in figura 2.18, già attualmente l'aumento di transistor, che pure è continuato, non corrisponde ad un equivalente aumento dei Mhz, né della potenza, verosimilmente perché non è accompagnato da un adeguato miglioramento del parallelismo all'interno del processore.
É chiaro che il fattore su cui puntare perché la predizione nella sua formulazione definitiva valga ancora e non si fermino la crescita e le potenzialità dei sistemi è senza dubbio il **parallelismo**, ragione per cui le industrie hanno smesso di produrre microprocessori che contengano singole CPU. Per continuare a chiamare 'microprocessore' l'intero chip ed aggirare le ambiguità è stato inventato il termine *core* per indicare le componenti di un microprocessore che sono di fatto delle complete CPU.
Si può proporre allora una nuova formulazione della Legge di Moore:

Il numero di core per chip raddoppierà ogni due anni.

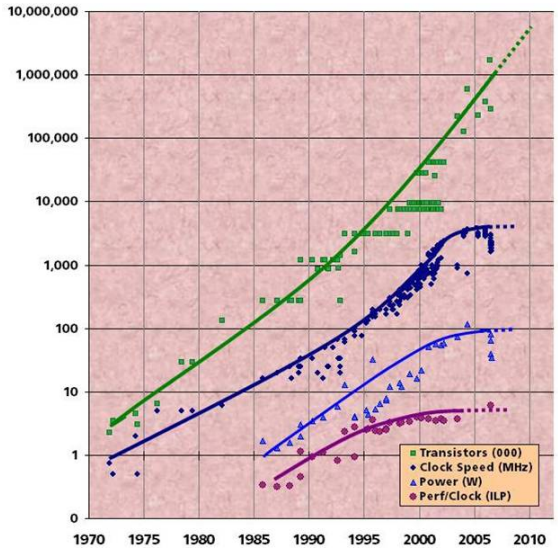


Figura 2.18: La frequenza di clock e le prestazioni non crescono con il numero di transistor.

2.6 L'importanza dei modelli

Come abbiamo fatto anche noi finora, quando si parla di parallelismo, per lo più l'attenzione viene posta sull'hardware. In realtà però è chiaro che una tecnologia di successo deve avere un modello di programmazione stabile ed efficace che sopravviva all'avvicendarsi delle applicazioni e all'evoluzione architetturale e sappia sfruttare a pieno le potenzialità della macchina sottostante. É quindi importante ogni fattore che semplifichi il processo di costruzione, testing e tuning di un'applicazione.

Essere in grado di comprendere a pieno i sistemi paralleli diventa fondamentale per la progettazione di algoritmi che non possono fare a meno di tenerne conto.

Per gli scienziati è sempre stata chiara la necessità di modellare funzionalmente le architetture perché si potessero costruire degli algoritmi efficienti, oltre che efficaci, che sapessero sfruttare le novità architetturali, con la consapevolezza del guadagno ottenibile nella scelta di un ambiente di calcolo piuttosto che un altro.

Da molti anni ormai chi progetta applicazioni in ambiente di calcolo parallelo

non può fare a meno di servirsi di parametri ben consolidati per il calcolo del *tempo d'esecuzione* del proprio algoritmo, del suo *speed up* rispetto ad un algoritmo sequenziale e quindi della relativa *efficienza* (rif. Capitolo 4), ed i modelli ormai classici per la valutazione delle performance, oltre a dare agli sviluppatori la consapevolezza delle possibilità offerte, hanno creato la giusta base di comprensione per la nascita di nuove idee e nuovi progetti.
Tuttavia questo approccio diventa sempre più difficile a mano a mano che le architetture si complicano. Le macchine attuali mescolano principi diversi e tentano di trarre beneficio da ogni sviluppo passato, e a volte è impossibile ricondurle ad una categoria precisa di cui esista un modello consolidato.
Prima di arrivare ad affrontare la costruzione di un modello per un'architettura molto recente quale quella delle GPU (rif. Capitolo 5), ricostruiamo l'organizzazione delle architetture parallele nelle forme in cui si sono evolute negli ultimi cinquant'anni, riconducendoci alla già citata *tassonomia di Flynn*.

Capitolo 3

Concorrenza nelle architetture

Come abbiamo visto, subito dopo la seconda guerra mondiale, John Von Neumann, a partire dalle esigenze di calcolo che l'evento bellico aveva sollevato, partecipando alla realizzazione dell'EDVAC, sviluppò il disegno di quell'architettura che è passata alla storia con il suo nome [1]. Qualche anno dopo i calcolatori erano diventati una realtà di mercato e si imponeva all'attenzione degli studiosi l'esigenza di sistematizzare i concetti progettuali per indirizzare il progresso. Su questo sfondo nacquero le intuizioni di Gordon Moore sull'andamento del miglioramento delle prestazioni, ma anche le idee di Michael Flynn riguardo l'importanza della contemporaneità di operazioni diverse nella realizzazione di sistemi di calcolo particolarmente veloci.
In questo capitolo approfondiremo il concetto di *concorrenza* e ne vedremo le possibilità realizzative, così come sono state fino ad oggi recepite. Nel prossimo capitolo ci soffermeremo sul concetto di *velocità* e di valutazione della prestazione del sistema.

3.1 Classificazione secondo Flynn

Nel 1966 Michael J. Flynn si concentrava sull'aumento delle performance dei sistemi di calcolo ed affermava che i computer 'molto veloci' potevano essere classificati come

- Single Instruction Stream / Single Data Stream (SISD),

- Single Instruction Stream / Multple Data Stream (SIMD),
- Multiple Instruction Stream / Single Data Stream (MISD),
- Multiple Instruction Stream / Multple Data Stream (MIMD).

Dove con la parola *stream* intendeva un flusso, una sequenza di dati o istruzioni, così come percepito da una macchina che sta eseguendo un programma. Ovviamente, le componenti, le possibilità e i limiti di un sistema da lui considerati in quell'ormai famosissimo lavoro [2] prendono spunto dalla tecnologia di allora, cioè quella che prendeva forma dopo gli sviluppi degli anni '60, e che oggi appare certamente obsoleta. Tuttavia, come è chiaro da quanto descritto nel capitolo precedente, non va dimenticato che tutto quello che in quegli anni accadde, fu studiato e fu prodotto, ha in qualche modo cambiato il mondo e certamente ha cambiato il problem solving: i computer conquistavano la scena, la scienza tutta ne traeva grande vantaggio (nasce la computational science), e venivano poste le basi per tutto quello a cui oggi siamo così abituati.

3.1.1 Tassonomia di Flynn

Flynn nel suo lavoro [2] parte dal definire il processo di calcolo come l'esecuzione di una sequenza di istruzioni su un insieme di dati, dove ogni istruzione manipola uno o due elementi di questo insieme, sottolineando che questa definizione è indipendente dal tipo di tali dati, e quindi dalla complessità delle relative istruzioni. L'importante è arrivare al concetto di programma come insieme ordinato di istruzioni. A questo punto si possono definire:

- Instruction Stream: sequenza di istruzioni eseguite dalla macchina,
- Data Stream: sequenza di dati richiesta dal flusso di istruzioni, compresi input e variabili d'appoggio.

Servendosi di queste due definizioni Flynn risolve l'ambiguità del termine *parallelismo*, stabilendo che nell'organizzazione del lavoro si può avere *molteplicità* di istruzioni e/o operandi nella stessa fase dell'esecuzione, arrivando a

concludere che l'efficacia di ogni classe di architetture dipende poi dalla natura del problema da risolvere e da algoritmi in grado di sfruttarne a pieno le potenzialitè.

Per descrivere bene le problematiche relative all'ottimizzazione delle performance di un sistema, Flynn pone le seguenti definizioni:

Definizione 3.1 (Larghezza di banda (bandwidth))**.** *La larghezza di banda è espressione della frequenza di occorrenze nel tempo.*

Definizione 3.2 (Computational bandwidth)**.** *Computational bandwidth è il numero di istruzioni processate al secondo.*

Definizione 3.3 (Storage bandwidth)**.** *Storage bandwidth è la velocità di recupero di operandi e istruzioni.*

Definizione 3.4 (Latenza, o periodo di latenza)**.** *La latenza è il tempo totale necessario per processare completamente un dato unitario in una fase del calcolo.*

Processare un'istruzione è un lavoro logicamente divisibile in più fasi successive:

1. generazione dell'indirizzo dell'istruzione,
2. recupero dell'istruzione (instruction fetch),
3. decodifica dell'istruzione,
4. generazione dell'indirizzo dell'operando,
5. recupero dell'operando (operand fetch),
6. esecuzione,

(dove le fasi 4 e 5 possono ripetersi per tutti gli operandi coinvolti).
Ognuna di queste fasi dovrà essere realizzata da una **componente del sistema**. La molteplicità nel flusso di istruzioni o di dati corrisponde ad una

molteplicità delle componenti: in un sistema dunque, la stessa componente può comparire più di una volta, in tal caso si dirà che quella componente ha *molteplicità maggiore di uno*. La progettazione delle singole componenti e della relativa molteplicità introduce *vincoli* all'aumento delle istruzioni o dei dati con cui si può lavorare contemporaneamente. In pratica per ogni componente resta definita una larghezza di banda e con essa un numero di istruzioni che possono essere elaborate contemporaneamente da quella componente.
Spiegati questi concetti, possiamo riportare la definizione che diede Flynn di:

Definizione 3.5 (Concorrenza di un sistema)**.** *La concorrenza di un sistema è il rapporto tra il numero di istruzioni processate simultaneamente dal sistema stesso e la molteplicità della componente che impone i vincoli più stretti.*

Con l'espressione 'vincolo più stretto' intendiamo la maggiore limitazione introdotta da una delle componenti, che inevitabilmente limita la concorrenza e tutta la prestazione del sistema. Per individuare a questo punto le componenti a cui riferirci, Flynn stabilisce che essenzialmente un sistema si può dire costituito da:

- sistema di memorizzazione,
- sistema di esecuzione,
- gestore delle istruzioni.

Su ognuna di queste si può intervenire per ottimizzarne il funzionamento.

Per quanto riguarda il *sistema di memorizzazione*, si possono prevedere tecniche di interleaving per aumentare la memory bandwidth, code e registri per gestire i conflitti di richieste dalla memoria in caso di branching, un registro istruzioni che possa contenere n istruzioni successive ed m precedenti a quella considerata, compresi i path alternativi, un registro operandi destinato all'*unità d'esecuzione*, che userà così istruzioni più corte per trovare i dati in questo buffer, mentre il *gestore di istruzioni* si occuperà in fase di decoding

di far arrivare gli operandi dalla memoria nel registro.

Relativamente all'esecuzione Flynn considera l'opportunità di destinare ogni tipo di operazione floating point ad una sottounità dedicata e quindi specializzata, nonché più piccola, con conseguente aumento dell'efficienza nonché della computational bandwidth, e suggerisce anche la possibilità di strutturare una sorta di pipeline all'interno di ognuna per ridurre ulteriormente la latenza.
Tuttavia, in ogni caso restano sempre da affrontare i branching, in particolare quelli data-dependent: il tempo per risolverli è una quantità praticamente fissa, data la latenza d'esecuzione e/o accesso caratteristica del sistema. Tale tempo viene ridotto in caso di utilizzo del registro istruzioni descritto, ma non si può annullare, e tipicamente influisce sulle performance di qualsiasi architettura, creando dipendenze che non si possono aggirare.

Esplicitati questi dettagli, Flynn suggerisce alcune alternative progettuali che sfruttano o meno le possibilità di concorrenza per aumentare le prestazioni, e le fa poi ricadere nelle quattro classi inizialmente citate.

SISD: anche in una 'semplice' macchina da uno solo flusso di istruzioni e un solo flusso di dati si può realizzare una forma di concorrenza permettendo l'esecuzione simultanea delle diverse fasi di processing per istruzioni diverse (in una sorta di *pipeline macro-funzionale*). Resterà però sempre il collo di bottiglia della decodifica: una sola istruzione per unità di tempo.

SIMD: (*molteplicità di sistemi d'esecuzione*) immaginando di poter avere n unità d'esecuzione universali (non specializzate) ognuna con il proprio registro operandi, il flusso unico di istruzioni agisce simultaneamente su tutte le unità (vedi figura 3.1). In questo modo non si elimina la latenza dovuta al recupero degli operandi, e si perde l'efficienza della specializzazione, ma si aumenta certo la computational bandwidth, almeno quando è possibile ottenere una certa regolarità nel trattamento dei dati dal punto di vista algoritmico.

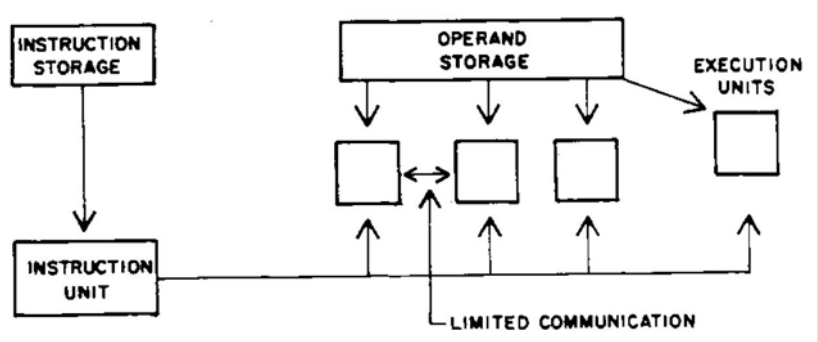


Figura 3.1: Struttura di un'architettura SIMD, secondo descrizione originale di Flynn (1966).

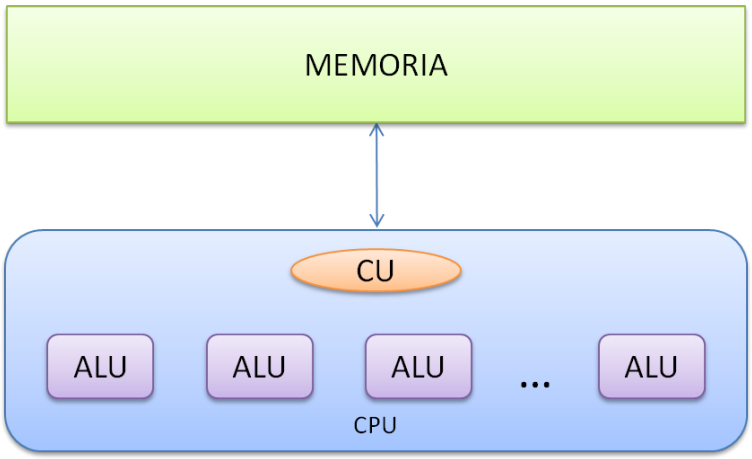


Figura 3.2: Struttura logica SIMD.

MISD: (*molteplicità di gestori d'istruzione*) se realizziamo n *macchine virtuali*[1], ognuna con il suo gestore di istruzioni (compreso il relativo registro) condividenti però il sistema di memorizzazione attraverso cui sono messe in relazione, si può immaginare che ogni macchina esegua una sequenza di programma indipendente (vedi figura 3.3). Perché la performance sia effettivamente migliorata è ovvio che la larghezza di banda per i dati deve essere circa n volte maggiore di quella associata ai registri istruzione. In una parti-

[1]Flynn si riferisce con questa espressione alle 'sottomacchine' che dal punto di vista logico compongono la macchina che si sta prendendo di volta in volta in considerazione.

colare versione di questa architettura (secondo alcuni), si suppone che ogni macchina virtuale abbia a disposizione un segmento dell'unità esecutiva e che possa agire solo sul dato risultante dalla macchina che la precede in numerazione, realizzando di fatto, anche in questo caso un meccanismo di pipeline.

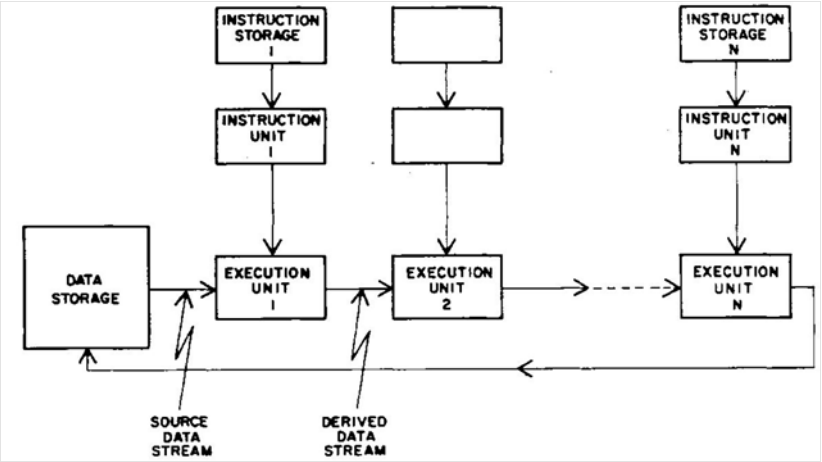


Figura 3.3: Struttura di un'architettura MISD, secondo descrizione originale di Flynn (1966).

MIMD: (*molteplicità di tutte le componenti*) se le n macchine virtuali che immaginiamo oltre al gestore di istruzioni e all'unità esecutiva hanno anche un proprio sistema di memorizzazione, con i relativi registri dati, allora ogni macchina è di fatto in grado di eseguire del tutto un flusso di istruzioni indipendente, su un certo flusso di dati, riducendo tutte le necessità di sincronizzazione, sempre ammesso che il problema da affrontare permetta una distribuzione del lavoro così netta. Sicuramente però si tratta di un tipo di architettura molto utile agli ambienti di time-sharing. Quelli che a Flynn sembravano i punti su cui soffermarsi, e su cui si sono effettivamente soffermati molti degli sforzi di avanzamento tecnologico successivi, sono:

- L'interconnessione tra le diverse *macchine virtuali*,
- La natura universale del singolo modulo che poteva limitarne l'efficienza,

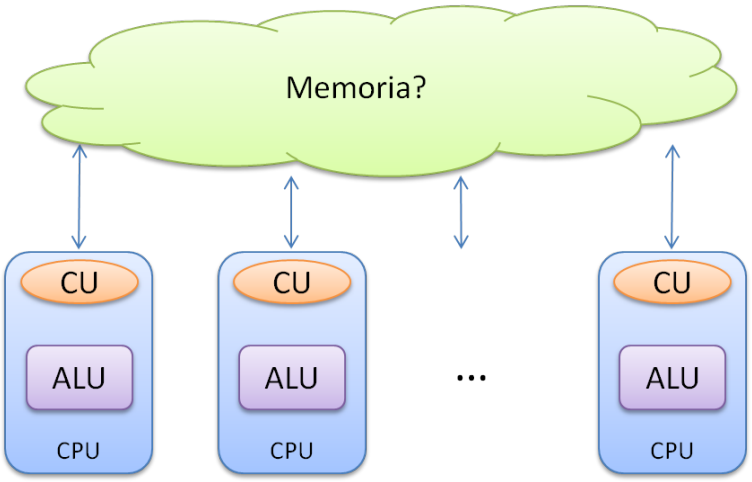


Figura 3.4: Struttura logica MIMD.

- La classe di problemi a cui potevano essere destinate macchine di questo genere.

3.2 Architetture Parallele reali

Quando si tratta di valutare un'applicazione e il sistema *reale*[2] su cui viene eseguita, l'aspetto più naturale da considerare, come vedremo nel Capitolo 4, è il **tempo d'esecuzione**, ovvero il tempo che passa tra l'inizio e il completamento di un task, che risente ovviamente delle latenze relative ad ogni fase del calcolo. Tuttavia, un altro parametro non meno importante, come evidenziato dall'analisi di Flynn, è la *bandwidht*, o anche **throughput**, cioè il numero di task portati a termine nell'unità di tempo: anche questo parametro influisce sul tempo d'esecuzione quando, come generalmente è, l'applicazione prevede più di un task.
Dunque, intuitivamente, per rendere più 'veloce' un processore, ci sono due possibili strade:

[2]Le macchine reali si contrappongono a quelle ideali spesso utilizzate per costruire modelli funzionali e metriche di prestazione. Quando si prendono in considerazione macchine reali, o concrete, non si può prescindere da problematiche che i modelli astratti possono o devono invece ignorare [33, 34].

- aumentare la velocità di clock e/o intervenire sui meccanismi tecnologico-fisici per ridurre le latenze;
- aumentare il numero di operazioni realizzabili contemporaneamente, riuscendo così ad aumentare il throughput, per mascherare le latenze;

Mentre obiettivi come la riduzione della velocità di clock sono raggiungibili entro dei limiti, relativi al momento storico ma anche assoluti, il parallelismo è un meccanismo a cui si può sempre ricorrere per aumentare le prestazioni. D'altra parte l'analisi di Flynn lo evidenzia benissimo.
Quando ci troviamo nell'ambito delle architetture parallele il throughput è l'aspetto che maggiormente ci interessa: più che il numero di cicli di clock per eseguire un istruzione (CPI) diventa importante un'altra unità di misura che è il numero di istruzioni per ciclo di clock (IPC) [11]. Aumentare l'IPC del sistema significa aumentarne il parallelismo.
A questo scopo, il parallelismo si può introdurre nell'architettura principalmente in più forme:

- a livello di istruzione (ILP): si realizza nell'ambito della singola istruzione, per ottenere che il singolo processore ne possa elaborare un maggior numero nell'unità di tempo;
- a livello di threads: si realizza la coesistenza di più Program Counter;
- a livello di processore: si mettono insieme più processori che possono lavorare simultaneamente su task diversi o sullo stesso task.

In generale trovare calcolatori classificabili come MISD è difficile, invece, come vedremo, sono molto implementate le categorie SIMD e MIMD.
Comunque, come si intuisce facilmente, quando si tratta di introdurre parallelismo nel proprio lavoro le possibilità sono molteplici, le combinazioni immaginabili di quanto stiamo per vedere sono potenzialmente illimitate. Per questo motivo, le macchine con cui abbiamo a che fare oggi non possono più essere catalogate nettamente nell'ambito della tassonomia di Flynn, così come da lui introdotta.
Quello che è accaduto negli anni è stato il progressivo sviluppo di architetture

sempre più articolate, nella cui progettazione entravano concetti appartenenti a punti di vista diversi e soluzioni che cercavano vantaggio nell'accostamento e/o *stratificazione* di modelli organizzativi anche non omogenei.

3.2.1 Parallelismo a livello di istruzioni

Per quanto riguarda l'ILP, l'intenzione è quella di (iniziare ad) eseguire più istruzioni nello stesso intervallo di tempo, a partire dallo stesso *program counter*. Sono stati sviluppati diversi modi per farlo.

Pipelining Supponendo separabili le diverse fasi di un'operazione, esse vengono assegnate ad unità funzionali diverse, in modo tale che, quando un'unità termina di lavorare per un'operazione possa già essere dedicata alla prossima, ammesso che essa sia indipendente dalla precedente. Per il dettaglio si guardi la figura 3.5. Il meccanismo è quello di una catena di montaggio, di cui possiamo trovare esempi nei più svariati ambienti del mondo reale (dalle lavanderie [11], alle pasticcerie [12], e così via), e questo tipo di parallelismo viene chiamato *temporale*. La pipeline è così suddivisa in *stadi*, ed il tempo che impiega un'operazione ad attraversarli tutti si chiama *latenza di pipeline* ed è funzione del numero di stadi. Maggiore è il numero di stadi, inoltre, maggiore sarà il throughput.
Nelle architetture esistono pipeline per l'operazione di prelievo-esecuzione delle istruzioni, ma anche nelle singole ALU, per esempio per le operazioni aritmetiche floating point, per le quali a segmenti diversi dell'unità viene assegnata l'esecuzione di una fase dell'operazione.

Processori superscalari Quando esistono più unità funzionali per la stessa fase di un'operazione, allora parliamo di architettura superscalare. In pratica, già negli anni '80 si sono visti nascere processori (come il primo Pentium di Intel) con più di una pipeline, o con diverse unità funzionali uguali associate alla stessa pipeline (Pentium II di Intel), come da figura 3.6. Un sistema del genere permette anche di bilanciare la diversa latenza richiesta

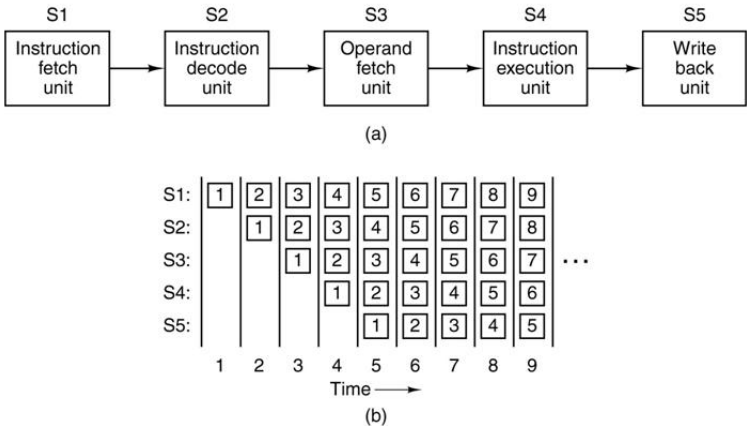


Figura 3.5: (a) Pipeline a 5 stadi. (b) Stato degli stadi in funzione del tempo (9 cicli di clock).

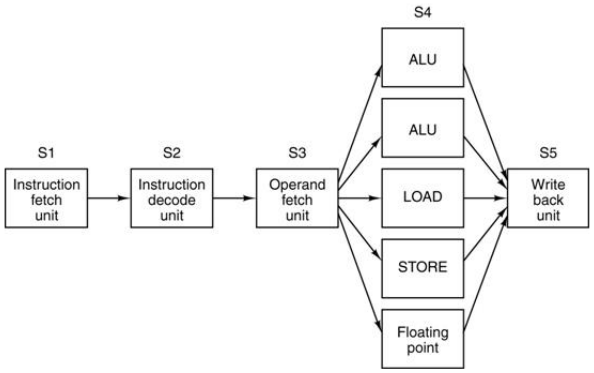


Figura 3.6: Processore superscalare con cinque unità funzionali.

dalle diverse fasi della pipeline [12], e può essere sfruttato anche attraverso l'introduzione di tecniche di VLIW (*Very Long Instruction Word*), in cui le istruzioni sono composte di più sottoistruzioni, ognuna per un'unità funzionale precisa.

Le architetture superscalari si sono poi 'complicate' ed oggi con questa espressione si indicano tutti quei processori che possono lanciare più di un'istruzione nello stesso ciclo di clock, o comunque che lanciano istruzioni con intervalli molto più piccoli di quelli necessari alla loro esecuzione.

3.2.2 Parallelismo a livello di thread

Nello stesso chip, quindi per quanto riguarda lo stesso processore, si può pensare ad un altro livello di parallelismo, quello basato sui **thread**. Mettendoci in questo punto di vista infatti, vediamo che, ogni volta che l'esecuzione di una sequenza di istruzioni di un thread da parte della pipeline subisce un periodo di stallo, questo tempo 'sprecato' potrebbe invece essere sfruttato da un altro thread. Questo è possibile perché thread diversi hanno stati differenti ed in particolare diversi *program counter* (PC), a patto che l'hardware sia capace di supportare rapidamente il passaggio da un thread all'altro.
Esistono molte diverse tecniche di implementazione del **multithreading** che qui abbiamo appena accennato, a seconda delle scelte dei momenti di commutazione.
L'Hyperthreading realizzato nel Pentium 4 di Intel è un esempio di parallelismo di questo tipo.

3.2.3 Parallelismo a livello di processore

Per sopperire ai limiti intrinseci degli approcci visti finora, si è introdotto fin dagli anni '70 un parallelismo a livello più alto che coinvolgeva diversi processori in modo sincrono o asincrono.

Array computer L'osservazione che molti rilevanti problemi dell'ingegneria o della fisica riguardano strutture con una forte regolarità, su cui vanno spesso fatti calcoli identici su porzioni diverse, portò alla progettazione di macchine composte di un gran numero di processori, tutti collegati ad una singola unità di controllo, che potevano quindi eseguire la stessa operazione su dati diversi, in perfetto spirito SIMD. Questo tipo di macchina differisce dai **processori vettoriali** (tipo i diversi modelli prodotti dalla Cray) che, pur lavorando anch'essi su array di dati, fanno eseguire tutto il lavoro ad un unico processore fortemente pipelined. Gli array computer sopravvivono ancora nelle idee di progettazione di alcuni microprocessori recenti, come il Pentium 4 [12].

Multiprocessori La differenza tra questa tipologia di macchine e le precedenti è che stavolta anche le unità di controllo sono diverse, rendendo possibili forme di parallelismo asincrono.
Un multiprocessore è un sistema composto da più CPU complete, collegate ad una memoria da intendersi fisicamente accessibile a tutte, e come tale sicuramente una macchina MIMD. Tuttavia, la necessità di organizzare gli accessi a tale memoria ha portato all'evoluzione in schemi diversi a seconda di come la immaginiamo logicamente assegnata, unica per tutte le CPU o strutturata in più memorie separate, ognuna privata ad una CPU [12].
I multiprocessori a memoria condivisa (*Shared Memory MultiProcessors*, o SMP), come si vede in figura 3.7, hanno a disposizione un unico spazio di indirizzamento a cui possono accedere in maniera (percepita) uniforme (impiegando sempre lo stesso tempo di accesso), rientrando così nella sottocategoria dei multiprocessori UMA (*Uniform Memory Access*), o in maniera non uniforme (con un tempo di accesso che dipende dal dato e dal processore), rientrando così nella sottocategoria dei multiprocessori NUMA (*Nonuniform Memory Access*) [11]. Tra gli SMP possiamo annoverare i microprocessori **multicore** che oggi sono (o possono essere) sulle scrivanie di tutti.
Si tratta certo di macchine in cui hanno particolare rilevanza i concetti di coerenza e consistenza della memoria e quello di sincronizzazione.
Gli SMP poi, possono poi essere di tipo simmetrico (Symmetric SMP, o S-SMP), come i Multicore prodotti da Intel (figura 3.8), o asimmetrico (Asymmetric SMP, o A-SMP), come i CELL di Sony/IBM (figura 3.9), e le *Graphics Processing Unit* (GPU), due esempi di architettura eterogenea a memoria condivisa.
Nei multiprocessori a memoria distribuita (anche detti *Multicomputer*) le diverse CPU devono comunicare attraverso un esplicito *scambio di messaggi*, o *message passing*, che garantisce la coordinazione del lavoro (figura 3.10). Sono stati fatti diversi progetti di calcolatori di questo tipo (*Massive Parallel Processors*, MMP), che hanno permesso il raggiungimento di prestazioni molto elevate, risultando tuttavia molto costosi.

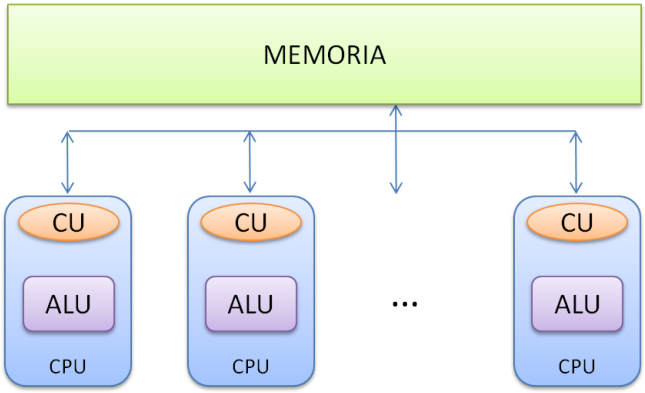


Figura 3.7: Struttura logica MIMD a memoria condivisa.

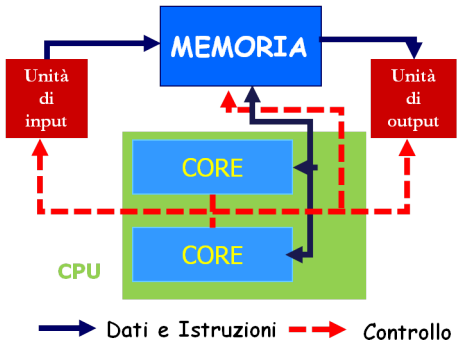


Figura 3.8: Schema logico di un multiprocessore Dual Core.

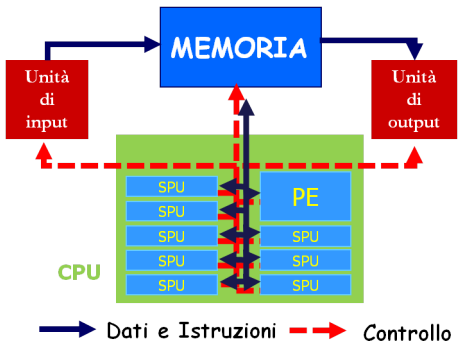


Figura 3.9: Schema logico del microprocessore CELL.

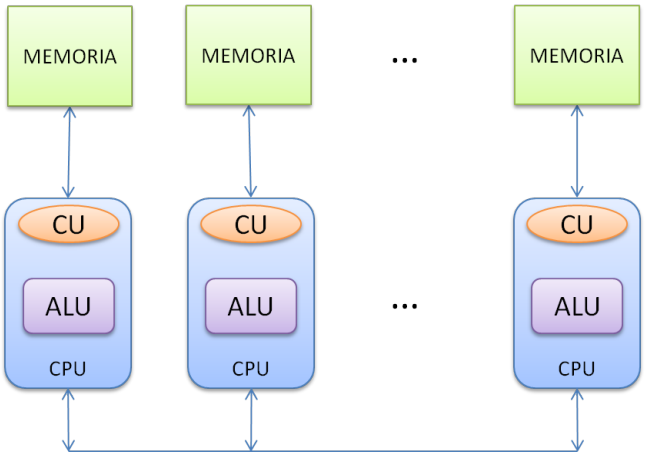


Figura 3.10: Struttura logica MIMD a memoria distribuita.

Cluster Per realizzare una macchina MIMD a memoria distribuita che abbia costi di produzione molto meno impegnativi, si ricorre in genere ai cluster, ovvero insiemi di calcolatori autonomi collegati tra loro attraverso le interconnessioni di I/O, e quindi con connettori e cavi tipici di una rete standard. Ogni calcolatore ha una sua copia distinta di sistema operativo, il che fa crescere i costi di amministrazione, ma questo inconveniente è facilmente rimediabile attraverso l'utilizzo di macchine virtuali (VM). Le connessioni di I/O hanno certo larghezza di banda inferiore e latenza maggiore rispetto alle connessioni interne, tuttavia, la facilità ed economicità della loro costruzione si dimostra compensare la perdita. Attualmente il cluster è l'esempio più diffuso di calcolatore parallelo basato su message passing [11].

Quando si programma una macchina MIMD, in generale si scrive un singolo programma che viene eseguito da tutti i processori, e sono le istruzioni di salto condizionato a fare eseguire parti diverse del codice da processori diversi. Questo metodo di programmazione ' chiamato SPMD (Single Program Multiple Data).

Il vantaggio delle architetture di tipo SIMD è che tutte le unità di elaborazione parallele sono sincronizzate tra loro e rispondono a una sola istruzione che proviene da un unico program counter (PC). Dal punto di vista del pro-

grammatore si tratta di qualcosa di molto simile alla programmazione di macchine SISD, che intanto sono di fatto scomparse dal mercato. Sebbene ciascuna unità funzionale esegua la stessa operazione, le unità funzionali hanno i loro registri degli indirizzi, per cui ciascuna unità può contenere degli indirizzi diversi dei dati.
La motivazione originale che ha portato alla realizzazione delle architetture SIMD ' l'ammortamento del costo dell'unità di controllo, che viene distribuito su dozzine di unità funzionali. Un altro vantaggio è rappresentato dalle dimensioni ridotte della memoria dei programmi, poiché le architetture SIMD richiedono solo una copia delle istruzioni, le quali vengono eseguite simultaneamente, mentre le architetture MIMD basate su scambi di messaggi possono richiedere una copia delle istruzioni per ogni processore e le MIMD basate sulla condivisione della memoria richiedono istruzioni di cache multiple.
Le architetture SIMD lavorano a un livello ottimale quando devono gestire vettori all'interno di cicli *for*; quindi, affinché il parallelismo in tali architetture funzioni, occorre che i dati siano strutturati in modo simile, per cui si parla di parallelismo a livello dei dati. Il punto debole delle architetture SIMD è l'esecuzione dei costrutti *case* o *switch*, dove ciascuna unità funzionale deve eseguire un'operazione diversa sui propri dati a seconda del dato ricevuto. Le unità funzionali abbinate ai dati sbagliati nelle SIMD vengono disabilitate, in modo tale che le unità che hanno ricevuto i dati corretti possano continuare l'elaborazione. Questa organizzazione fa sì che il codice venga eseguito a $1/n$ delle prestazioni, dove n ' il numero dei casi nei costrutti citati.

Un tentativo di inserire le macchine viste nella tassonomia di Flynn è illustrato nella figura 3.11
Supponendo di volere avere a disposizione una macchina 'pronta' ad eseguire efficientemente ogni tipo di applicazione, superando talvolta i limiti del microprocessore, qualunque sia il suo progetto di base, si può ne di affiancargli un altro processore, specializzato, che chiamiamo **coprocessore**. In particolare sia l'aritmetica che la grafica si prestano all'uso di un coprocessore. Dal punto di vista fisico, può trattarsi di un dispositivo separato, magari su

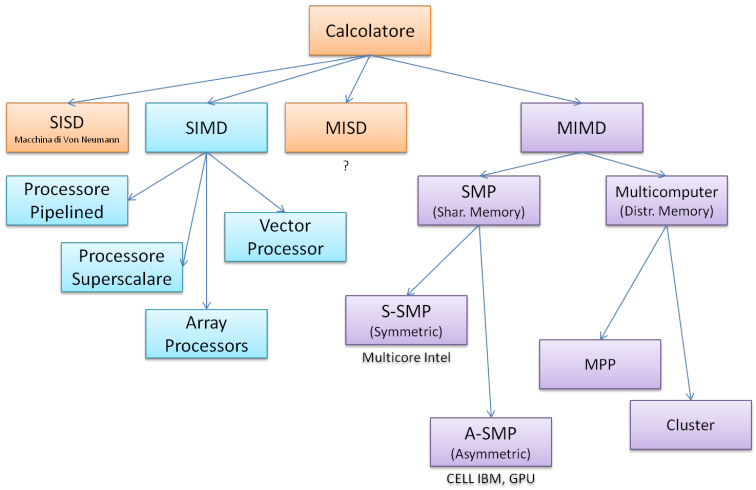


Figura 3.11: Estensione della tassonomia di Flynn.

scheda a innesto o sullo stesso chip, che è comunque subordinato al controllo del processore principale di cui assiste il lavoro.
Le Graphics Processing Units (GPUs) che ad oggi si sono guadagnate tanta fama, sono appunto dei coprocessori nati per essere destinati all'elaborazione grafica, la cui architettura si è evoluta molto e molto rapidamente, fino a riunire quasi tutti, se non tutti, i concetti visti nel capitolo 3.
Più avanti in questo lavoro arriveremo a proporre un modello per architetture di questo tipo da cui partiremo per la nostra analisi.

Capitolo 4

Valutazione delle Prestazioni

Come abbiamo visto nei primi due capitoli, fin dai primi giorni dalla nascita dei calcolatori elettronici si è parlato di *prestazioni*, mettendo a confronto i sistemi e misurando il progresso in questi termini. Per esempio i progettisti dell'ENIAC descrivevano il loro sistema appena costruito specificando che era mille volte più veloce del Mark-I. Cosa questo davvero significasse però, non è stato subito chiaro.
Ci si può riferire al concetto di *velocità* di una macchina o di un algoritmo in maniera intuitiva, ma le esigenze di confronto richiedono la sistematizzazione di questi concetti e di altri che ne conseguono.

4.1 Velocità di una macchina

Intuitivamente, il **tempo d'esecuzione** di un algoritmo (su una data macchina) è il tempo intercorso tra l'inizio dell'algoritmo e la sua fine.
Dato un algoritmo, allora, possiamo dire certamente se viene eseguito più velocemente su una macchina o su un'altra, dopo averlo eseguito su entrambe, ammesso che sia possibile.
Cosa significa però che una macchina è più veloce di un'altra, in generale? E' chiaro che serve un sistema di riferimento.
Inizialmente come misura della performance si intendeva il tempo impiegato a svolgere una singola operazione[11], per esempio una addizione. Dal

momento che le istruzioni cominciavano a diversificarsi, questo tipo di definizione non si adattava più alle esigenze di confronto. Allora, per considerare queste differenze, veniva calcolata una combinazione delle frequenze relative di ogni istruzione su un computer in diversi programmi, poi si moltiplicava il tempo di ognuna per il suo peso nella combinazione, ottenendo il tempo medio di un'istruzione generica (o anche il numero di clock per istruzione, CPI): essendo gli insiemi di istruzioni molto simili tra loro, questo risultava un buon metodo di confronto, un passo verso la definizione dell'unità di misura MIPS (Million Instructions Per Second).
Poi col complicarsi dei linguaggi di programmazione già dall'introduzione delle gerarchie di memoria e del pipelining non ha avuto più molto senso soffermarsi sul tempo d'esecuzione di una singola istruzione e il MIPS ha perso parte della sua attendibilità.
Dal momento che le applicazioni si basano per lo più sull'esecuzione di operazioni in virgola mobile, l'unità di misura a cui più spesso si fa riferimento per parlare di velocità di una macchina è il FLOPS (numero di operazioni in virgola mobile eseguite in un secondo).
In generale, per facilitare il confronto, la cosa più giusta da fare sembrò presto la costruzione di un insieme di applicazioni che potessero essere utilizzate come *benchmark*[1].
Il compito della costruzione di tali benchmark però non è mai stato facile, data la varietà di sistemi operativi e linguaggi, tuttavia col tempo si sono affermati alcuni *kernel benchmark*, ovvero piccole porzioni di programmi reali in grado di stressare la macchina sotto il punto di vista inteso, estratti ed utilizzati come riferimento. Tra questi, Linpack[2] è il benchmark utilizzato,

[1] letteralmente "segno, punto di riferimento" [rif. Dizionario Garzanti Inglese-Italiano], è una parola comunemente usata anche in discipline economiche, e indica un parametro ogettivo di riferimento, che consente un confronto sistematico tra attività o prodotti, in questo caso tra macchine calcolatrici.

[2] LINPACK è una libreria di software per l'algebra lineare, scritta in FORTRAN [19]. Il LINPACK Benchmark consiste nella risoluzione di un sistema di equazioni lineari la cui dimensione viene scalata per ottimizzare l'esecuzione sulla macchina data. Non permette certo di stabilire in assoluto la velocità di un calcolatore, ma, trattandosi di un problema molto regolare la prestazione ottenuta è abbastanza alta da poter essere considerata una buona approssimazione della *peak performance* della macchina. In caso di calcolatori paralleli si usa anche il pacchetto HPL [20, 18].

ad esempio, per la stesura della già citata classifica (vedi Capitolo 2) dei Top500.
In ogni caso nucelei di questo tipo sono più adatti ad isolare le prestazioni di alcuni particolari aspetti di un computer e a spiegare la ragione della differenza in termini di performance dei programmi sulle diverse architetture.
Nel 1988 fu creata la System Performance Evaluation Cooperative (SPEC) che comprendeva (e comprende) rappresentanti di molte compagnie del settore accordatesi su un certo insieme di programmi da utilizzare come benchmark. Nel 1991 fu aggiunta la misura del **throughput**, più utile per valutare il confronto tra un monoprocessore e un multiprocessore. Nel tempo SPEC ha aggiornato molte volte la propria suite di benchmark per le CPU ed ha cominciato ad affiancarle altre suite, dedicate ad altri aspetti architetturali. Tra gli altri, dal 2008 fornisce un insieme di benchmark anche specifici per la grafica, per il calcolo scientifico ad alte prestazioni e per l'energia.

Tuttavia, i dati che si possono raccogliere in questo modo non servono alla reale comprensione del miglioramento tecnologico che pure quantificano, né aiutano la progettazione di algoritmi adatti a sfruttare le potenzialità delle macchine. C'è bisogno di metriche che tengano conto sia delle caratteristiche dell'architettura, sia dei quelle dell'algoritmo scritto per essere eseguito su di essa.

4.2 Complessità di un Algoritmo

Uno dei risultati più signignicativi ottenuti da Alan Turing[3] riguarda la progettazione di un modello di calcolo universale, astratto, ma tale da poter essere considerato equivalente ad ogni altro modello di calcolo generale, la *macchina di Turing* (1936) [21], con cui diede i natali alla *teoria della calcolabilità e della complessità* [28, 29].
La complessità è un concetto indipendente dalla macchina. Considera solo una stima del numero di operazioni, dipendente dalla dimensione dell'input,

[3]Come già accennato nel Capitolo 2, Alan Turing è stato uno dei più grandi matematici del Novecento ed è considerato tra i padri del'informatica.

e non prende in considerazione limiti fisici delle macchine.

Le funzioni di complessità sono funzioni discrete, non continue, definite sui numeri naturali, positive e non decrescenti.
In generale, ci si occupa di studiare il comportamento dell'algoritmo, la sua complessità, nel caso peggiore, ossia per quelle istanze del problema la cui soluzione richiede il massimo numero di operazioni[4].

Complessità di tempo La complessità di tempo [44] di un algoritmo A che risolve un problema di dimensione N rappresenta la *quantità di lavoro dell'algoritmo* in termini di operazioni sia di calcolo sia di memoria svolte, e si misura in genere attraverso il numero delle operazioni elementari. Fissato l'algoritmo, è funzione della dimensione del problema.

Complessità di spazio La complessità di spazio [44] di un algoritmo A che risolve un problema di dimensione N rappresenta la *quantità di memotia utilizzata dall'algoritmo*, spesso escludendo input ed output, e si misura in genere in byte, o in parole. Fissato l'algoritmo, è funzione della dimensione del problema.

Sono metriche di solito associate al **tempo d'esecuzione** e alla velocità dei processori per valutare le caratteristiche e prevedere le prestazioni degli algoritmi.

4.3 Modello PRAM

Molto lavoro teorico sulla complessità del calcolo parallelo si è concentrato sul modello PRAM (Parallel Random Access Model) [4]. Si tratta di un modello teorico di computer a memoria condivisa; ne esistono diverse varianti, che si differenziano per i dettagli sul modo in cui si suppongono gestiti gli

[4]La valutazione di un algoritmo nel 'caso peggiore' non è banale, in quanto tale caso non è necessariamente noto a priori [22].

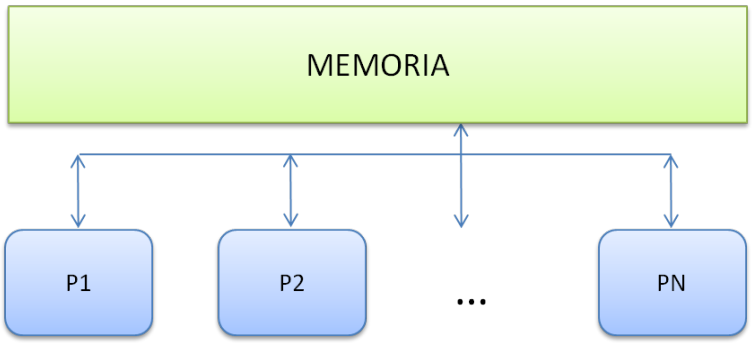


Figura 4.1: Modello PRAM.

accessi allo stesso indirizzo di memoria da parte di diversi threads o processi. Di solito il tempo di accesso viene considerato una costante, se non c'è conflitto: niente cache, niente diverse locazioni. Il valore di questo modello è nella possibilità di comprensione dei limiti degli algoritmi paralleli, ma rappresenta un'astrazione che non può di fatto essere implementata nella realtà.

PRAM è una generalizzazione del modello RAM, in cui c'è un singolo processore e un blocco di memoria illimitato, con tempo di accesso unitario a qualsiasi locazione di memoria [24, 25, 27]. Una macchina PRAM consiste in un numero arbitrario di processori ed arbitraria quantità di memoria in cui ogni processore ha tempo d'accesso alla memoria unitario a qualsiasi locazione. Cioè il calcolo prosegue secondo un unico clock, ed ogni processore esegue una sola operazione per clock, comprese quelle di lettura e scrittura (con cui si realizza anche la comunicazione tra i processi). Naturalmente questo pone delle complicazioni, per esempio in caso di tentativo di accesso in scrittura da parte di più processi contemporaneamente.
Ci sono diverse versioni del modello PRAM, a seconda di come si risolve questa questione. Alcune varianti sono [24, 25]:

EREW - Exclusive Read/Exclusive Write Un solo processore alla volta puà accedere ad un dato in scrittura o lettura, e sarà lo stesso programmatore a fare in modo che non accada diversamente, pena l'inaccettabilità del programma. É il tipo più semplice da implementare.

CREW - Concurrent Read/Exclusive Write Le letture possono essere in concorrenza, la scrittura no. É più difficile da implementare.

CRCW - Concurrent Read/Concurrent Write Sia le letture che le scritture possono essere concorrenti. Va quindi rifinito il modello con i dettagli su cosa accade quando più di un processore cerca di scrivere in una locazione di memoria. Le principali sottocategorie riguardo le scelte di gestione della concorrenza in scrittura sono [23]:

Common Scritture concorrenti sono ammesse solo se i processori coinvolti intendono scrivere lo stesso valore.

Arbitrary Viene scritto un valore arbitrario tra quelli che i processori tentano di scrivere.

Priority Il valore scritto è quello del processore con identificativo minore.

Combining Il valore scritto è una combinazione (in genere associativa e commutativa, come la somma o il 'massimo') dei valori che i processori coinvolti tentano di scrivere.

Una macchina PRAM sincrona di fatto lavora in modalità SIMD, ma si può definire anche una macchina PRAM asincrona in cui i processori abbiano anche una memoria locale privata, oltre a quella globale condivisa [26]. Le diverse varianti del modello viste non differiscono per quanto riguarda la potenza espressiva, ma solo dal punto di vista teorico del calcolo della complessità.

Per una macchina PRAM, comunque sia, si scrivono dei programmi, che chiamiamo *algoritmi PRAM*, che si prestano facilmente alla valutazione attraverso metriche di complessità appositamente definite quali:

- il **tempo** dell'algoritmo parallelo su un numero arbitrario di processori, $T(n)$, o per un numero preciso di processori p, $T_p(n)$ per un problema di dimensione n;

- il **lavoro**, definito come

$$W(n) = \sum_{i=1}^{T}(n)P(i),$$

dove $P(i)$ è i numero di processori attivi durante l'i-ma iterazione. Questa è una metrica importante che può essere usata per mettere in relazione $T(n)$ con $T_p(n)$, attraverso il Principio di Brent, espresso dal seguente

Teorema 4.1. *Data una macchina PRAM con p processori, ed un algoritmo A, se $T(n)$ è il suo tempo e $W(n)$ il suo lavoro, vale che*

$$T_p(n) \leq T(n) + \frac{W(n)}{p}.$$

- lo **speed up** dell'algoritmo parallelo che utilizza p processori rispetto al migliore algoritmo sequenziale che risolve lo stesso problema:

$$SP(n,p) = \frac{T_S(n)}{T_C(n,p)}.$$

Il valore ideale per lo speed up è $\Theta(p)$ [23].

Gli algoritmi costruiti su queste definizioni possono essere la base per la costruzione di programmi paralleli, sebbene abbandonare l'astrazione possa creare diverse difficoltà pratiche e far perdere parte del valore alla valutazione che si è fatta attraverso il modello. Questo perchè se già il modello RAM, rispetto soprattutto alle moderne architetture, con la sua ipotesi di tempo di accesso alla memoria unitario, rappresenta già un modello di costo inadeguato per gli algoritmi sequenziali, il modello PRAM lo è ancora di più per gli algoritmi paralleli, contenendo un maggior numero di componenti [23].

4.4 Valutazione di algoritmi paralleli

L'analisi che segue si inserisce nell'ambito delle valutazioni delle architetture reali[5] Quando non ci limitiamo ad un modello astratto, altri elementi prendono importanza, come le latenze di sincronizzazione e di comunicazione (che possiamo riassumere con la parola *overhead*), e sorge l'esigenza di individuare i limiti delle prestazioni degli algoritmi scritti per queste architetture.
Rivediamo innanzitutto a quale contesto di definizioni e risultati ormai classicamente riconosciuti ci stiamo riferendo.

Definizione 4.1 (Sistema Parallelo)**.** *Chiamiamo* ***sistema parallelo*** *[37] l'ambiente costituito da un algoritmo parallelo e dall'architettura su cui esso viene implementato.*

Definizione 4.2 (Dimensione del Problema)**.** *Intendiamo* ***dimensione di un problema*** *[37] una misura del numero di operazioni elementari che il più veloce algoritmo sequenziale disponibile compie per risolverlo. È una funzione della dimensione dell'input, ed è in pratica determinata dalla complessità di tempo del più veloce algoritmo sequenziale noto.*

Definizione 4.3. *Il* ***tempo d'esecuzione*** *su un processore di un algoritmo A, che chiamiamo $T_s(A, N)$, dove N è la dimensione del problema risolto, è sempre divisibile in due componenti [31]:*

- *T_{seq} tempo d'esecuzione della parte A_{seq} che deve essere eseguita in sequenziale,*
- *T_{par} tempo d'esecuzione (sequenziale) della parte A_{par} che può essere suddivisa in componenti parallele da eseguire su processori diversi.*

Allora

$$T_s(A, N) = T_{seq} + T_{par}.$$

Per il ragionamento seguente, normalizziamo ponendo unitario il tempo di esecuzione dell'algoritmo su un processore, quindi:

$$T_s(A, N) = T_{seq} + T_{par} = 1.$$

[5] secondo G. Fox [33].

Supponiamo fissati l'algoritmo A e la dimensione N. Se potessimo costruire architetture tali che il tempo d'esecuzione degli algoritmi raddoppiasse con il numero di processori, allora il limite alle prestazioni sarebbe dovuto esclusivamente al fatto che si ha un numero finito di processori a disposizione [31]. Tuttavia, anche supponendo di poter aumentare arbitrariamente il numero di processori, sappiamo che una prima ulteriore limitazione alle prestazioni, viene dallo stesso algoritmo parallelo: secondo la *legge di Amdahl*[6] [30] la parte sequenziale A_{seq} di un algoritmo A non è mai azzerabile, e di conseguenza T_{seq} costituisce comunque un limite inferiore per il tempo d'esecuzione di A. Nella pratica però, il guadagno, ovvero il miglioramento nelle prestazioni, ottenibile dall'esecuzione parallela è limitato anche da quelle quantità che abbiamo accennato prima, e che sono sempre difficili da individuare con precisione, dipendendo inoltre dalle peculiarità della singola architettura in esame.

Allora possiamo affermare che, se il sistema parallelo ha P processori ed il lavoro di A_{par} è equamente suddiviso tra le sue P componenti , il tempo d'esecuzione di A su tale sistema è

$$T_c(A, P, N) = T_{seq}(A, N) + \frac{T_{par}(A, N)}{P} + T_O(A, P, N) \tag{4.1}$$

dove $T_O(A, P, N)$ è l'**overhead totale**.

Elenchiamo alcune proprietà che assumiamo appartengano all'overhead[7] [31]:

1. $T_O(A, P, N)$ è funzione continua e due volte differenziabile rispetto a P,

[6]Secondo la legge di Amdahl il guadagno ottenuto da un algoritmo parallelo per P processori rispetto ad uno sequenziale che risolva lo stesso problema, ovvero lo speed up, è dato dalla relazione

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

dove α è la frazione delle operazioni di A che non può essere eseguita in parallelo. Nei termini che abbiamo usato, possiamo riscriverla come

$$S(A, P, N) = \frac{1}{T_{seq} + \frac{T_{par}}{P}}.$$

[7]Flatt and Kennedy [31], **Fundamental assumptions** a p. 3.

2. $T_O(A, 1, N) = 0$, per cui $T_c(A, 1, N) = T_s(A, N)$,
3. $T_O(A, P, N)$ è non-negativa crescente in P
4. $P \cdot T''_O(A, P, N) - 2 \cdot T'_O(A, P, N)$, cioè ogni zero della derivata è unico,
5. esiste un $P_1 \geq 1$ tale che $T_O(A, P_1, N) = T_s(A, N) = 1$, cioè T_O cresce in qualche intervallo.

Dalle prime tre proprietà si dimostra il lemma [31]:

Lemma 4.2. *Date le proprietà 1-3 dell'overhead, la proprietà 4 implica che, se la derivata del tempo concorrente, $T'_c(A, P, N)$ ha uno zero, questo è unico.*

Questo significa che la funzione del tempo d'esecuzione ha un solo punto di minimo o massimo. Quindi, considerando le prime quattro proprietà, vale il seguente [31]

Lemma 4.3. *Date le proprietà 1-4 dell'overhead, se esiste un valore di P, detto P_0 tale che*

$$T'_c(A, P_0, N) = 0$$

allora

$$T''_c(A, P_0, N) > 0$$

e T_c è monotona crescente per $P > P_0$.

Dunque, l'unico punto di cui abbiamo precedentemente assicurato l'esistenza, P_0, è un punto di minimo per $T_c(A, P, N)$.
Si può quindi provare [31] il:

Teorema 4.4. *Supponendo valide le proprietà 1-5, esiste un unico valore di P, che chiamiamo P_0, per il quale*

$$T_c(A, P_0, N) \leq T_c(A, P, N) \qquad \forall P \geq 1.$$

Se $T'_O(A, 1, N) < T_{par}$, P_0 è soluzione di

$$P_0^2 \cdot T'_O(A, P_0, N) = T_{par},$$

altrimenti $P_0 = 1$.

A questo punto possiamo dire che, con particolari ipotesi sull'overhead e sulla sua derivata, la funzione T_c è decrescente tra 1 e P_0 che risulta punto di minimo, ovvero il numero di processori con cui si avrà il tempo d'esecuzione minimo. Quando la derivata dell'overhead in 1 è maggiore del tempo T_{par}, il tempo minimo si ha per un solo processore, e l'aggiunta di altri non comporta miglioramenti.
Definiamo il costo di un'esecuzione parallela [31].

Definizione 4.4 (Funzione Costo)**.** *La funzione* ***costo del sistema*** *è il prodotto del tempo d'esecuzione parallelo e il numero di processori:*

$$Q(A,P,N) = P \cdot T_c(A,P,N) = P \cdot T_{seq}(A,N) + T_{par}(A,N) + P \cdot T_O(A,P,N). \tag{4.2}$$

La funzione Q è chiaramente proporzionale al numero di processori usati. Siccome il costo della versione sequenziale è unitario, $P \cdot T_c(A,P,N)$ è il rapporto tra il costo dell'esecuzione parallela e quella sequenziale [31].

Teorema 4.5. *Date le proprietà dell'overhead* $T_O(A,P,N)$*, risulta che:*

- $Q(A,1,N) = 1$*,*
- $Q(A,P,N)$ *è monotona crescente con* P*,*
- $Q'(A,P,N)$ *è monotona crescente con* P*, dunque la funzione costo ha una forma convessa.*

Segue il corollario [31]:

Corollario 4.6.

$$Q(A,P,N) > 1 \qquad \forall P > 1.$$

Intuitivamente, quanto più vicina a 1 è Q tanto più economico (e quindi efficiente) risulta il calcolo.
Mettiamo in relazione il costo con il punto di minimo di T_c.

Teorema 4.7. *Se $T'_O(A,1,N) < T_{par}$, cioè il tempo d'esecuzione T_c ha punto di minimo in P_0, la linea tangente a Q in P_0, che indichiamo con $l(P_0,x)$, in 0 ha valore nullo (cioè passa per l'origine del sistema di riferimento), ovvero*

$$l(P_0,0) = 0.$$

L'esecuzione parallela in pratica comporta lo spreco di qualche risorsa per diminuire il tempo d'esecuzione.
Per misurare lo spreco di risorse, introduciamo la seguente definizione [31]:

Definizione 4.5 (Efficienza)**.** *La funzione **efficienza**, misura lo sfruttamento del sistema e si definisce come*

$$E(A,P,N) = \frac{T_s(A,N)}{Q(A,P,N)} = \frac{T_s(A,N}{P \cdot T_c(A,P,N)} \leq 1 \qquad \forall P > 1.$$

Riguardo l'efficienza, dato il teorema 4.5, vale il seguente:

Corollario 4.8. *Per l'efficienza valgono le seguenti proprietà:*

- $E(A,1,N) = 1$,
- $E(A,P,N)$ *è monotona decrescente con* P,
- $E'(A,P,N)$ *è monotona crescente con* P, *quindi ha una forma convessa.*

Quindi, diversamente dal costo, l'efficienza decresce a partire da 1, che risulta il suo valore massimo: non si ha nessuno spreco di risorsa solo utilizzando un singolo processore.
Questo è un comportamento di cui tenere certamente conto. Ovviamente è conveniente che E sia vicina all'unità per P più grande possibile.
Poichè ci interessa prevedere l'efficienza al diminuire del tempo d'esecuzione, enunciamo il seguente teorema [31]:

Teorema 4.9. *Supponendo valide le proprietà 1-5 dell'overhead, se $T'_O(A,1,N) < T_{par}$, cioè il tempo d'esecuzione T_c ha punto di minimo in P_0, vale che*

1. *se* $T'_0(A,P,N) = k \cdot P - P$

$$E(A,P_0,N) \leq \frac{1}{2 + (P_0 - 2) \cdot T_{seq} - \frac{T_{par}}{P_0})},$$

2. *se* $(d/dP)(\frac{T_0(A,P,N)}{P}) \leq 0 \qquad perP = P_0$

$$E(A,P,N) \leq \frac{1}{2 - (P_0 - 2)T_{seq}}.$$

Questo teorema implica che anche immaginando che l'overhead abbia un andamento meno che lineare in corrispondenza di $P_0 > 2$ punto di minimo per il tempo d'esecuzione l'efficienza sarà minore di $\frac{1}{2}$. Quello che appare già è che, se il crescere dei processori fa diminuire il tempo d'esecuzione, fa anche crescere il costo e quindi diminuire l'efficienza.
Introduciamo a questo punto un'altra definizione:

Definizione 4.6 (Speed Up)**.** *Lo* ***speed up****, misura il guadagno ottenuto rispetto all'algoritmo sequenziale e si definisce come:*

$$S(A,P,N) = \frac{T_s(A,N)}{T_c(A,P,N)} = P \cdot E(A,P,N) = \frac{P}{Q(A,P,N)} \leq P = S^I(A,P,N)$$

dove $S^I(A,P,N)$ *è lo* ***speed up ideale****.*

La principale proprietà dello speed up è data dal seguente:

Teorema 4.10. *Assumendo le proprietà 1-5 dell'overhead,* $S(A,P,N)$ *ha un unico massimo, per* $P_0 \geq 1$*. Se* $T'_O(A,1,N) < T_{par}$*, cioè il tempo d'esecuzione* T_c *ha punto di minimo in* P_0*,*

$$S(A,P_0,N) = \frac{1}{Q'(A,P_0,N)}.$$

Osservazione 1. Questo significa che possiamo trovare un numero di processori che ci dia il massimo guadagno. Invece, il limite superiore per lo speed up secondo Amdhal, $1/T_{seq}$, che non tiene conto del ruolo dell'overhead, non è raggiungibile con un numero finito di processori.

Diamo ancora la seguente definizione:

Definizione 4.7 (Operating Point)**.** ***Operating point*** *il numero P_{op} di processori che massimizza lo sfruttamento del sistema da parte del nostro algoritmo.*

Per quanto detto finora, scegliere $P_{op} = P_0$ non è affatto detto sia la migliore possibilità, visto il degrado dell'efficienza. Insomma, né lo speed up, né l'efficienza, da soli, sembrano bastare ad individuare l'operating point. Quello che serve è un compromesso tra le due quantità, tra la riduzione del tempo d'esecuzione e lo spreco della risorsa: occorre un **criterio di ottimalità**.
L'operating point proposto da Kuck [32] nel 1976 ed ancora utilizzato spesso per alcune classi di architetture, è P_{op} tale che il prodotto:

$$E(A, P, N) \cdot S(A, P, N) = \frac{S(A, P, N)}{Q(A, P, N)}$$

sia minimo: per $P < P_{op}$ lo speed up prevale sul costo, per $P > P_{op}$ accade l'inverso.

Noti questi limiti, pesano molto meno se quello che vogliamo da una macchina parallela è che possa risolvere problemi sempre più grandi a parità di tempo, come aveva già osservato Fox [33], qualche anno prima delle formulazioni di Kennedy e Flatt, e come avevano espresso Gustafson et al. [35, 36]. Introduciamo allora un'altra definizione.

Definizione 4.8 (Speed Up Scalato)**.** *Sia $k \geq 1$, $P \geq 1$ un numero di processori, ed N la dimensione di un problema risolvibile con l'algoritmo A. Lo* ***speed up scalato*** *dell'algoritmo A si definisce come:*

$$SS(k, A, P, N) = k \cdot \frac{T_c(A, P, N)}{T_c(A, kP, kN)}.$$

Uno speed up scalato vicino a k significa che l'algoritmo *scala bene* con il numero di processori.
Poniamo le seguenti **assunzioni di scalabilità**:

S1 $T_O(A, P, N) = T_O(A, P)$, cioé l'overhead non dipende dalla dimensione del problema,

S2 $T_{seq}(A, kN) = T_{seq}(A, N)$, cioé il tempo d'esecuzione della parte sequenziale di A è lo stesso per ogni dimensione.

Kennedy e Flatt [31] ne ricavano alcuni risultati tra cui il seguente teorema:

Teorema 4.11. *Sotto le assunzioni di scalabilità S1 ed S2, fissati N e P,*

$$SS(k, A, P, N) = O(\frac{k}{T_O(kP)}).$$

Osserviamo cosa accade allo speed up originalmente definito se facciamo crescere la dimensione del problema lasciando invariato il numero di processori. Sarà

$$S(A, P, kN) = \frac{T_s(A, kN)}{T_c(A, P, kN)}.$$

E, sostituendo la seconda assunzione con una più debole:

S2' $lim_{k\to\infty} \frac{T_{seq}(A,N)}{N} = 0$,

vale anche il seguente:

Teorema 4.12. *Sotto le assunzioni di scalabilità S1 ed S2', fissati N e P, al crescere di k*

$$lim_{k\to\infty} S(A, P, kN) = P.$$

Questo significa che, dato un certo numero di processori, ci si avvicina allo speed up ideale tanto quanto cresce la dimensione del problema.

Quando A da semplice algoritmo deve diventare software, ovvero quando lo caliamo su un'architettura concreta, oltre che composto di operazioni di calcolo, che impegneranno il processore direttamente, dovrà essere associato anche ad operazioni di accesso alla memoria che costringono il processore a restare inattivo in attesa della lettura o della scrittura dei dati necessari attraverso la gerarchia di memoria. Queste operazioni contribuiscono al tempo di esecuzione almeno quanto quelle di calcolo. In una valutazione realistica

delle prestazioni di A diventa allora importante anche la *latenza di memoria* (rif. capitolo 3).

Definizione 4.9. *Il tempo d'esecuzione di un algoritmo A sequenziale è sempre esprimibile come*

$$T_s(A,N) = T_{s[flop]}(A,N) + T_{s[mem]}(A,N)$$

dove

- $T_{s[flop]}$*, misura il tempo speso in operazioni aritmetiche floating point, comprendendo la relativa latenza,*
- $T_{s[mem]}$*, misura il tempo speso in accessi alla (gerarchia di) memoria, comprendendo anche le relative latenze.*

Nella condizione ideale in cui la latenza di memoria non pesa, allora è

$$T_{s[flop]}(A,N) + T_{s[mem]}(A,N) = T_{s[flop]}(A,N). \tag{4.3}$$

Cioè

$$\frac{T_{s[flop]}(A,N) + T_{s[mem]}(A,N)}{T_{s[flop]}(A,N)} = 1. \tag{4.4}$$

Allora per migliorare le prestazioni di un software bisogna innanzitutto ridurre il peso degli accessi alla memoria, e avvicinare il rapporto nell'equazione 4.4 a 1. Poichè

$$\frac{T_{s[flop]}(A,N) + T_{s[mem]}(A,N)}{T_{s[flop]}(A,N)} == \frac{T_{s[flop]}(A,N)}{T_{s[flop]}(A,N)} + \frac{T_{s[mem]}(A,N)}{T_{s[flop]}(A,N)} = 1 + \frac{T_{s[mem]}(A,N)}{T_{s[flop]}(A,N)} \tag{4.5}$$

allora l'intento sarà quello di portare il rapporto

$$\frac{T_{s[mem]}(A,N)}{T_{s[flop]}(A,N)} \tag{4.6}$$

quanto più vicino possibile a 0. Osserviamo ora che

$$T_{s[flop]}(A,N) = \#operazioni\_floating\_point \times t_{flop} \tag{4.7}$$

e

$$T_{s[mem]}(A, N) = \#accessi\_memoria \times t_{mem} \tag{4.8}$$

dove t_{flop} è il tempo medio di una operazione floating point e t_{mem} è il tempo medio di acceesso a una locazione di memoria, ed entrambe dipendono dall'architettura.
A seconda dell'architettura quindi questi tempi possono essere più o meno alti, e in ogni caso non è possibile ridurli a zero, nè ridurli all'infinito.

Anche in caso di un algoritmo parallelo associato ad una macchina concreta vale che:

Definizione 4.10. *Il tempo d'esecuzione concorrente di un algoritmo A parallelo è sempre esprimibile come*

$$T_c(A, P, N) = T_{c[flop]}(A, P, N) + T_{c[mem]}(A, P, N) + T_O(A, P, N)$$

dove

- $T_{c[flop]}$*, misura il tempo speso in operazioni aritmetiche floating point, comprendendo la relativa latenza,*
- $T_{c[mem]}$*, misura il tempo speso in accessi alla (gerarchia di) memoria, comprendendo anche le relative latenze,*
- T_O*, misura l'overhead totale.*

Per $T_c(A, P, N)$ valgono relazioni analoghe a quelle da 4.3 a 4.8.
Richiamiamo ora:

Definizione 4.11. *Chiamiamo* ***grado di concorrenza*** *di un algoritmo A che risolve un problema di dimensione N il massimo numero di componenti in cui la sua parte parallelizzabile A_{par} può essere divisa perché vengano eseguite contemporaneamente. Denotiamo il grado di concorrenza con $C(A, N)$, ed è indipendente dall'architettura del sistema.*

Data N, non si possono usare più di $C(A, N)$ processori per eseguire A una volta parallelizzato [33, 37]. Quando $T_O(A, P, N)$ cresce meno velocemente di

P (cioé non vale la proprietà 3 dell'overhead definita precedentemente) o cresce linearmente rispetto P, nell'espressione 4.1 la componente dell'overhead pesa meno di $\frac{T_{par}(A,N)}{P}$, e l'operating point è dato dal grado di concorrenza dell'algoritmo stesso, oltre il quale non ha senso far crescere il numero di processori, cioé

$$P_{op} = C(A, N)$$

e conviene utilizzare tanti processori quante sono le componenti in cui è divisibile il lavoro di A_{par}, se sono a disposizione: il rapporto $\frac{T_{par}(A,N)}{P}$ è così ridotto al minimo e l'overhead non sarà comunque cresciuto abbastanza da costituire un limite importante. È per esempio il caso di algoritmi per alcune macchine SIMD [37].
Quando invece $T_O(A, P, N)$ cresce invece molto velocemente rispetto a P (cioè vale la proprietà 3), la componente dell'overhead influisce pesantemente nell'equazione 4.1 e l'operating point può essere minore del grado di concorrenza. Questo perchè facendo crescere il numero di processori l'overhead può diventare tanto alto da annullare il beneficio ottenuto dalla divisione del lavoro, quindi può non aver senso dividere la parte parallelizzabile dell'algoritmo in tutte le componenti possibili, quindi

$$P_{op} \leq C(A, N).$$

Di conseguenza, sebbene il grado di concorrenza sia dipendente esclusivamente dall'algoritmo, il criterio di ottimalità per individuare l'operating point è strettamente legato all'architettura del sistema e definirlo può richiederne profonda conoscenza.

Capitolo 5

Prestazioni sulle GPU

L'intenzione di questo capitolo è quella di costruire un modello di valutazione per gli algoritmi su GPU, che sia riconducibile al modello di valutazione classico e permetta una stima reale delle performance, ma soprattutto induca la definizione di un criterio di ottimalità per l'individuazione dell'*operation point* che sia caratteristico dell'architettura.

5.1 Graphics Processing Units

5.1.1 Dalla grafica alle GPGPU

Uno dei motivi principali per l'aggiunta di istruzioni SIMD alle architetture è che la maggior parte dei microprocessori è collegata ad un terminale grafico di I/O, per cui una frazione sempre più grande del tempo di elaborazione è utilizzata per gestire la grafica. Quindi, poiché il numero di transistor disponibili su un processore ha continuato a crescere in linea con la legge di Moore, sembra sensato migliorare l'elaborazione grafica. Risulta sensato anche aggiungere altre funzionalità ai chip contenenti i controllori delle operazioni di grafica e video, con lo scopo di rendere più veloce la grafica 2D e 3D.
L'industria dei videogiochi ha rappresentato una delle maggiori forze propulsive per il miglioramento dell'elaborazione grafica. La rapida crescita del mercato ha incoraggiato molte società a investire ingenti risorse per sviluppare hardware grafico sempre più veloce, e questo meccanismo ha fatto sì

che nel 2000 le architetture per l'elaborazione grafica avessero guadagnato autonomia e cominciassero a svilupparsi ad una velocità maggiore rispetto ai microprocessori tradizionali.
Poiché l'industria della grafica ha obiettivi diversi da quelli delle aziende che sviluppano i microprocessori, in quell'ambito sono nati uno stile di elaborazione ed una terminologia differenti. Innanzitutto i neonati processori grafici presero il nome di **graphics processing unit** (**GPU**), che li distingue con chiarezza dalle CPU [11].

Pipeline Grafica L'idea è che il processo di **rendering** (generazione di un'immagine a partire da una descrizione di oggetti 3D) passi attraverso la cosiddetta **pipeline grafica logica**, che poi andrà realizzata attraverso un hardware più o meno specializzato. Dunque l'applicazione, che è stata lanciata sulla CPU, invia alla GPU una sequenza di vertici raggruppati in primitive geometriche quali punti, linee, triangoli e poligoni. L'**assemblatore di input** raccoglie i vertici e le primitive e il **programma di shading dei vertici** esegue l'elaborazione vertice per vertice, mappa la sua posizione, espressa in virgola mobile, sullo schermo, eventualmente modificando il colore e l'orientamento. Il **programma di shading della geometria** esegue elaborazioni sulle primitive geometriche e può aggiungere o eliminare primitive. L'**unità di impostazione e di rasterizzazione** genera dei *frammenti di pixel* (potenziali contributi ai pixel) che vengono ricoperti da una primitiva geometrica. Il **programma di shading dei pixel** effettua elaborazioni sui singoli frammenti, tra cui l'interpolazione dei parametri di ognuno e l'applicazione della *tessitura* e del colore. Gli shader di pixel fanno largo uso delle funzioni di mappatura (campionamento e filtraggio) su matrici 1D, 2D e 3D, chiamate appunto tessuture, soprattutto in caso di mappe, funzioni, decalcomanie, immagini e dati. Infine, lo stadio di **elaborazione delle operazioni di rasterizzazione**, ovvero di assemblaggio dell'output, effettua poi il controllo della profondità e il controllo degli stencil, e può effettuare un'operazione di mescolamento dei colori del pixel e del frammento.

Le GPU, in quanto coprocessori ad integrazione della CPU, non hanno biso-

gno di eseguire tutti i compiti che sono eseguiti normalmente da una CPU. Questo loro diverso ruolo gli consentiva di dedicare tutte le risorse alla grafica. E' infatti accettabile che una GPU esegua alcuni compiti in maniera non efficiente o che non li esegua affatto, poiché in un sistema che contiene sia una GPU che una CPU sarà la CPU ad eseguire questi compiti; l'accoppiamento delle due unità di elaborazione costituisce un esempio di **multiprocessing eterogeneo**, in cui non tutti i processori sono identici (multiprocessore asimmetrico, concetto già utilizzato per il CELL, anch'esso progettato per accelerare la grafica 2D e 3D).
Col tempo però sono diventate sempre più programmabili e precise nei calcoli. Sono nate interfacce di programmazione come le API (Application Programming Interface) OpenGL o DirectX, e i linguaggi di programmazione che si stanno evolvendo rapidamente. I dispositivi più moderni hanno altissime potenzialità, sono multiprocessori altamente paralleli e multithread, che nascondono l'elevata latenza di memoria sfruttando il parallelismo tra molti thread, per cui la memoria principale viene ottimizzata per massimizzare la larghezza di banda del trasferimento invece che per minimizzare la latenza. Vengono ormai progettate utilizzando processori di uso generico tra loro identici, rendendo così le GPU più simili alle architetture multicore dei microprocessori convenzionali. Sebbene tradizionalmente eseguano istruzioni SIMD, gli ultimi prodotti sono basati su istruzioni scalari per migliorarne la programmabilità. Dal 2008 poi, supportano anche l'aritmetica in doppia precisione, sebbene queste operazioni restino sensibilmente più lente di quelle in singola.

Allora, questi dispositivi, inizialmente progettati per un numero ristretto di applicazioni, sono oggi diventate *General Purpose GPUs* (GPGPU), sono cioè tali da poter essere utilizzate per applicazioni generiche. Questo ha spinto lo sviluppo di linguaggi, per lo più ispirati al C, che permettessero di scrivere programmi direttamente per la GPU.

La CPU, che attualmente è multicore (con un numero sempre maggiore di core) resta molto meno parallela ma mantiene il controllo sulla GPU che le

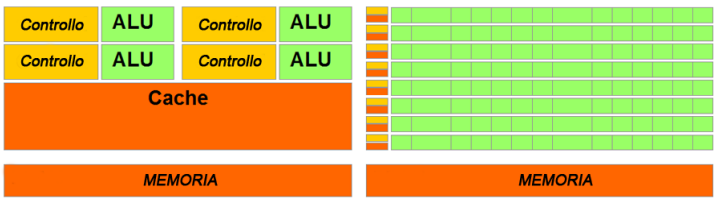


Figura 5.1: Lo schema di un microprocessore con due core a confronto con lo schema di una GPU.

viene affiancata.

5.1.2 Architetture sul mercato

Dal punto di vista funzionale, a partire dalla loro comparsa, le GPU si sono evolute da controllori VGA cablati, di limitate capacità, ad acceleratori delle funzioni grafiche che liberavano la CPU di un po' di lavoro, a processori paralleli programmabili. Tutto ciò è avvenuto modificando la pipeline grafica logica con l'aggiunta di elementi programmabili e despecializzando gli stadi della pipeline hardware sottostante. Alla fine è risultato conveniente unire i diversi elementi della pipeline programmabile in un'unica schiera di svariati processori programmabili. Nella generazione di GPU NVIDIA GeForce serie 8 l'elaborazione della geometria, dei vertici e dei pixel veniva effettuata dallo stesso tipo di core: questo permetteva estrema scalabilità ed un migliore bilanciamento del carico, aumentando le prestazioni globali.

GPU AMD Nel 2006 l'AMD, che ha comprato l'ATI Technologies, con la prima generazione di schede FireStream introdusse per le proprie GPU l'interfaccia hardware CTM (*Close To Metal*) per permettere ai programmatori di accedere all'insieme nativo di istruzioni e alla memoria, a basso livello. Questo aumentava certamente le potenzialità del prodotto, tuttavia la difficoltà di approccio con questa modalità di programmazione ne limitava il successo. Così nel 2007, con la seconda generazione delle FireStream, le FireStream 9170 ha introdotto nella realtà del GPGPU l'aritmetica in doppia precisione e ha distribuito la ATI Stream SDK che aggiungeva un nuovo linguaggio di alto livello, il Brook+ [5], da un progetto della Stanford Univer-

Figura 5.2: AMD FireStream 9350, sul mercato da metà del 2010.

sity. CTM si era anche evoluta in ATI CAL (Compute Abstraction Layer).

GPU NVIDIA Negli stessi anni NVIDIA cominciò a produrre la serie 8 della GPU GeForce con cui introdusse **CUDA** (Compute Unified Device Architecture), una particolare architettura di elaborazione in parallelo, che è poi diventata parte integrante delle GPU GeForce, ION, Quadro e Tesla [14]. Con questa architettura nascono anche estensioni per i principali linguaggi di programmazione, soprattutto per C, che individuano un modello di programmazione parallela scalabile a memoria condivisa. Attraverso CUDA/C lo sviluppatore può servirsi di alcune virtualizzazioni dell'architettura che semplificano molto la programmabilità della GPU. Gli ultimi prodotti NVIDIA supportano anch'essi la doppia precisione, in modo che possono essere utilizzate più efficacemente anche per il calcolo scientifico.

Entrambi i tipi di architettura, in ogni caso, si possono intendere come *architettura a shader unificati*, ovvero composte da una serie di multiprocessori programmabili che appunto unificano le operazioni sui vertici, sulla geometria e sui pixel, oltre a poter essere utilizzabili per tutt'altro tipo di calcolo, intuitivamente adatti soprattutto ai casi di alto parallelismo spaziale, ma non solo.
Nel 2008, AMD ed NVIDIA, insieme ad altre industrie del settore, hanno

Figura 5.3: NVIDIA Tesla M2090, architettura codice Fermi, sul mercato dal 2011.

investito sulla realizzazione di **OpenCL**, linguaggio che segue la filosofia e i principi di CUDA/C ma è concepito come opensource, che dal 2009 è supportato dai prodotti di entrambe le case, pur essendo ancora in evoluzione.

5.1.3 CUDA e la tassonomia di Flynn

Approfondire CUDA e CUDA/C, a cui è ispirato anche OpenCL, è un buon modo per approcciare lo studio delle GPU. In particolare, l'architettura NVIDIA G80 ha segnato una tendenza nelle scelte progettuali, non radicalmente diverse da quelle fatte da AMD, che cominciano a caratterizzare questo tipo di dispositivi. Inoltre il modello CUDA è applicabile anche ad altre architetture di elaborazione parallela a memoria condivisa, come le CPU multicore. Le astrazioni introdotte da CUDA rispecchiano esattamente le caratteristiche dell'hardware delle GPU NVIDIA, che riuniscono più livelli di parallelismo: tali astrazioni guidano il programmatore nella suddivisione del problema e nella sincronizzazione del lavoro. Il modello scala in modo trasparente su un numero molto elevato di processori, per cui il programma compilato potrà essere eseguito su un numero di core decidibile a runtime. Nei capitoli avanti avremo modo di approfondirne le caratteristiche.

Tuttavia, sorge il problema di come inquadrarle, dettagliatamente, nella tassonomia di Flynn per proseguire nella strutturazione di un modello che ci

permetta di comprenderne le performance [11].
Prendiamo in considerazione l'architettura del tipo NVIDIA Tesla GT200 (figura 5.4). E' ormai piuttosto diffusa, soprattutto in ambito scientifico, contiene tutti i principi della G80 e pone le basi per la successiva Tesla Fermi. Vediamo subito che sono costruite in modo che i programmatori possano gestirle come delle MIMD a memoria condivisa: comprendono un certo numero di multiprocessori, chiamati SM (*Streaming Multiprocessor*), collegati alla stessa memoria. Ogni SM è però multithread: al suo interno comprende diversi SP (*Streaming Processor*) a cui è affidata l'esecuzione dei thread. Ogni thread ha il suo program counter, quindi non si potrebbe definire l'esecuzione propriamente SIMD, eppure le prestazioni volute si possono ottenere solo se le istruzioni sono tali da consentire l'esecuzione del programma in stile SIMD: per questo ci si riferisce in genere ad una programmabilità SIMT (Single Instruction Multiple Treads). Inoltre un'architettura di questo tipo si differenzia dai processori vettoriali perché non c'è un compilatore che riconosce il parallelismo spaziale e trasforma le operazioni, tutto è affidato alla divisione tra i thread, per cui resta molto semplificata anche la gestione del lavoro che non prevede parallelismo spaziale.

5.2 Architettura di una GPU

Definiamo ora una GPU come un'architettura a più livelli [42] di parallelismo:

- MIMD shared-memory a livello più alto, in cui ci sono un certo numero P di *core* che condividono una memoria (con eventuale cache) (vedi figura 5.8),
- SIMD ad un livello intermedio, cioè all'interno di ogni core, dove ci sono Q unità calcolanti, che interpretiamo come delle ALU (vedi figura 5.7),
- ogni ALU è pipelined, come si conviene ai calcolatori moderni, ed r è la *profondità della pipeline* (vedi figura 5.6).

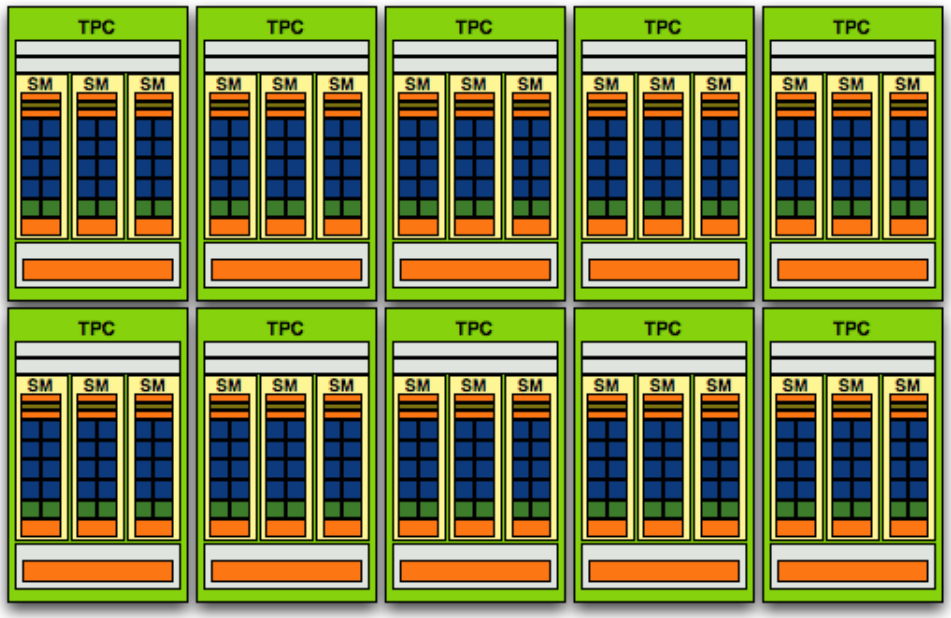


Figura 5.4: Esempio di schema a blocchi di una GPU NVIDIA Tesla.

Ricordiamo di aver accennato nei precedenti capitoli alle architetture vettoriali: in un processore vettoriale, ogni istruzione decodificata viene eseguita (sfruttando massicciamente tecniche di pipelining) su un *vettore* di dati, opportunamente indirizzato, in maniera automatica, cioè al processore corrisponde una dimensione fissa, che viene chiamata *lunghezza vettoriale* (VL), e che stabilisce il numero di dati su cui verrà eseguita ogni istruzione.
Di fatto, dal punto di vista logico, i core dell'architettura GPU che stiamo descrivendo, ognuno con la sua memoria locale, possono essere visti come dei processori vettoriali, in cui non è fissata la VL , ma per i quali è definita comunque una VL massima [45] (vedi fig. 5.5): la VL sarà data di volta in volta dal numero di flussi d'esecuzione lanciati contemporaneamente sul core.
Ci riferiamo ai core della GPU come **Multi-Processori (MP)**. Se la CU (Control Unit) dell'MP impiega

k cicli per la decodifica di un'istruzione,

ma nella pipeline di ogni ALU può entrare

un'operazione per ciclo,

allora per ogni istruzione decodificata, possono partire

k operazioni.

Per ogni istruzione quindi sull'MP partono

$k \cdot Q$ operazioni.

In un calcolatore di tipo SIMD, ci si riferisce al numero di esecuzioni fatte *simultaneamente* in corrispondenza della singola istruzione decodificata come alla sua **SIMD-width**. Sfruttando ancora l'idea del parallelismo di tipo SIMD che caratterizza il singolo MP, possiamo dire che esso avrà una propria *SIMD-width logica*, cioè un parametro caratterizzante che indica quante istruzioni partono su di esso per istruzione decodificata: la chiamiamo $dimW$. Allora

$$k \cdot Q = dimW.$$

Se ad ognuna di queste operazioni associamo un thread, rispettando un modello funzionale SIMT come introdotto all'inizio del capitolo (sezione 5.1.3), allora diamo la seguente definizione:

Definizione 5.1 (Warp)**.** *Chiamiamo* ***warp*** *il gruppo di $dimW$ thread che sarà attivo sull'MP ad ogni decodifica di istruzione. Di fatto il warp rappresenta l'*unità d'esecuzione *degli MP, ovvero non è possibile eseguire su un MP un numero minore di $dimW$ thread alla volta. Se l'algoritmo eseguito prevede la divisione del lavoro in $q < dimW$ thread, ne verranno creati altri $dimW - q$ che eseguiranno del lavoro inutile.*

Complessivamente sulla GPU verranno avviate nello stesso istante

$P \cdot dimW$ operazioni,

distribuite su altrettanti thread.

Dal punto di vista della memoria, l'architettura comprende:

- una memoria globale, la cui latenza di accesso è l_g (in cicli di clock),

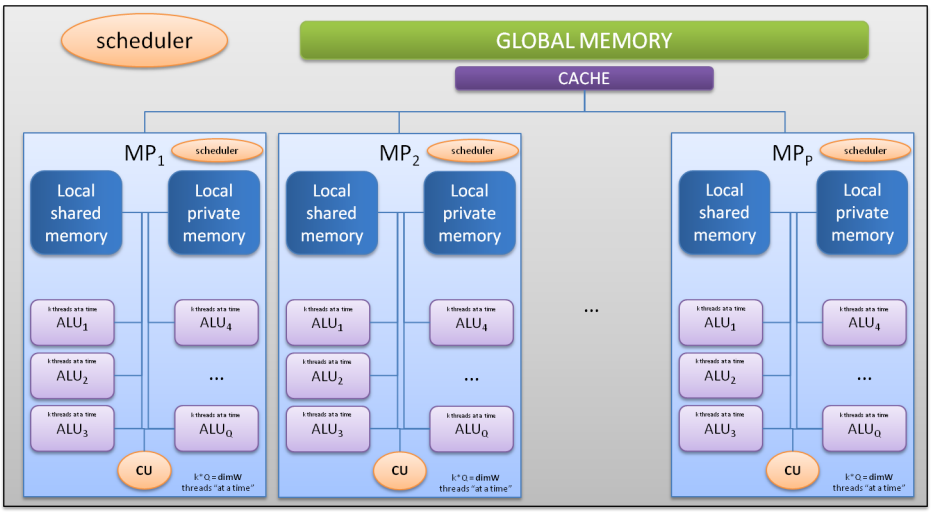


Figura 5.5: Schema funzionale di un'architettura GPU-based.

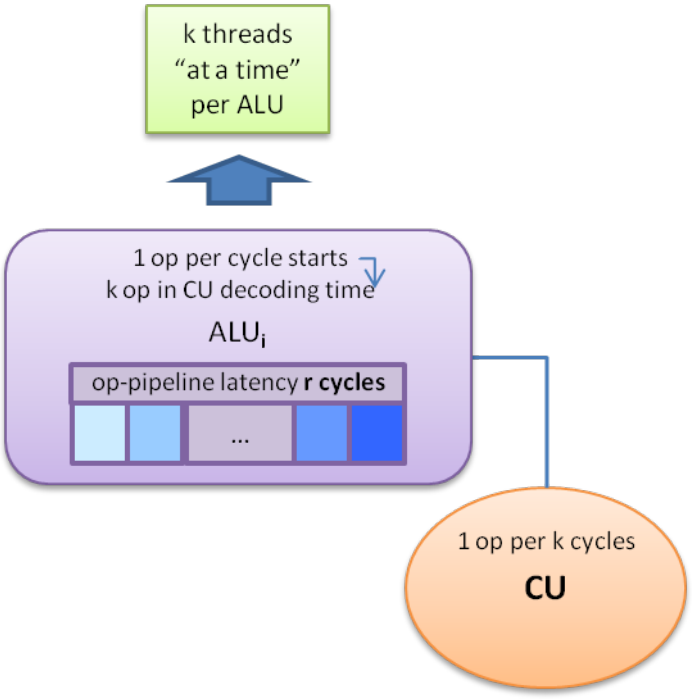


Figura 5.6: III livello di parallelismo: Schema funzionale di una delle ALU dell'architettura.

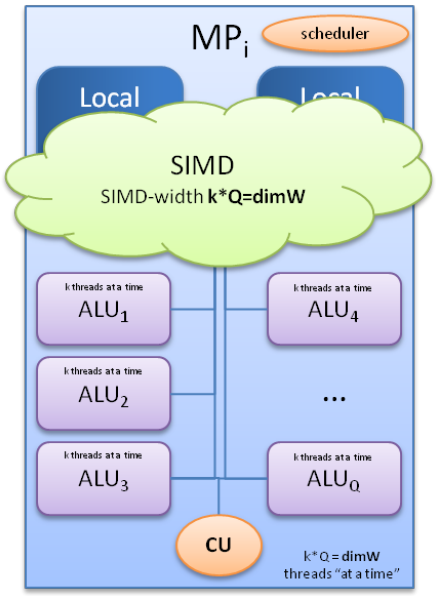


Figura 5.7: II livello di parallelismo: Struttura SIMD di un MP.

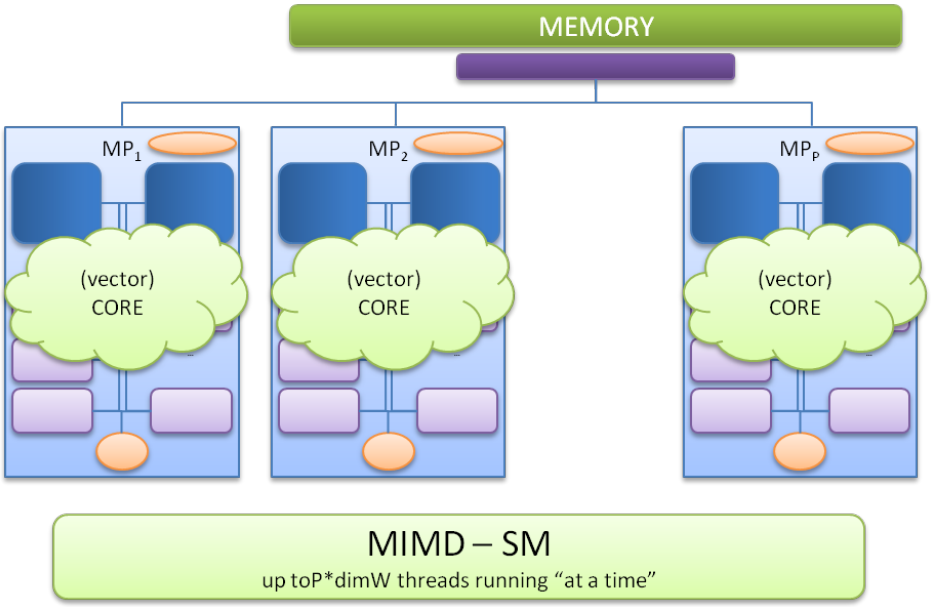


Figura 5.8: I livello di parallelismo: MIMD - Shared Memory.

- una memoria locale ad ogni MP, divisa in
  - una memoria destinata alla condivisione, che chiamiamo **shared memory**, la cui latenza di accesso è l_s (in cicli di clock),
  - una memoria destinata a contenere variabili private per i diversi flussi d'esecuzione, che chiamiamo **registri**[1]), la cui latenza di accesso è l_r (in cicli di clock),
- $l_r \leq l_s \leq l_g$.

Posto che l'algoritmo A sia stato progettato per essere eseguito su un sistema ibrido CPU/GPU, la strategia di parallelizzazione della parte parallelizzabile A_{Par} (vedi sezione 4.4) dovrà tenere conto dei diversi livelli di parallelismo, del coprocessore su cui saranno eseguite. La parte A_{seq}, invece, si suppone eseguita sulla CPU.
Nella pratica lo stesso lavoro di A_{Par} viene organizzato in più livelli. Per questo introduciamo la seguente definizione.

Definizione 5.2 (Blocco)**.** *Chiamiamo **blocco** un gruppo di thread che sarà interamente da un singolo **MP** della GPU.*

Quindi

- l'esecuzione dell'algoritmo sarà affidata a p *blocchi* divisi tra i P MP della GPU (vedi figura 5.9),
- ogni blocco sarà composto a sua volta da q *thread*,
- i thread del blocco saranno eseguiti un *warp* alla volta sulle Q ALU di ogni MP (vedi ficuta 5.10),
- Il tempo d'esecuzione sarà funzione sia del numero di blocchi che del numero di thread per blocco, cioè sarà

$$T_c(A_{par}, p, q, N).$$

[1] *Registro* è il nome che diamo all'area di memoria più vicina all'unità calcolante, caratterizzata dal più veloce tempo di accesso.

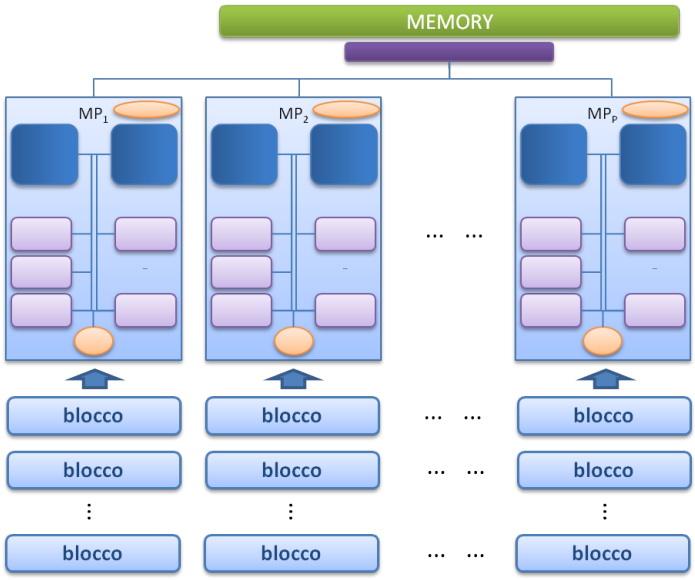


Figura 5.9: I diversi blocchi creati vengono distribuiti tra gli MP.

Facciamo alcune altre assunzioni sull'esecuzione di un algoritmo di questo tipo sull'architettura:

1. sulla GPU può essere eseguito un solo algoritmo alla volta,
2. tutti i thread di qualsiasi blocco possono accedere alla memoria globale,
3. ogni blocco schedulato su un MP ha un'area della shared memory dedicata, che sarà condivisa tra tutti i suoi thread,
4. ogni thread schedulato su un MP ha una porzione dei registri ad esso dedicata,
5. i warp lavorano concorrentemente tra loro rispetto alle diverse risorse dell'MP e della GPU, quindi anche rispetto agli accessi in memoria.

L'alto grado di concorrenza che un'architettura di questo tipo permette, quando ben sfruttato, permettere di sovrapporre buona parte delle latenze (aritmetiche e di memoria) e dell'overhead, facendo in modo che esse, nel tempo d'esecuzione complessivo, risultino molto meno pesanti, ed eventualmente trascurabili.

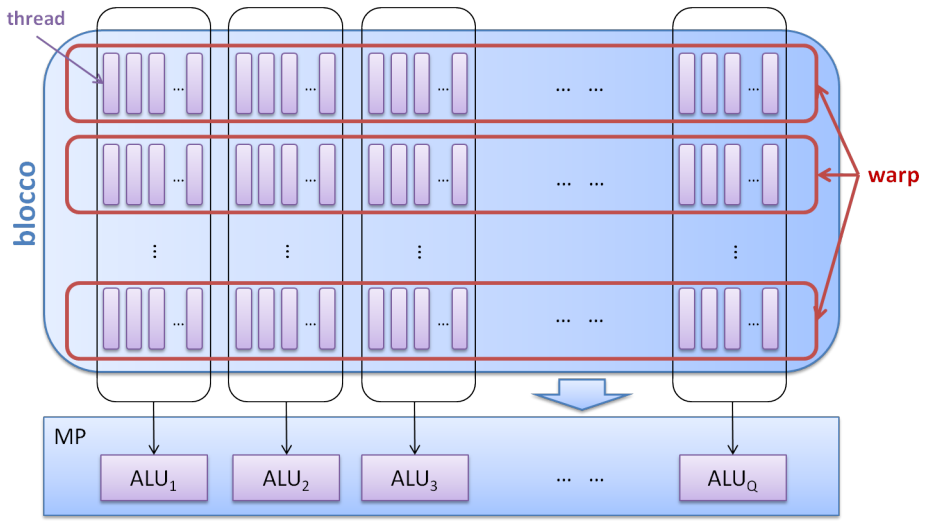


Figura 5.10: All'interno di ogni blocco, i thread vengono distribuiti tra le ALU, ed eseguiti un warp alla volta.

5.3 Criterio di ottimalità

L'obiettivo che ci siamo posti è individuare l'**operating point**, ovvero il miglior numero di thread in cui decomporre il lavoro dell'algoritmo A_{par}. Ricordando che l'architettura è composta essenzialmente da:

- P MultiProcessori (MP) e
- Q ALU per ogni MP,

tale punto deve essere tale da permetterci di ottenere la migliore prestazione possibile dalle $P \cdot Q$ unità calcolanti a disposizione. Lo speed up ideale è uguale al numero di unità calcolanti (come visto nella sezione 4.4),

$$S^I = P \cdot Q.$$

Come per tutte le architetture, bisogna capire quante unità utilizzare e in quali casi.
Come segue dall'analisi fatta nella sezione 4.4, algoritmi in cui l'overhead cresce troppo velocemente non risultano adatti all'esecuzione su GPU: questo perché in algoritmi del genere non ha senso dividere il lavoro in troppe

componenti (rivedere la definizione 4.11 di *grado di concorrenza di un algoritmo*), e il numero di unità calcolanti di un'architettura come la GPU potrebbe essere già troppo, soprattutto considerando che in genere la CPU dello stesso sistema ha latenze di memoria minori, potendo risultare più performante.
Pensiamo quindi a quegli algoritmi per cui non vale questo limite. Dobbiamo limitarci ad utilizzare $P \cdot Q$ thread per avvicinarci ad S^I? In realtà possiamo osservare che il numero di thread eseguiti da un MP, e quindi dalle sua ALU equivale a fargli compiere più operazioni, quindi ad assegnargli un *sottoproblema di dimensione maggiore*, e allora, per la 4.12, aumentando il lavoro sui diversi MP e ALU lo speed up ci dobbiamo avvicinare allo speed up ideale.
Per quanto riguarda le latenze, esse, come abbiamo detto, il loro peso viene ridotto molto dalla contemporaneità del lavoro di molti thread, quindi approfittare dell'alta possibilità di concorrenza può portare grande vantaggio. Un maggior numero di thread significa però anche maggiore overhead.
Siccome latenze ed overhead crescono in modo da non poter essere resi del tutto trascurabili con la concorrenza, se vogliamo davvero ottimizzare la prestazione di un algoritmo dobbiamo trovare un compromesso. Dobbiamo cioè trovare una coppia (p, q) che individui il numero di blocchi e di thread per blocco in cui dividere il lavoro dell'algoritmo per ottenere una buona prestazione tenendo l'overhead e le latenze quanto più bassi è possibile: è allora importante individuare il minimo valore di (p, q) per cui la prestazione è soddisfacente.

Prima di tutto chiariamo il funzionamento e il tempo dell'esecuzione su un solo MP, che, come abbiamo già detto, è assimilabile ad una macchina SIMD che può eseguire $dimW$ operazioni identiche alla volta, o anche ad un processore vettoriale la cui VL è un multiplo di $dimW$. Un blocco di A_{Par} verrà eseguito su un MP un warp alla volta (anche se $q \leq dimW$).

CASO $p = 1$, $q \leq dimW$ Consideriamo innanzitutto che sia $p = 1$ e $q \leq dimW$: sull'MP viene eseguito un solo warp, di cui tutti i thread, per definizione, vengono eseguiti contemporaneamente. Dunque, possiamo dire che il tempo d'esecuzione del warp è il massimo tra i tempi d'esecuzione dei

singoli thread che lo compongono[2]:

$$T_c(A_{par}, 1, q, N) = T_{c[flop]}(warp_i, dimW, N =$$

$$= \max_{0 \leq i \leq q}(T_{s[flop]}(A_{par}, th_i, N_i) + T_{s[mem]}(A_{par}, th_i, N_i) + T_O(A_{par}, th_i, N_i)) +$$

$$+ T_O(A_{par}, warp, q, N)$$

dove

- th_i è l'i-mo thread del warp,
- N_i la dimensione del sottoproblema assegnato all'i-mo thread,
- $T_O(A_{par}, th_i, N_i)$ è l'overhead proprio dello singolo thread,
- $T_O(A_{par}, warp, N)$ è l'overhead dovuto alla convivenza dei diversi thread.

Se per semplicità supponiamo il carico di lavoro ben bilanciato, allora, con $i \in \{0..q\}$

$$T_c(A_{par}, 1, q, N) = T_{c[flop]}(warp_i, dimW, N =$$

$$= T_{s[flop]}(A_{par}, th_i, \frac{N}{q}) + T_{s[mem]}(A_{par}, th_i, \frac{N}{q}) + T_O(A_{par}, warp, N).$$

dove $T_O(A_{par}, warp, N)$ stavolta comprende sia l'overhead proprio dei singoli thread (supposto uguale per ognuno), sia quello dovuto alla convivenza dei diversi thread.

Osservazione 2. $T_{s[mem]}(th_i, \frac{N}{q})$ può essere ridotto gestendo opportunamente la gerarchia di memoria.

Osservazione 3. $T_{s[flop]}(th_i, \frac{N}{q})$ può essere ridotto introducendo ILP, cioè organizzando le operazioni del thread in modo cda sfruttare il parallelismo permesso dalla pipeline.

[2]Questo è caratteristico della macchina SIMT (vedi sezione 5.1.3), che dal punto di vista della programmazione è assimilabile ad una macchina MIMD a memoria condivisa. Tuttavia l'architettura resta più vicina ad una SIMD, e lo sbilanciamento nella divisione del carico di lavoro si traduce in 'lavoro inutile'di alcune unità.

Osservazione 4. $T_{s[mem]}(th_i, \frac{N}{q})$ può essere ridotto introducendo ILP, cioè organizzando le istruzioni di accesso alle variabili in modo che possano essere realizzate in maniera indipendente.

Osservazione 5. $T_O(A_{par}, warp, N)$ può essere ridotto riducendo opportunamente le divergenze ed il numero di cicli.

Assunto questo, possiamo cominciare ad utilizzare il tempo d'esecuzione di un warp come base per le analisi successive.

CASO $p > 1$, $q > dimW$ Sapendo che il numero di warp in un blocco e il numero di blocchi sono in genere maggiori di quanto visto finora, supponiamo senza perdere generalità che p sia multiplo del numero P di MP, e q sia multiplo di $dimW$. Introduciamo la seguente definizione:

Definizione 5.3 (Occupazione)**.** *L'**occupazione** di un MP è una funzione dello stesso algoritmo A_{Par}, del numero q di thread di un blocco eseguito su quell'MP e del numero di blocchi che devono essere eseguiti su quell'MP, ed è definita come:*

$$\vartheta(A_{par}, q, p) = \frac{\#active\_warps\_block}{\#max\_warps\_per\_MP} =$$

$$= \frac{q}{dimW} \cdot \frac{1}{\#max\_warps\_per\_MP} \cdot \#active\_blocks\_MP(A_{par}, q, p)$$

dove

- $\#max\_warps\_per\_MP$ *dipende dalle caratteristiche hardware della GPU,*
- $\#threads\_per\_warp$ *dipende dalle caratteristiche hardware della GPU,*
- $\#active\_blocks\_MP = min(p, \#max\_blocks\_MP(A_{par}, q)$*,*
- $\#max\_blocks\_MP(A_{par}, q)$ *è una quantità che dipende strettamente dai limiti caratteristici dell'hardware, ma anche dalle caratteristiche dell'algoritmo (soprattutto dall'utilizzo che esso prevede della memoria locale, cioè di* shared memory *e* registri*) e dal numero di threads q.*

Descrive 'quanto' è sfruttato l'MP.

I diversi warp a cui è stavolta affidata l'esecuzione, verranno eseguiti concorrentemente sull'MP, idealmente 'uno dopo l'altro', ma in realtà concorrendo per l'utilizzo delle risorse.
Esisteranno dunque due funzioni dell'occupazione,

- $\varphi_1(\vartheta(A_{par}, p, q))$,
- $\varphi_2(\vartheta(A_{par}, p, q))$,

che descriveranno l'andamento delle prime due componenti del tempo d'esecuzione (rif. definizione 4.1[3]), espresse rispetto alla somma dei tempi d'esecuzione dei singoli warp.
Sarà cioè, posto $nw = p/P \cdot q/dimW$ il numero di warp attivi:

$$T_c(A_{par}, p, q, N) =$$

$$= \frac{\sum_{i=0}^{nw-1} \left(T_{c[flop]}(warp_i, dimW, N/nw)\right)}{\varphi_1(\vartheta(A_{par}, p, q))} +$$

$$+ \frac{\sum_{i=0}^{nw-1} \left(T_{c[mem]}(warp_i, dimW, N/nw)\right)}{\varphi_2(\vartheta(A_{par}, p, q))} +$$

$$+ T_O(A_{Par}, p, q, N)$$

Per minimizzare il tempo $T_c(A_{par}, p, q, N)$ dobbiamo allora massimizzare le due funzioni φ_1 e φ_2. Le due funzioni, intuitivamente e sperimentalmente, sono crescenti e limitate. Ma non crescono necessariamente come l'occupazione. Infatti, secondo quanto descritto in [47], [50] e [45], massimizzare l'occupazione non è l'unico modo per nascondere le latenze: si dimostra che esistono altre tecniche[4].
Se chiamiamo

[3] Ricordiamo che

$$T_c(A, P, N) = T_{seq}(A, N) + \frac{T_{par}(A, N)}{P} + T_O(A, P, N).$$

[4] Le tecniche suggerite da Demmel e Volkov mirano a massimizzare l'utilizzo dei registri e ad aumentare il numero di operazioni per thread per ottenere la riduzione sia della latenza aritmetica che della latenza di memoria. In questo modo si ottengono alte percentuali

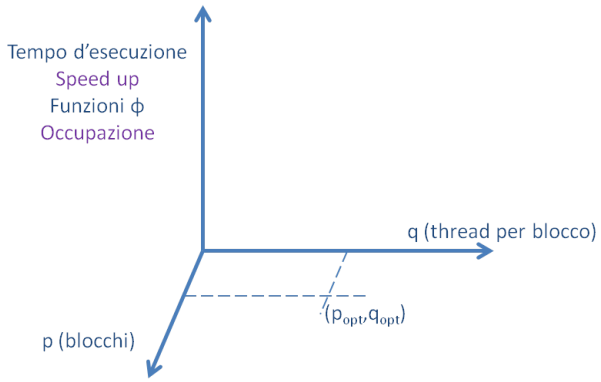


Figura 5.11: Le funzioni *tempo d'esecuzione*, *speed up*, *occupazione* e le funzioni φ si possono disegnare in funzione di p e q, quindi al punto di ottimo corrisponderà un particolare valore di ognuna di esse.

- $\#\_max\_blocks\_MP$ il numero massimo di blocchi attivi nello stesso momento su un MP e
- $\#\_max\_warps\_MP$ il numero massimo di warp attivi nello stesso momento su un MP,

possiamo dire che l'**operating point** che stiamo cercando è dato dalla coppia (vedi figura 5.11)

$$(p_{opt} \leq P \cdot \#\_max\_blocks\_MP, q_{opt} \leq dimW \cdot \#\_max\_warps\_MP)$$

che massimizza φ_1 e φ_1 .

Funzione φ_1 Tale funzione, che dipende dal numero di thread totali e quindi dal numero di warp, cresce tra un valore minimo e un valore massimo. Il valore minimo di φ_1 coincide con il valore minimo dell'occupazione (1 warp attivo), cioè:

$$\varphi_1(\vartheta(A_{Par}, 1, dimW)) \leq \varphi_1(\vartheta(A_{Par}, p, q)) \qquad \forall p, q$$

del picco di performance della macchina già con un numero limitato di warp per MP (approfondiremo nel capitolo 6). Anche in [52], approfondendo il caso dell'operazione di riduzione, si evidenzia come la performance possa crescere alzando il grado di ILP (Instruction Level Parallelism) ed ottimizzando il codice in modo che ci sia meno overhead.

Invece il (primo) valore massimo si ha per un numero di warp

$$nw_{[f]opt} = \frac{r}{\alpha \cdot k}$$

dove

- $\frac{r}{\alpha \cdot k}$ è quindi il numero di warp attivi nel caso di valore massimo,
- α è il numero di operazioni indipendenti tra loro per ogni threads del warp, e quindi è il grado di concorrenza delle stesse componenti in cui è stato diviso il lavoro di A_{Par} (che quindi permettono l'ILP), ovvero il numero di output calcolati dallo stesso thread in maniera indipendente l'uno dall'altro,
- r è la profondità della pipeline aritmetica di ognuna delle Q ALU,
- $k = dimW/Q$,

cioè

$$\varphi_1(\vartheta(A_{Par}, p, q)) \leq \varphi_1(\vartheta(A_{Par}, p_f, q_f) \qquad \forall p, q$$

dove

$$(p_f, q_f)$$

è tale che

$$\frac{p_f}{P} \cdot \frac{q_f}{dimW} = \frac{r}{\alpha \cdot k}$$

cioè

$$p_f \cdot q_f = \frac{r}{\alpha \cdot k} \cdot P \cdot dimW.$$

Notiamo che $nw_{[f]opt}$ e il prodotto $p \cdot q$, possono essere molto minori dei valori che massimizzano l'occupazione.

Funzione φ_2 Anche tale funzione, dipende dal numero di thread e quindi dal numero di warp. Cresce tra un valore minimo e un valore massimo. Il valore minimo di φ_2 coincide con il valore minimo dell'occupazione (1 warp attivo), cioè:

$$\varphi_2(\vartheta(A_{Par}, 1, dimW)) \leq \varphi_2(\vartheta(A_{Par}, p, q)) \qquad \forall p, q$$

Invece il (primo) valore massimo si ha per un numero di warp

$$nw_{[m]opt} = \frac{l}{\psi(\beta) \cdot dimW}$$

dove

- l è la latenza media di accesso alla memoria,
- β è il numero di operazioni di memoria indipendenti tra loro che compie ogni thread del warp e quindi il grado di concorrenza delle operazioni di memoria di ogni thread, anche questo in relazione al numero di output calcolati dallo stesso thread in maniera indipendente l'uno dall'altro e quindi dall'ILP,
- ψ è non decrescente con β

cioè

$$\varphi_2(\vartheta(A_{Par}, p, q)) \leq \varphi_2(\vartheta(A_{Par}, p_m, q_m) \qquad \forall p, q$$

dove

$$(p_m, q_m)$$

è tale che

$$\frac{p_m}{P} \cdot \frac{q_m}{dimW} = \frac{l}{\psi(\beta) \cdot dimW}$$

cioè

$$p_m \cdot q_m = \frac{l}{\psi(\beta)} \cdot P.$$

Notiamo che $nw_{[m]opt}$ può essere minore del valore nw che massimizza l'occupazione.

Operating Point Il nostro **operating point** sarà dato dalla coppia (p_{opt}, q_{opt}) tale che:

$$p_{opt} \cdot q_{opt} = max(p_f \cdot q_f, p_m \cdot q_m).$$

Infatti, in questo modo certamente saranno massimizzate entrambe le funzioni φ_1 e φ_2.

Considerando che nella pratica la latenza di memoria è in genere molto maggiore della latenza aritmetica, cioè

$$l >> r$$

c'è da aspettarsi che risulti

$$nw_{[m]opt} > nw_{[f]opt}$$

e che quindi l'operating point coincida con il punto di massimo della funzione φ_2, ovvero che sia

$$(p_{opt}, q_{opt}) = (p_m, q_m).$$

Capitolo 6

Confronto con architettura reale

Le esperienze che riporteremo sono state fatte su un'architettura della famiglia NVIDIA CUDA.

6.1 Architettura NVIDIA CUDA

L'architettura CUDA delle GPU NVIDIA attuali si può descrivere genericamente nel seguente modo ([45, 47, 53, 55, 56]):

- È composta da P Streaming Multiprocessors (**SM**), o core, ognuno costituito da:
  - 1 instruction unit - **IU**
  - Q unità calcolanti per le operazioni **MAD in singola precisione** (2 operazioni fp) (pipelined) - **SP**,
  - $Q2$ unità calcolanti per le **funzioni speciali** (pipelined) - **SFU**,
  - $Q3$ unità calcolanti per le operazioni **MAD in doppia precisione** (2 operazioni fp) (pipelined) - **DP**.
- Gli SM sono raggruppati in T **TPC** (*Texture Processing Cluster*).
- La gerarchia di memoria è fatta di

  - una certa quantità di memoria on-chip per i **registri**[1],
  - una certa quantità di **shared memory**[2] anch'essa on-chip,
  - diversi livelli di **cache** off-chip,
  - una certa quantità di **constant** memory (sola lettura),
  - una certa quantità di **texture memory**,
  - una **memoria globale**[3] DRAM.

- L'IU è in grado di decodificare 1 istruzione ogni k **cicli di clock**. Ogni SP può far partire 1 istruzione per ogni ciclo di clock[4]. Quindi su un SM ogni istruzione in singola precisione decodificata verrà eseguita $k \cdot Q$ volte [53], ovvero la dimensione del **warp** (vedi definizione 5.1) è

$$dimW = k \cdot Q.$$

.

CUDA stabilisce anche un modello di programmazione. Secondo questo modello, la parte parallela dell'algoritmo A_{Par} deve essere divisa in un certo numero di *kernels*, ovvero nuclei computazionali (di fatto dei sotto-algoritmi) scritti sottoforma di funzioni che vengono eseguite interamente sulla GPU. La struttura di ogni kernel sarà organizzata in livelli di parallelismo in corrispondenza dei livelli dell'architettura[5]. Nella pratica il lavoro viene diviso tra:

- p **blocchi di threads**[6]: virtualizzazione degli SM; ogni blocco viene eseguito interamente su un SM.

- q **thread per blocco**[7]: virtualizzazione CUDA degli SP-SFU-DP; ogni thread verrà eseguito da un'unità dell'SM.

[1]Corrispondenti ai **registri** definiti nel Capitolo 5.
[2]Corrispondente alla **shared memory** definita nel Capitolo 5.
[3]Corrispondente alla **memoria globale** definita nel Capitolo 5.
[4]Gli SP si comportano come le **ALU** definite nel Capitolo 5.
[5]L'opportunità di far corrispondere una struttura logica dell'algoritmo alla struttura dell'architettura è stata messa in evidenza anche nel capitolo 5.
[6]corrispondente ai *work-group* di OpenCL.
[7]corrispondenti ai *work-item* di OpenCL.

Le quantità p e q vengono assegnate dal programmatore al momento della chiamata del kernel.
Al momento dell'esecuzione i *blocchi* vengono suddivisi in *warp*.
I thread dello stesso warp si possono intendere come organizzati per lavorare in maniera **SPMD**[8] (Single Program Multiple Data), per questo si può considerare il blocco di thread come un unico *thread vettoriale*[9] [45]. In caso di branch la cui condizione viene valutata diversamente da qualche unità del warp, complessivamente dovranno essere considerati entrambi i casi, con le ovvie conseguenze sulla performance: ci si riferisce a queste situazioni con l'espressione **divergenza** (di valutazione della condizione) [54].
Per quanto riguarda la gestione della memoria, essa ripercorre il funzionamento descritto nel capitolo 5, cioè:

- i thread contemporaneamente attivi su un SM si dividono lo spazio del registro in modo che ognuno abbia la sua sezione privata.
- i threads dello stesso blocco condividono la shared memory,
- la shared memory durante l'esecuzione di un kernel viene divisa tra i blocchi contemporaneamente attivi sull'SM,
- tutti i blocchi e i thread in essi contenuti condividono la cache, la constant memory e la memoria globale.

Inoltre

- i thread della stessa metà di un warp (**half-warp**) leggono contemporaneamente lo stesso dato dalla constant memory,
- i thread dello stesso harf-warp eseguono accessi coalescenti alla memoria nel tempo di un'unica transazione,
- la memoria globale è condivisa da tutti i blocchi o thread sulla GPU ed è persistente rispetto ai successivi lanci di kernel.

[8] La definizione SPMD evidenzia il fatto che i singoli threads possono seguire branch diversi, ma che corrisponde architetturalmente ad un modello SIMD.
[9] L'analogia con i processori vettoriali è stata evidenziata già nel capitolo 5.

L'architettura in genere definisce un numero massimo di blocchi e di thread che possono essere eseguiti con un kernel.
Lo **scheduler** di ogni SM farà poi in modo che i blocchi vengano gestiti contemporaneamente secondo

- un numero massimo di blocchi, che dipende soprattutto dalla dimensione della shared memory e dei registri,
- un numero massimo di thread e quindi di warps.

6.1.1 Architettura NVIDIA: dal codice G80 al codice GT200

Nel 2006 NVIDIA mette sul mercato le prime schede video basate su architettura **G80**. Famose tra queste restano le GeForce 8800.
L'anno dopo la casa lancia una nuova linea di prodotti destinata appositamente al calcolo parallelo general purpose e all'HPC, a cui dà il nome commerciale *Tesla*, e che si basa proprio sulla G80. Questa linea dalle caratteristiche esclusive si distingue dalle linee GeForce e Quadro, pensate rispettivamente per la grafica di consumo e per le visualizzazioni professionali.
Le prime architetture G80, tra cui la GeForce 8800 (vedi figura 6.1) portavano nel mondo delle GPU delle importanti novità [15]:

- G80 è stata la prima architettura GPU a supportare il linguaggio C, permettendo ai programmatori di sfruttare il processore grafico senza dover imparare altri linguaggi,
- G80 è stata la prima architettura GPU ad architettura unificata, che superava la separazione tra le pipeline di vertici e pixel,
- G80 è stata anche la prima architettura ad utilizzare il meccanismo dei thread,
- G80 ha introdotto il modello d'esecuzione SIMT[10] in cui più thread indipendenti eseguono concorrentemente una singola istruzione,

[10] Al modello SIMT si è già accennato nel capitolo 5.

Figura 6.1: NVIDIA GeForce 8800 GTX, sul mercato a metà del 2006, primo prodotto con architettura G80.

- G80 ha introdotto la shared memory e la possibilità di sincronizzazione tra thread.

Lo schema a blocchi dell'architettura G80 è mostrato in figura 6.2. L'architettura si può descrivere nel seguente modo:

- È composta da 16 Streaming Multiprocessors (**SM**), ognuno costituito da:
  - 1 instruction unit - **IU**
  - 8 unità calcolanti per le operazioni **MAD in singola precisione** (2 operazioni fp) (pipelined) - **SP**,
  - 2 unità calcolanti per le **funzioni speciali** (pipelined) - **SFU**,
- Gli SM sono raggruppati in 8 **TPC** (*Texture Processing Cluster*) e ogni TPC contiene 2 SM.
- La gerarchia di memoria è fatta di
  - 32KB di memoria on-chip per i **registri**,
  - 16KB di **shared memory** anch'essa on-chip,
  - diversi livelli di **cache** off-chip,
  - una **memoria globale** DRAM.

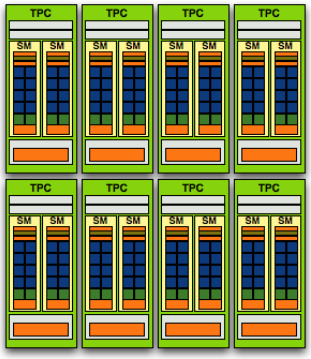


Figura 6.2: Schema a blocchi dell'architettura NVIDIA G80.

- L'IU è in grado di decodificare 1 istruzione ogni 4 **cicli di clock**. Ogni SP può far partire 1 istruzione per ogni ciclo di cloc. Quindi la dimensione del **warp** è $dimW = 32$.

Sull'onda del successo dell'architettura G80, nel 2008 NVIDIA è pronta a mettere sul mercato i primi prodotti basati su una nuova architettura che segue lo stesso spirito: l'architettura **GT200**, sulla quale vengono costruite schede GeForce e Quadro, ma anche le nuove Tesla serie '10'. Numeri a parte, la novità più importante introdotta dalla GT200 è la possibilità di calcolo in *doppia precisione*.
Oltre a questo, il numero di core è cresciuto da 128 a 240, i registri sono stati raddoppiati, è stato aggiunto un meccanismo di accesso alla memoria coalescente. L'architettura NVIDIA GT200 si può descrivere nel seguente modo ([45, 47, 53, 55, 56]):

- È composta da 30 Streaming Multiprocessors (**SM**),ognuno costituito da:
  - 1 instruction unit - **IU**
  - 8 unità calcolanti per le operazioni **MAD in singola precisione** (2 operazioni fp) (pipelined) - **SP**,
  - 2 unità calcolanti per le **funzioni speciali** (pipelined) - **SFU**,
  - 13 unità calcolante per le operazioni **MAD in doppia precisione** (2 operazioni fp) (pipelined) - **DP**.

Figura 6.3: NVIDIA Tesla C1060, sul mercato a metà del 2008, pensata per l'HPC, architettura GT200.

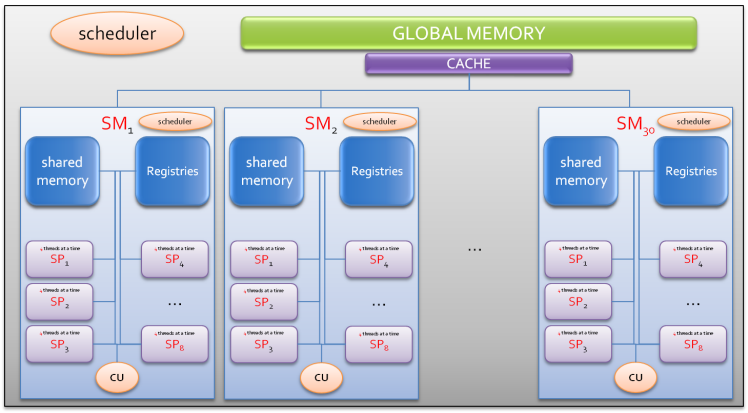


Figura 6.4: Schema funzionale dell'architettura NVIDIA GT200.

- La profondità della pipeline per le operazioni in singola precisione è stimata come $r = 24$
- Gli SM sono raggruppati in 10 **TPC** (*Texture Processing Cluster*) e ogni TPC contiene 3 SM.
- La gerarchia di memoria è fatta di
  - 64KB di memoria on-chip per i **registri**,
  - 16KB di **shared memory** anch'essa on-chip,
  - tre livelli di **cache** off-chip: **L1** dedicata per ogni SM, **L2** condivisa per TPC ed **L3** condivisa da gli SM di tutti i TPC),
  - una **memoria globale** DRAM.
- L'IU è in grado di decodificare 1 istruzione ogni 4 **cicli di clock**. Ogni SP può far partire 1 istruzione per ogni ciclo di cloc. Quindi la dimensione del **warp** è $dimW = 32$.
- possono essere schedulati al massimo 65536×65536 blocchi per kernel,
- possono essere schedulati al massimo 512 thread per blocco,
- possono essere schedulati al massimo 1024 thread per SM, quindi 32 warp.

6.1.2 Architettura NVIDIA codice GF100 - *Fermi*

Forte dell'esperienza fatta dai propri clienti con le sue prime GPU *general purpose*, NVIDIA si pone l'obiettivo di apportare alcuni miglioramenti che emergono come esigenze degli sviluppatori di applicazioni. In particolare, intende [15]:

- migliorare la performance per le operazioni in doppia precisione,
- creare una gerarchia di memoria cache più facilmente sfruttabile, soprattutto in caso di algoritmi che non si servono di shared memory (area che allora resta sprecata),

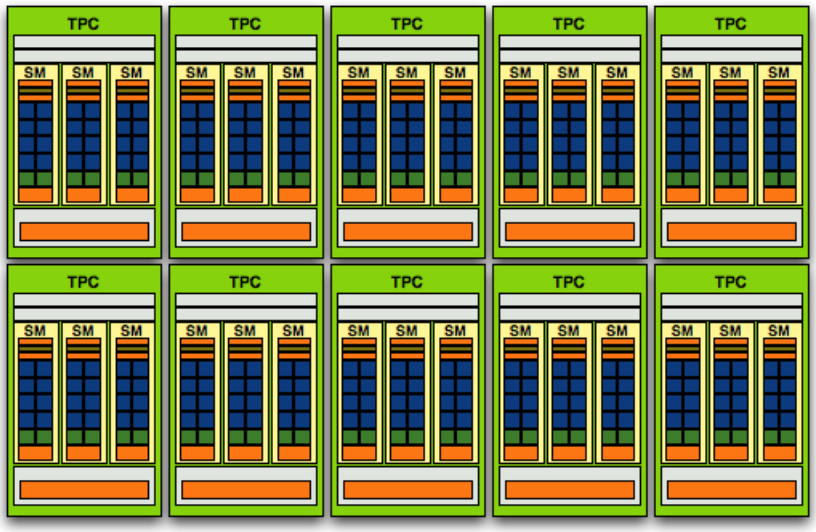


Figura 6.5: Schema logico di una GPU NVIDIA GT200.

- aumentare la shared memory disponibile nei casi in cui invece sia richiesta, in modo da poter velocizzare le applicazioni che ne fanno uso,
- implementare un più veloce context switching, anche tra applicazioni diverse,
- velocizzare le operazioni atomiche.

Il lavoro su questi punti porta lo sviluppo di un nuovo progetto, che mira a migliorare la prestazione ma anche la programmabilità: nel 2009 NVIDIA annuncia l'architettura *Fermi* (GT300/GF100/GF104). Su di essa vengono costruiti dispositivi GeForce, Quadro e anche le nuove Tesla serie '20'. Questa nuova architettura viene pubblicizzata come '4 volte più veloce della GT200'. La sua logica non è diversa, ma l'organizzazione contiene delle importanti novità, rispetto alle due precedenti architetture che tra loro differivano solo di qualche parametro.

Evidenziando le differenze con le architetture precedenti, in un'architettura NVIDIA Fermi:

Figura 6.6: NVIDIA GeForce GTX 480, sul mercato nel 2010, architettura Fermi.

- ci sono al massimo $P = 16$ Streaming Multiprocessor (SM di *terza generazione*[11].),
- negli SM viene implementato il nuovo standard IEEE per le operazioni floating point sia in singola che in doppia precisione, cioè l'operazione base è la Fused Multiply-Add (FMA)[12] che risulta più veloce ed accurata della semplice MAD,
- ogni SM è composto di
  - 32 unità calcolanti in singola precisione (SP),
  - 4 unità per le funzioni speciali (SFU),
- ogni SM può compiere 16 operazioni FMA in doppia precisione, quadruplicando in questo caso la performance della GT200,
- gli SM contengono **due** scheduler, potendo così procedere indipendentemente con l'esecuzione di due warp diversi, che concorreranno sui 32 SP, o sulle SFU,
- la peak performance per le operazioni floating point risulta in generale 8 volte maggiore di quella della GT200,

[11] Così si riferisce a questi nuovi SM NVIDIA stessa [15]
[12] L'architettura GT200 implementa l'FMA solo in doppia precisione.

- ogni SM possiede una memoria RAM di 64 KB che può essere configurata per essere divisa tra **shared memory** e cache di primo livello **L1**, in modo che l'una occupi 48 KB e l'altra 16 KB o viceversa,
- si utilizzano indirizzi a 64 bit,
- sulla GPU possono concorrere fino a 16 kernel: questa è la novità che salta immediatamente all'occhio, dal momento che fino alla GT200 la GPU veniva interamente dedicata all'unico kernel in esecuzione, con importanti limitazioni all'organizzazione del lavoro parallelo.

Questa architettura si è velocemente fatta spazio e anzi, si sta già imponendo, nell'ambito dello sviluppo di applicazioni, anche per il calcolo scientifico e l'HPC.

6.2 Validazione dell'analisi teorica

Le prestazioni delle GPU NVIDIA sono oggetto di analisi scientifica da parte di molti ricercatori nel mondo. In particolare, al fine di validare l'analisi teorica condotta in questo lavoro di tesi si è seclto di fare riferimento alla scuola di Jim Demmel[13] dell'Università della California a Berkeley, presso cui sono stati condotti molti esperimenti orientati a stimare le diverse latenze attraverso benchmark [45], a individuare la *peak performance* di volta in volta dell'architettura in questione ed osservando quale organizzazione del lavoro tra i thread riuscisse ad ottenerlo.
In [46, 47, 50] viene osservato che preoccupandosi di massimizzare l'occupazione senza fare attenzione ad altri aspetti può invece portare addirittura ad una perdita in termini di prestazioni, nonostante le guide commerciali sull'utilizzo di CUDA incoraggino l'utilizzo del maggior numero di thread possibile per ottenere le più alte performace sottolineando che questo permette di ammortizzare le latenze aritmetiche e di memoria. Questa è l'idea

[13]Professore di Matematica e Informatica all'Università di Berkeley, California, fondatore e direttore del CITRIS (Center for Information Technology Research in the Interest of Society).

che le funzioni φ descrivono.

Nel seguito prenderemo in considerazione le esperienze effettuate su due architetture GPU NVIDIA:

1. un'architettura **GT200**, in [50],
2. una GPU NVIDIA GeForce GTX480, ovvero su un'architettura di tipo Fermi, codice **GF100**, in [47].

Per ogni esperienza mostreremo che il punti di ottimo ricavati nella sezione 5.3

$$nw_{[f]opt} = \frac{r}{\alpha \cdot k}$$

e

$$nw_{[m]opt} = \frac{l}{\psi(\beta) \cdot dimW}$$

corrispondono a quanto si verifica sperimentalmente.

Caso 1 Per l'architettura GT200 si osserva che:

- la latenza aritmetica è di circa 24 cicli, cioè $r = 24$,
- il completamento dell'esecuzione di un warp avviene in 4 cicli, cioè $k = 4$
- la latenza si può coprire attivando contemporaneamente $24/4 = 6$ warp ($6 \cdot 32 = 192$ thread),
- si stima che la latenza di memoria sia dell'ordine di centinaia di cicli, in particolare per la memoria globale nelle guide NVIDIA alla programmazione CUDA viene documentata una latenza di circa 400-600 cicli. Per il seguito poniamo che sia $l \approx 400$.

Considereremo separatamente 5 case study, descritti da 5 algoritmi predisposti a mettere in evidenza 5 situazioni differenti.

Algoritmo 1.1 Un ciclo in cui si esegue un'operazione MAD ad ogni passo e per ogni thread (vedi figura 6.7), supponendo che tutti i dati siano in registro, dunque limitando il peso della latenza di memoria e rendendo il tempo d'esecuzione dipendente soprattutto dal tempo di calcolo. Il risultato dell'esecuzione al variare dell'occupazione degli SM è riportato in termini di frazione della *peak performance*. Nella pratica si osserva che si raggiunge una prestazione molto vicina a quella di picco con 192 thread per SM (vedi figura 6.8).

Nei termini dell'analisi della sezione 5.3 a tale algoritmo corrisponde un valore $\alpha = 1$. Allora sarà:

$$nw_{[f]opt} = \frac{r}{\alpha \cdot k} = \frac{24}{4} = 6 \Longrightarrow 6 \cdot 32 = 192thread,$$

cioè esattamente quanto osservato.

Algoritmo 1.2 Simile al precedente, nel ciclo però si eseguono 3 operazioni MAD indipendenti ad ogni passo e per ogni thread (vedi figura 6.9). Il risultato dell'esecuzione al variare dell'occupazione degli SM è riportato in termini di frazione della *peak performance*. Nella pratica si osserva che si raggiunge una prestazione molto vicina a quella di picco con 64 thread per SM, ovvero con il 6% di occupazione dell'SM (vedi figura 6.10).

Nei termini dell'analisi della sezione 5.3 a tale algoritmo corrisponde un valore $\alpha = 3$. Allora sarà:

$$nw_{[f]opt} = \frac{r}{\alpha \cdot k} = \frac{24}{4 * 3} = 2 \Longrightarrow 2 \cdot 32 = 64thread,$$

cioè esattamente quanto osservato.

Algoritmo 1.3 Copia di una parola di 64 bit dalla memoria globale al registro per ogni thread (vedi figura 6.11), evitando ogni calcolo per rendere il tempo d'esecuzione dipendente soprattutto dagli accessi alla memoria. Il risultato dell'esecuzione al variare dell'occupazione degli SM è riportato in termini di frazione della *peak performance*. Nella pratica si osserva che si raggiunge una buona percentuale del picco con 320

thread per SM, ovvero con circa il 30% di occupazione dell'SM (vedi figura 6.12).
Nei termini dell'analisi della sezione 5.3 a tale algoritmo corrisponde un valore $beta = 1$. Allora $\psi(\beta)$ è minima, il valore ottimo da noi stimato si dovrebbe avere per

$$nw_{[m]opt} = \frac{l}{\psi(\beta) \cdot dimW} \approx \frac{400}{32} = 12,5 \Longrightarrow 13warp$$

che corrisponde a circa il 40% di occupazione, e non si allontana molto da quello verificato sperimentalmente.

Algoritmo 1.4 Simile al precendente, però si realizzano 2 copie indipendenti per ogni thread (vedi figure 6.13). Il risultato dell'esecuzione al variare dell'occupazione degli SM è riportato in termini di frazione della *peak performance*. Nella pratica si osserva che si raggiunge la stessa percentuale del picco del caso precedente con 192 thread per SM, ovvero con circa il 20% di occupazione dell'SM (vedi figura 6.14). Nei termini dell'analisi della sezione 5.3 a tale algoritmo corrisponde un valore $\beta = 2$. Allora, supponendo che $\psi(\beta) = \beta$ il valore ottimo da noi stimato si dovrebbe avere per

$$nw_{[m]opt} = \frac{l}{\psi(\beta) \cdot dimW} \approx \frac{400}{2 * 32} = 6,5 \Longrightarrow 7warp$$

che corrisponde a circa il 20% dell'occupazione, cioè esattamente quanto osservato.

Algoritmo 1.5 Simile al precendente, però si realizzano 4 copie indipendenti per ogni thread. Il risultato dell'esecuzione al variare dell'occupazione degli SM è riportato in termini di frazione della *peak performance*. Nella pratica si osserva che si raggiunge la stessa percentuale del picco del caso precedente con 128 thread per SM, ovvero con circa il 10% di occupazione dell'SM (vedi figura 6.15). Nei termini dell'analisi della sezione 5.3 a tale algoritmo corrisponde un valore $\beta = 4$. Allora, supponendo che $\psi(\beta) = \beta$ il valore ottimo da noi stimato si dovrebbe

avere per

$$nw_{[m]opt} = \frac{l}{\psi(\beta) \cdot dimW} \approx \frac{400}{4 * 32} = 3,15 \Longrightarrow 4warp$$

che corrisponde a circa il 10% dell'occupazione, cioè esattamente quanto osservato.

Algoritmo 1.6 Simile al precendente, però si realizzano 8 copie indipendenti per ogni thread. Il risultato dell'esecuzione al variare dell'occupazione degli SM è riportato in termini di frazione della *peak performance*. Nella pratica si osserva che si raggiunge una buona percentuale del picco del caso precedente già con 64 thread per SM, ovvero con circa il 6% di occupazione dell'SM (vedi figura 6.16). Nei termini dell'analisi della sezione 5.3 a tale algoritmo corrisponde un valore $\beta = 8$. Allora, supponendo che $\psi(\beta) = \beta$ il valore ottimo da noi stimato si dovrebbe avere per

$$nw_{[m]opt} = \frac{l}{\psi(\beta) \cdot dimW} \approx \frac{400}{8 * 32} = 1,56 \Longrightarrow 2warp$$

che corrisponde a circa il 6% dell'occupazione, cioè esattamente quanto osservato.

Caso 2 Per l'architettura GF100 si osserva che

- ci sono **15 SM** da **32 SP** ognuno,
- la latenza aritmetica è di circa 18 cicli, cioè $r = 18$,
- il completamento dell'esecuzione di un warp avviene in 1 ciclo, cioè $k = 1$
- la latenza si può coprire attivando contemporaneamente 18 warp ($32 * 18 = 576$ thread),
- si stima la latenza di memoria globale [47] di $l \approx 800$ cicli.

Considereremo separatamente 5 case study, descritti da 5 algoritmi predisposti a mettere in evidenza 5 situazioni differenti. Precisamente:

Algoritmo 2.1 Un ciclo in cui si esegue un'operazione MAD ad ogni passo e per ogni thread (vedi figura 6.7), supponendo che tutti i dati siano in registro, dunque limitando il peso della latenza di memoria e rendendo il tempo d'esecuzione dipendente soprattutto dal tempo di calcolo. Il risultato dell'esecuzione al variare dell'occupazione degli SM è riportato in termini di frazione della *peak performance*. Nella pratica si osserva che si raggiunge una prestazione molto vicina a quella di picco con circa 576 thread per SM (vedi figura 6.17).
Nei termini dell'analisi della sezione 5.3 a tale algoritmo corrisponde un valore $\alpha = 1$. Allora sarà:

$$nw_{[f]opt} = \frac{r}{\alpha \cdot k} = \frac{18}{1} = 18 \Longrightarrow 18 \cdot 32 = 576 thread,$$

cioè esattamente quanto osservato.

Algoritmo 2.2 Simile al precedente, nel ciclo però si eseguono 2 operazioni MAD indipendenti ad ogni passo e per ogni thread (vedi figura 6.9). Il risultato dell'esecuzione al variare dell'occupazione degli SM è riportato in termini di frazione della *peak performance*. Nella pratica si osserva che si raggiunge una prestazione molto vicina a quella di picco con circa 288 thread per SM (vedi figura 6.18).
Nei termini dell'analisi della sezione 5.3 a tale algoritmo corrisponde un valore $\alpha = 2$. Allora sarà:

$$nw_{[f]opt} = \frac{r}{\alpha \cdot k} = \frac{18}{2} = 9 \Longrightarrow 9 \cdot 32 = 288 thread,$$

che corrisponde a quanto verificato sperimentalmente.

Algoritmo 2.3 Copia di una parola di un *float* dalla memoria globale al registro per ogni thread (vedi figura 6.11), evitando ogni calcolo per rendere il tempo d'esecuzione dipendente soprattutto dagli accessi alla memoria. Il risultato dell'esecuzione al variare dell'occupazione degli

SM è riportato in termini di frazione della *peak performance*. Nella pratica si osserva che per avere una buona percentuale del picco di performance in questo caso c'è bisogno di almeno il 100% di occupazione degli SM (vedi figura 6.19).
Nei termini dell'analisi della sezione 5.3 a tale algoritmo corrisponde un valore $beta = 1$. La funzione ψ per quest'architettura però, date le importanti differenze rispetto alla GT200, avrà un comportamento diverso, supponendo che

$$\psi(\beta) = \frac{\beta}{\gamma} \qquad \gamma \geq 1 \quad costante$$

e stimando $\gamma = 1,3$ il valore ottimo da noi stimato si dovrebbe avere per

$$nw_{[m]opt} = \frac{l}{\psi(\beta) \cdot dimW} \approx \frac{800}{\frac{32}{1,3}} = \frac{800}{25} = 32$$

che corrisponde al 100% dell'occupazione, cioè esattamente quanto osservato.

Algoritmo 2.4 Simile al precendente, però si realizzano 2 copie indipendenti per ogni thread (vedi figure 6.13). Il risultato dell'esecuzione al variare dell'occupazione degli SM è riportato in termini di frazione della *peak performance*. Nella pratica si osserva che per avere una buona percentuale del picco di performance in questo caso basta il 50% di occupazione degli SM (vedi figura 6.20).
Nei termini dell'analisi della sezione 5.3 a tale algoritmo corrisponde un valore $beta = 2$. Supponendo che

$$\psi(\beta) = \frac{\beta}{\gamma} \qquad \gamma \geq 1 \quad costante$$

e stimando $\gamma = 1,3$ il valore ottimo da noi stimato si dovrebbe avere per

$$nw_{[m]opt} = \frac{l}{\psi(\beta) \cdot dimW} \approx \frac{800}{\frac{2 \cdot 32}{1,3}} = \frac{800}{50} = 16$$

che corrisponde al 50% dell'occupazione, cioè esattamente quanto os-

```
for( int i = 0; i < 1024*1024; i += 1024 )
{
#pragma unroll
    for( int j = 0; j < 1024; j++ )
    {
        a = a * b + c;
    }
}
```

Figura 6.7: Algoritmo 1.1 [50].

servato.

Algoritmo 2.5 Simile al precendente, però si realizzano 4 copie indipendenti per ogni thread. Il risultato dell'esecuzione al variare dell'occupazione degli SM è riportato in termini di frazione della *peak performance*. Nella pratica si osserva che per avere una buona percentuale del picco di performance in questo caso basta il 25% di occupazione degli SM (vedi figura 6.21).
Nei termini dell'analisi della sezione 5.3 a tale algoritmo corrisponde un valore $beta = 4$. Supponendo che

$$\psi(\beta) = \frac{\beta}{\gamma} \qquad \gamma \geq 1 \quad costante$$

e stimando $\gamma = 1,3$ il valore ottimo da noi stimato si dovrebbe avere per

$$nw_{[m]opt} = \frac{l}{\psi(\beta) \cdot dimW} \approx \frac{800}{\frac{4 \cdot 32}{1,3}} = \frac{800}{100} = 8$$

che corrisponde al 25% dell'occupazione, cioè esattamente quanto osservato.

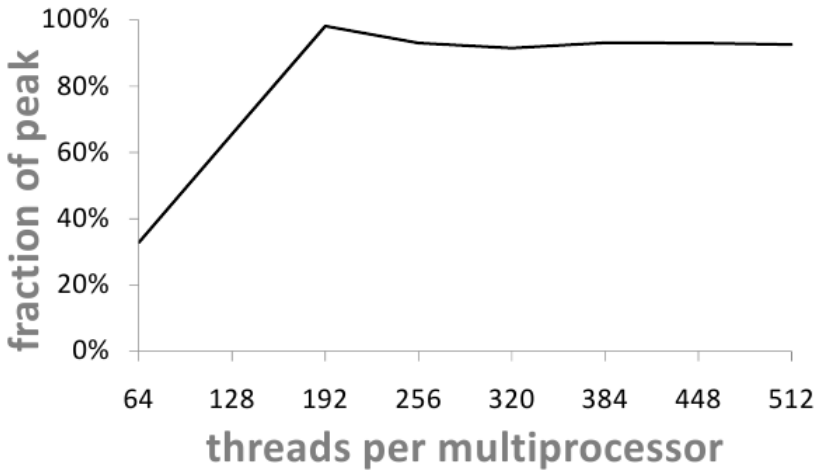


Figura 6.8: Frazione del picco di performance ottenuta al variare del numero di thread per l'Algoritmo 1.1 6.7 [50].

```
for( int i = 0; i < 1024*1024; i += 128 )
{
#pragma unroll
    for( int j = 0; j < 128; j++ )
    {
        a = a * b + c;
        d = d * b + c;
        e = e * b + c;
    }
}
```

Figura 6.9: Algoritmo 1.2 [50].

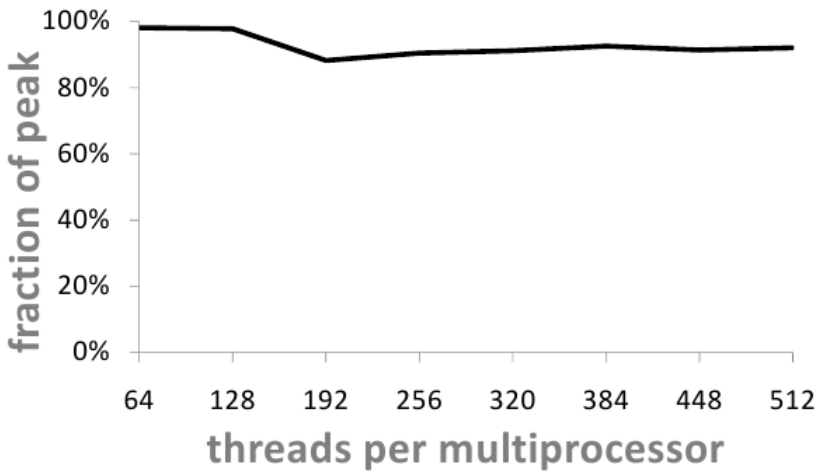


Figura 6.10: Frazione del picco di performance ottenuta al variare del numero di thread per l'Algoritmo 1.2 [50].

```
__global__ void memcpy( float2 *dst, float2 *src )
{
    int iblock = blockIdx.x
               + __mul24( blockIdx.y, gridDim.x );
    int index = threadIdx.x
              + __mul24( iblock, blockDim.x );

    float2 a0 = src[index];
    dst[index] = a0;
}
```

Figura 6.11: Algoritmo 1.3 [50].

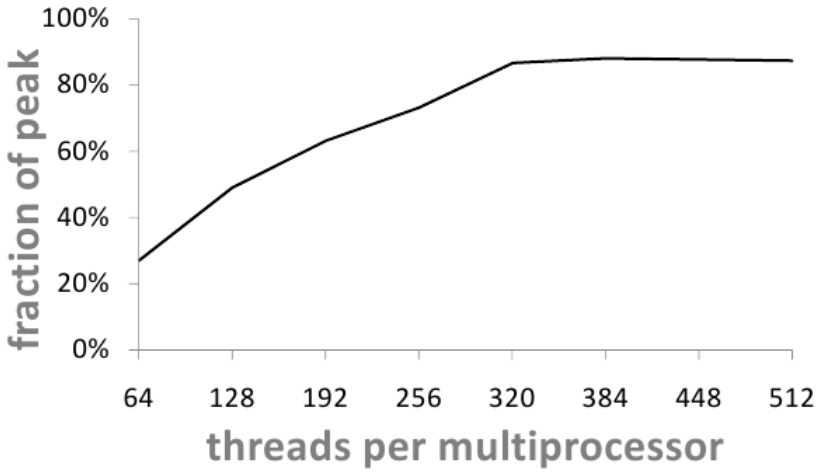


Figura 6.12: Frazione del picco di performance ottenuta al variare del numero di thread per l'Algoritmo 1.3 [50].

```
__global__ void memcpy( float2 *dst, float2 *src )
{
    int iblock = blockIdx.x
               + __mul24( blockIdx.y, gridDim.x );
    int index = threadIdx.x
              + __mul24( iblock, blockDim.x * 2 );

    float2 a0 = src[index];
    float2 a1 = src[index+blockDim.x];
    dst[index] = a0;
    dst[index+blockDim.x] = a1;
}
```

Figura 6.13: Algoritmo 1.4 [50].

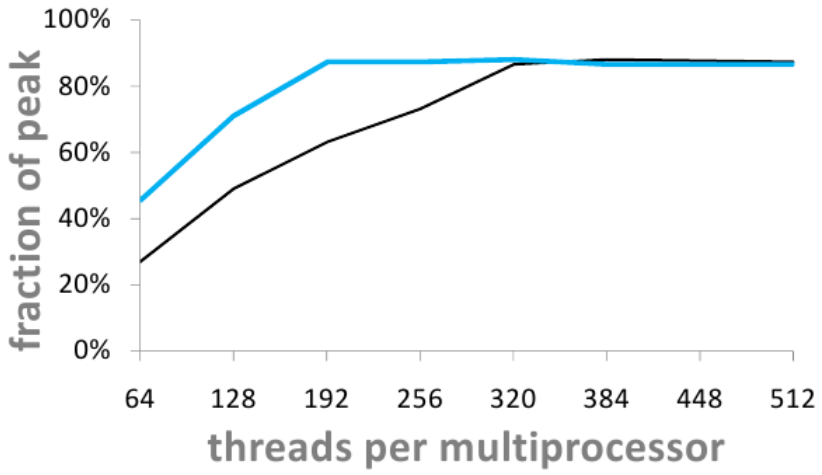


Figura 6.14: Frazione del picco di performance ottenuta al variare del numero di thread per l'Algoritmo 1.4 [50].

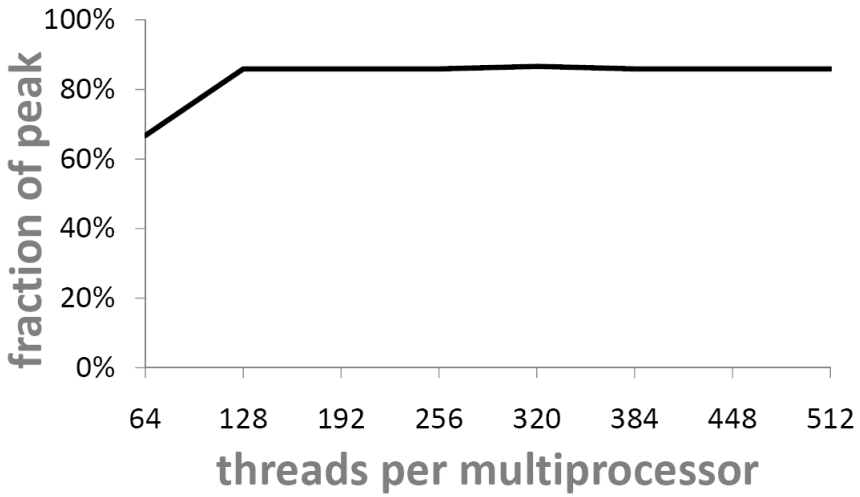


Figura 6.15: Frazione del picco di performance ottenuta al variare del numero di thread per l'Algoritmo 1.5 [50].

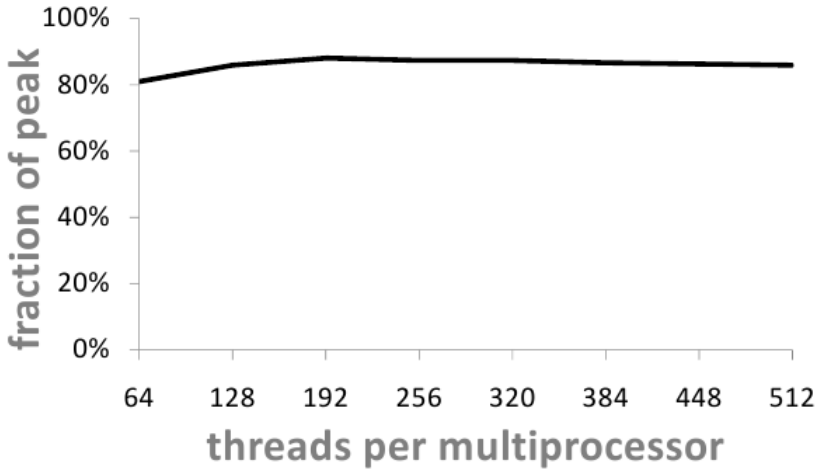


Figura 6.16: Frazione del picco di performance ottenuta al variare del numero di thread per l'Algoritmo 1.6 [50].

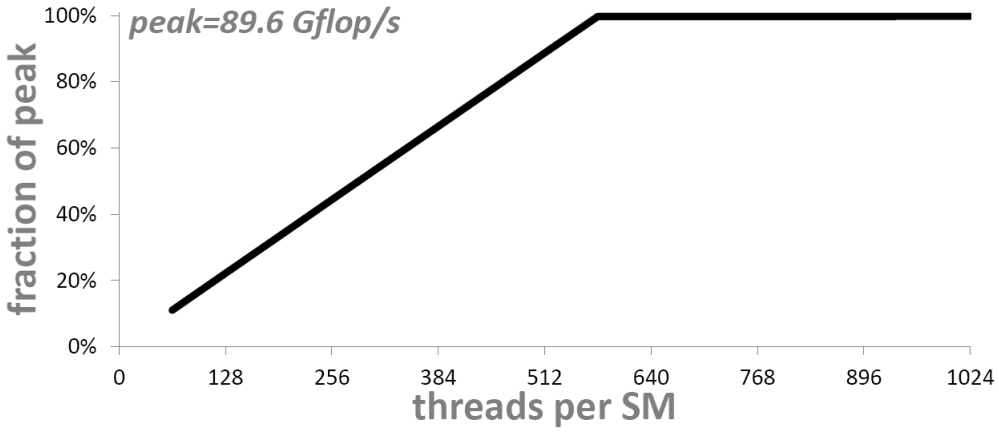


Figura 6.17: Frazione del picco di performance ottenuta al variare del numero di thread per l'Algoritmo 2.1 [47].

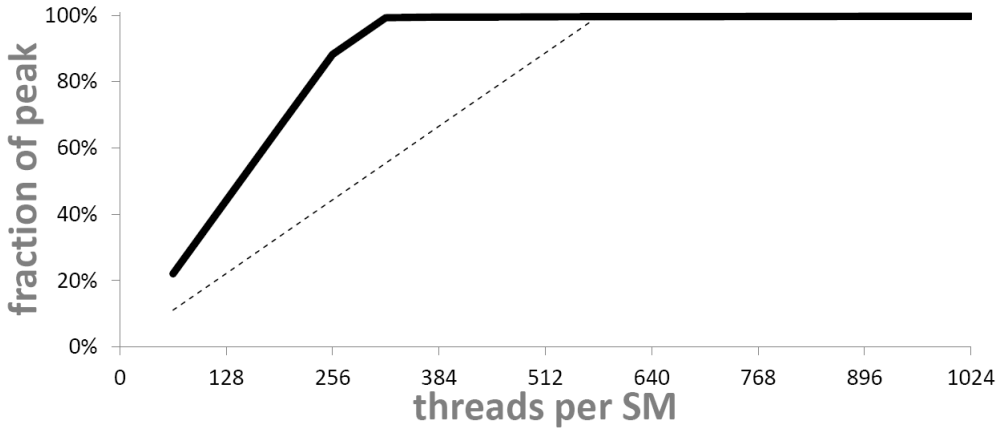


Figura 6.18: Frazione del picco di performance ottenuta al variare del numero di thread per l'Algoritmo 2.2 [47].

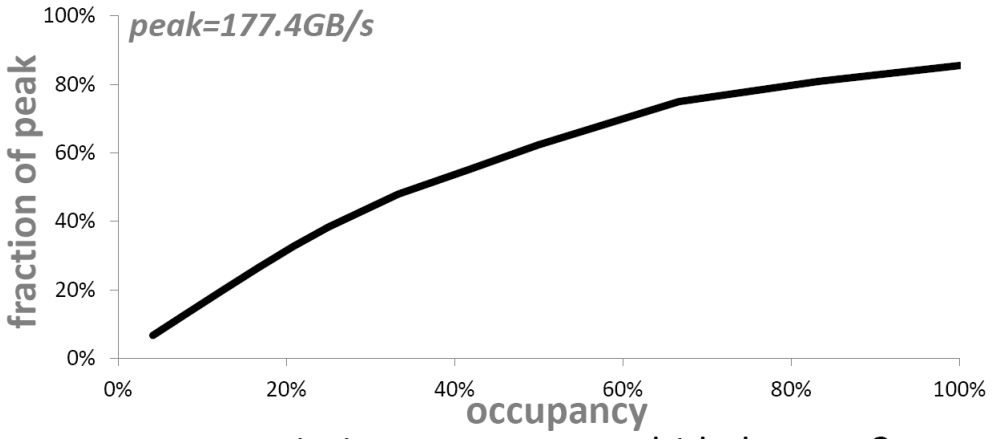


Figura 6.19: Frazione del picco di performance ottenuta al variare del numero di thread per l'Algoritmo 2.3 [47].

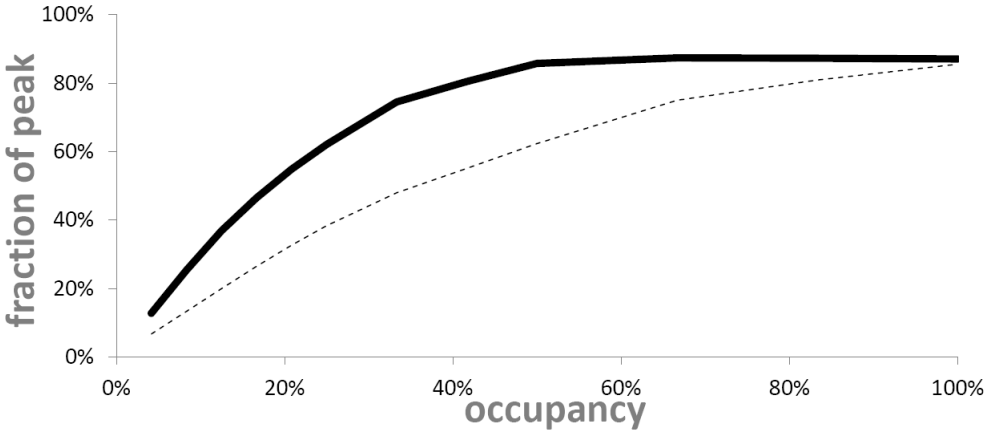


Figura 6.20: Frazione del picco di performance ottenuta al variare del numero di thread per l'Algoritmo 2.4 [47].

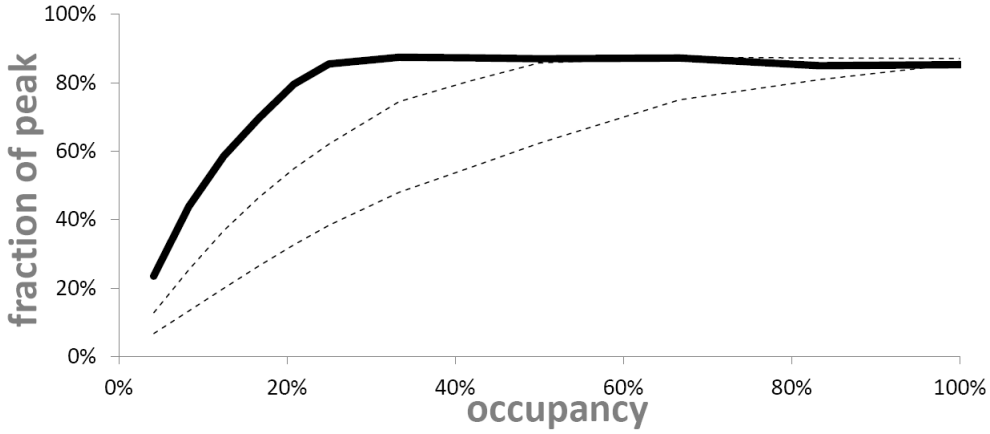


Figura 6.21: FFrazione del picco di performance ottenuta al variare del numero di thread per l'Algoritmo 2.5 [47].

Capitolo 7

Conclusioni

Questo lavoro di tesi si colloca nell'ambito del **calcolo parallelo** per inquadrare le **GPU** in tale contesto e ponendo le basi per uno studio sistematico delle prestazioni degli algoritmi su questo tipo di architetture.
Negli ultimi anni, i risultati sperimentali hanno indotto le case produttrici ad investire risorse cospicue nel miglioramento della tecnologia hardware e software delle GPU, come d'altra parte è sempre avvenuto nella storia delle architetture degli elaboratori (il capitolo 2 ha mostrato come la stessa introduzione del parallelismo sia stata una delle risposte alle speranze (di grandezza) riposte nel calcolo elettronico, che oggi sono diventate realtà).
Dunque le GPU, se pure non si possa dire che rappresentino *il* futuro, ne fanno certamente parte, forti di un approccio alla implementazione di algoritmi che si presta molto bene alla risoluzione di classi di problemi caratterizzate da un alto grado di concorrenza, come evidenziato nella sezione 5.3.
Oltre all'evidente ruolo che già svolgono egregiamente e che verosimilmente continueranno a svolgere di 'accompagnatori' della CPU, esse però rappresentano un'interessante novità progettuale che non mancherà di influenzare anche altre realtà architetturali.
Il **criterio di ottimalità** (sezione 5.3) che abbiamo introdotto, raccoglie le esperienze maturate dai ricercatori nell'utilizzo delle GPU, aiuta a comprendere in quale direzione sta andando la progettazione di alogoritmi innovativi e introduce alla possibilità di studiare le prestazioni delle applicazioni isolan-

do il concetto di **occupazione** (sezione 5.3), dallo sfruttamento dei diversi livelli di parallelismo (compreso quello a livello di istruzione, descritto nella sezione 3.2.1), e da altri parametri su cui si può ulteriormente intervenire per ottimizzare la prestazione del software, e che contribuiscono a definire il concetto di **overhead** (le cui implicazioni sono ben note classicamente e sono approfonditamente descritte nel capitolo 4.4).
In particolare abbiamo ricavato formalmente come non sia strettamente necessario *massimizzare l'occupazione* per mirare ad un'alta performance: questa osservazione era già emersa dagli esperimenti condotti al Dipartimento di Informatica dell'Università della California a Berkeley, come mostrato in sezione 6.2. Le implicazioni di una tale formalizzazione sono tutt'altro che banali. In particolare sottolineiamone due:

1. Una delle conseguenze più ovvie di un'alta occupazione è la riduzione dello spazio di registro a disposizione dei singoli thread e della shared memory allocata per i blocchi, come si deduce dalla descrizione dei meccanismi di allocazione della memoria del capitolo 5. Questo comporta che, per avere un'alta occupazione l'algoritmo debba ridurre l'utilizzo di questa memoria, con conseguenze anche molto pesanti sule latenze di accesso (il terzo tipo di memoria utilizzabile è notevolmente più lento dei due ora citati), altrimenti, bisogna prevedere un'occupazione bassa sfruttando i vantaggi di una gestione più 'economica 'della memoria. Sapere che massimizzare l'occupazione non deve essere ritenuto una *condicio sine qua non*, non solo alleggerisce l'impresa di chi sviluppa il software ma lo incoraggia a trovare altre strade di miglioramento.

2. Nel momento in cui nelle GPU possono essere eseguiti concorrentemente più kernel, come avviene nelle recenti architetture Fermi di NVIDIA (descritte in sezione 6.1.2), quindi si realizza il parallelismo anche a livello di applicazioni, diventa imperativo evitare che un solo kernel si impadronisca di tutta la risorsa senza che questo sia davvero necessario. Dunque, individuare il migliore compromesso tra buona occupazione, buona gestione della memoria e contenimento dell'overhead deve essere e sarà l'obiettivo principale degli sviluppatori. Notiamo che l'idea di

moltiplicare i kernel attivi sulla stessa GPU è forse stata resa possibile proprio dall'osservazione empirica che non fosse necessario dedicare tutta l'architettura ad un singolo problema alla volta per ottenerne un buon guadagno.

Questo approccio alla programmazione delle GPU che le mette di fatto a disposizione dell'HPC, le inserisce di diritto nel panorama delle possibilità di cui si avvale lo sforzo internazionale di superare la prossima frontiera nelle prestazioni degli ambienti di **supercalcolo**: l'**exascale**.

7.1 Futuro Prossimo

[61, 62, 63] Negli ultimi 20 anni, la comunità *open source* ha prodotto molti software da cui dipende l'HPC internazionale. Sono stati investiti milioni di dollari ed anni di lavoro. Ma, sebbene questi investimenti siano notevoli, c'è una perdita rispetto alla possibile produttività, dovuta alla mancanza di pianificazione, coordinazione e integrazione della tecnologia necessaria a far lavorare insieme regolarmente ed efficientemente componenti di provenienza diversa. É impossibile allora che procedendo in questo modo si realizzi il software necessario a supportare sia il parallelismo senza precedenti che richiederà il calcolo peta/esascale su milioni di core, sia la flessibilità necessaria per sfruttare nuovi modelli hardware, come le GPU.
Con queste osservazioni in mente, all'inizio del 2009, un folto gruppo di collaboratori provenienti da tutto il mondo ha dato vita all'**International Exascale Software Project**, o IESP [61], per pianificare ed organizzare quanto necessario a compiere questo balzo in avanti nelle prestazioni dei sistemi di calcolo. Tale progetto viene finanziato sia da enti governativi (Stati Uniti, Europa e Giappone) sia dalle industrie, e descrive la sua missione così [61]:

"Lo scopo dell'IESP è quello di guidare lo sviluppo per fornire maggiore potenza alla ricerca nel campo dell'ingegneria e delle scienze [...] fino al 2020, stabilendo un piano per (1) un ambiente di calcolo comune e di alta qualità

Figura 7.1: Logo dell'International Exascale Software Project.

per i sistemi petascale/esascale e (2) catalizzare, coordinare e sostenere lo sforzo della comunità internazionale [...] per creare questo ambiente quanto più velocemente possibile."

Per alimentare il progetto é stata quindi scritta (e viene regolarmente aggiornata) una precisa **roadmap**[1]: la metafora della mappa serve a sottolineare che questo documento intende essere di riferimento per la comunità globale del calcolo scientifico. É il prodotto di diversi passi compiuti per indirizzare l'impegno degli interessati, e nasce dalla convergenza di tre fattori:

1. la necessità impellente di una sempre maggiore potenza di calcolo della ricerca in campi di importanza vitale per l'umanità,

2. la riconosciuta inadeguatezza dell'attuale infrastruttura software, in tutte le sue componenti, al supporto di tale sfida,

3. la quasi completa mancanza di pianificazione e coordinazione della comunità scientifica per superare questi ostacoli e adeguare tale infrastruttura.

Adeguare però significa riprogettare completamente e sostituire sistemi operativi, modelli di programmazione, librerie e strumenti vari da cui l'*high-end computing* dipende. Tutto ciò deve anche tenere conto della complessa rete di interdipendenze ed effetti collaterali che esiste tra le diverse componenti

[1]Letteralmente, 'mappa stradale'

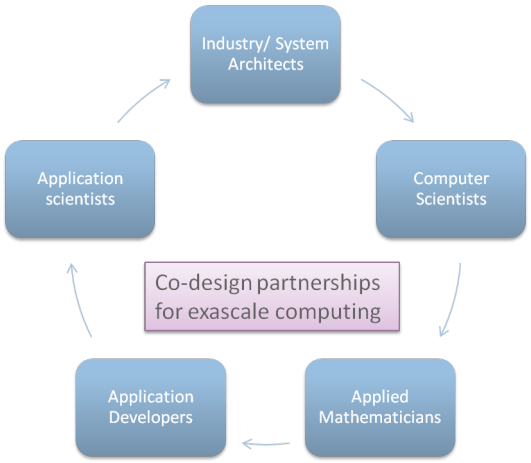


Figura 7.2: Schematizzazione del significato di *co-design*: ricercatori di ogni campo delle scienze devono essere coinvolti per la riuscita del progetto [65].

software: è quindi necessario un alto grado di coordinazione e *collaborazione*[2], perchè sbagliare nell'identificare punti critici o potenziali conflitti tra gli ambienti software e rimandare occasioni di integrazione ritarderebbe il progresso di tutti, rendendo vana buona parte del lavoro e dell'investimento. É poi chiaro che questo progetto deve avere respiro internazionale: porta vantaggio ad ogni la comunità scientifica e soprattutto a quelle che potranno lavorare insieme su problemi di importanza globale. D'altra parte la fattibilità di un compito di tali dimensioni, come quello di ricreare in pochi anni le basi della scienza computazionale per raggiungere l'*extreme-scale computing* è decisamente troppo per una nazione sola o poche nazioni.

La roadmap prevede che piattaforme di esascale computing facciano la loro comparsa tra il 2018 e il 2020, e chiama l'insieme di tutto il software integrato per lo scopo **eXtreme-scale/exascale software Stack**, o **X-stack**.
L'idea è che l'X-stack rappresenti una convergenza delle tendenze tecnologiche imposte da mezzo secolo di sviluppo di algoritmi ed applicazioni.
Tra le tendenze emerse dagli studi del progetto IESP, insieme alla ricerca su *affidabilità*, *risparmio energetico*, *capacità di memoria*, *efficienza* e *programmabilità*, compare la textbfconcorrenza. Infatti, sebbene ci si aspetti che la legge di Moore (vedi capitolo 2) sulla densità dei transistor continui

[2]Si parla di **co-design** [64, 63, 65]

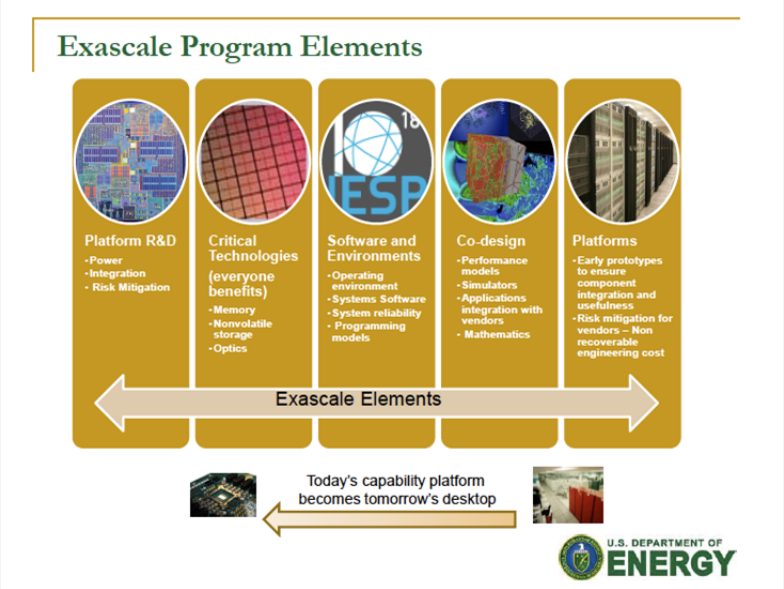


Figura 7.3: Aspetti del programma dell'IESP [65].

a valere per ancora i prossimi dieci anni, per ridurre il consumo di energia, tale densità potrebbe essere frenata o addirittura ridotta. Dunque non c'è dubbio che i sistemi esascale su cui vivrà X-stack saranno composti di centinaia di milioni di ALU, su ognuna delle quali verranno eseguiti diversi thread per coprire le latenze di memoria e di rete: le applicazioni del futuro allora conterranno anche *bilioni* di thread.
Allora possiamo certamente dire che i nuovi modelli di programmazione dovranno facilitare l'utilizzo della concorrenza e la realizzazione di applicazioni *scalabili* (vedi sezione 4.4), ovvero con un ridotto **overhead**.
I sistemi saranno generalmente eterogenei, perchè è ormai chiaro che l'**eterogeneità** offre l'opportunità di sfruttare le potenzialità di sottosistemi specializzati (soprattutto GPU), a vantaggio della prestazione globale di ogni applicazione. Le applicazioni stesse stanno diventando eterogenee, mirando alla soluzione di problemi trasversali a diverse discipline.

Anche la **gerarchia di memoria** subirà modifiche e miglioramenti (influenzando a sua volta i modelli di programmazione), per limitare i colli di bottiglia che oggi limitano moltissimo la performance dei diversi sistemi di calcolo.

Questa tesi è stata sviluppata con la consapevolezza di queste necessità, e con uno sguardo proprio al futuro delle strutture delle architetture e dei modelli di programmazione.

Elenco delle figure

Bibliografia

[1] J. von Neumann, *First draft of a report on the EDVAC*, Technical report, University of Pennsylvania, 1945.

[2] M.J. Flynn, *Very High-Speed Computing Systems*, Proceedings of the IEEE, vol. 54, p.p. 1901-1909, December 1966.

[3] M.J. Flynn, *Some Computer Organizations and Their Effectiveness*, IEEE Transactions on Computers, vol. c-21, n. 9, 1972.

[4] J. Dongarra, I. Foster, G. Fox, W. Gropp, K. Kennedy, L. Torczon, A. White, *The Sourcebook of Parallel Computing*, Morgan-Kaufmann, 2002.

[5] http://graphics.stanford.edu/projects/brookgpu/

[6] E. Brooks, *The attack of the killer micros*, Teraflop Computing Panel Discussion at Supercomputing, Reno, NV, 1989.

[7] H.H. Goldstine, *The Computer from Pascal to von Neumann*, Princeton University Press, 1972.

[8] B.A. Toole, *Ada, the enchantress of numbers: Poetical Science*, ebook ed. Clarity, 2010

[9] B. Pascal, *Lettera dedicatoria al Monsignor Cancelliere*, 1645.

[10] L.F. Menabrea, *Sketch of the Analytical Engine Invented by Charles Babbage. With notes upon the Memoir by the Translator*, trad. Ada Augusta Lovelace, 1843.

[11] D.A. Patterson, J.L. Hennessy, *Struttura e progetto dei calcolatori*

[12] A.S. Tanenbaum, *Architettura dei calcolator: un approccio strutturale*, quinta edizione, Prentice Hall, 2006.

[13] G.M. Amdahl, G.A. Blaauw, F.P. Brooks Jr., *Architecture of the IBM System/360*, IBM Journal of Research and Development, pp.21-36, 2000.

[14] http://www.nvidia.com

[15] *NVIDIA's Next Generation CUDA^{TM}Compute Architecture: Fermi^{TM}. Whitepapers.* Nvidia Corporation, 2009.

[16] G.E. Moore *Cramming more components onto integrated circuits*, Electronics magazine, Volume 38, Number 8, aprile 1965.

[17] *Excerpts from A Conversation with Gordon Moore: Moore's Law*, Intel Corporation, 2006.

[18] www.Top500.org

[19] J. Dongarra, *Performance of Various Computers Using Standard Linear Equations Software*, University of Manchester, 2011.

[20] http://www.netlib.org/benchmark/hpl/

[21] A.M. Turing, *On computable numbers, with an application to the Entscheidungsproblem*, Proceedings of the London Mathematical Society, Series 2, 42, pp.230-265, 1937.

[22] Burattini E., Tamburrini G. *Una nota su modelli cognitivi e complessità di calcolo*, Atti del Convegno Reasoning: the logic and psychologic perpspectives, Padova Maggio 1999, Ed. Giarretta et al., ediz.Giunti, 1999.

[23] S. Chatterjee, J. Prins, *PRAM Algorithms*, note del corso di Parallel Computing, Cornell University, New York, USA, 2011.

[24] S. Rao, *The PRAM*, note del corso di Foundations of Parallel and Distributed Systems, Berkeley University, California, USA, 2010.

[25] C.H. Papadimitriou, K. Chatterjee, *The PRAM*, note del corso di Foundations of Parallel and Network Computing, Berkeley University, California, USA, 2003.

[26] P.B. Gibbons, *A More practical PRAM Model*, in Proc. SPAA, pp.158-168, 1989.

[27] S. Fortune, J. Wyllie, *Parallelism in Random Access Machines*,in Proc. STOC, pp.114-118, 1978.

[28] M. Davis, R. Sigal, E. J. Weyuker, *Computability, Complexity, and Languages*, 2nd edition, Morgan Kaufmann, 1994.

[29] M. Sipser, *Introduction to the Theory of Computation*, 2nd edition, Course Technology, 2005.

[30] G.M. Amdahl, *Validity of the single-processor approach to achieving large scale computing capabilities*, in AFIPS Conference Proceedings, vol. 30 (Atlantic City, N.J.. Apr. 18-20). AFIPS Press, Reston. Va., pp. 483-485, 1967.

[31] H.P. Flatt, K. Kennedy, *Performance of Parallel Processors*, presented at Parallel Computing, pp.1-20, 1989.

[32] D.J. Kuck, *On the speedup and cost of parallel computation*, in Proceedings of The Complexity of Computational Problem Solving, University of Queensland Press, St. Lucia, Australia, 1976.

[33] G.C. Fox, M.A. Johnson, G.A. Lyzenga, S.W. Otto, J.K. Salmon, D.W. Walker, *Solving Problems on Concurrent Processors, Volume I: General Techniques and Regular Problems*, Prentice-Hall, Englewood Cliffs, NJ, 1988.

[34] G.C. Fox, *What Have We Learnt from Using Real Parallel Machines to Solve Real Problems?*, Proceedings of the Third Conference on Hypercube Concurrent Computers and Applications, pp 897-95, 1988.

[35] J.L. Gustafson, *Reevaluating Amdahl's Law*, CACM, 31(5), pp. 532-533, 1988.

[36] J.L. Gustafson, G.R. Montry, R.E. Benner, *Reevaluating Amdahl's Law*, CACM, 31(5), pp. 532-533, 1988.

[37] A. Grama, A. Gupta, Eui-Hong (Sam) Han, V. Kumar, *Parallel Algorithm Scalability Issues in Petaflops Architectures*, Ultrascale Computing, 2000.

[38] A.H. Karp, H.P. Flatt, *Measuring Parallel Processor Performance*, Communication of the ACM 33 (5): 539-543, 1990.

[39] G. Laccetti, M. Lapegna, D. Romano, V. Mele, *Synchronization and Data Caching for Numerical Linear Algebra Algorithms in Distributed and Grid Computing Environments*, in Computing Frontiers 2009, ACM press 2009.

[40] G.Laccetti, M.Lapegna, V.Mele, D.Romano, *Multilevel Algorithms for Multidimensional Integration in High Performance Computing Environments*, in: Proceedings of the Final Workshop of Grid Projects funded by PON Ricerca 2000-2006 avviso 1575, Catania: consorzio COMETA, p. 227-232, 2009.

[41] G.Laccetti, M.Lapegna, V.Mele, D.Romano, *Some Performance Issues on Linear Algebra Algorithms in Distributed and Grid Computing Environments*, Advances in Computer Science and Engineering, Pushpa Publishing House, vol. 6, no. 2, pp. 181-197, 2011.

[42] L. D'Amore, L. Marcellino, V. Mele, D. Romano, *Deconvolution of 3D Fluorescence Microscopy Images using Graphics Processing Units*, Workshop on Models, Algorithms and Methodologies for Hierarchical Parallelism in new HPC Systems, PPAM 2011.

[43] A. Murli, *Lezioni di Calcolo Parallelo*, Liguori Editore, 2006.

[44] A. Murli, *Matematica Numerica: metodi, algoritmi e software - Parte Prima*, Liguori Editore, 2007.

[45] V. Volkov, J.W. Demmel, *Benchmarking GPUs to tune dense linear algebra*, ACM/IEEE Conference on Supercomputing, 2008.

[46] J.W. Demmel et al., Note dal corso di Applications of Parallel Computers, Berkeley University, California, USA, 2011.

[47] V. Volkov, *Better performance at lower occupancy*, GPU Technology Conference, 2010.

[48] A. Gupta, V. Kumar, *The Scalability of FFT on Parallel Computers*, IEEE Transactions on Parallel and Distributed Systems, Volume 4, Number 8, pp 922-932, August 1993.

[49] A. Gupta, V. Kumar, *Performance Properties of Large Scale Parallel Systems*, Journal of Parallel and Distributed Computing, Volume 19, Number 3, Novemeber 1993.

[50] V. Volkov, *Use registers and multiple outputs per thread on GPU*, International Workshop on Parallel Matrix Algorithms and Applications, 2010.

[51] M. Papadopoulou, M. Sadooghi-Alvandi, H. Wong, *Micro-benchmarking the GT200 GPU*, Technical report, Computer Group, ECE, University of Toronto, 2009.

[52] M. Harris, *Optimizing Parallel Reduction in CUDA*, Nvidia, Tech. Rep., 2007.

[53] *www.gpgpu.org*, sito fondato da M. Harris nel 2002.

[54] J. Owens, D. Luebke, N. Govindaraju, M. Harris, J. Kruger, A. Lefohn, T. Lefohn, *A Survey of General-Purpose Computation on Graphics Hardware*, Computer Graphics Forum. v.26, n. 1, 80–113, 2007.

[55] J. Sanders, E. Kandrot, *CUDA by example*, Addison-Wesley 2011.

[56] D.B. Kirk, W.W. Hwu, *Programming massive parallel processors*, Morgan-Kaufmann 2010.

[57] Y. Zhang, J.D. Owens, *A quantitative performance analysis model for GPU architectures*, in proceedings of the 17th IEEE International Symposium on High-Performance Computer Architecture, pages 382-393, February 2011.

[58] S. Gupta, M.R. Babu, *Performance Analysis of GPU compared to Single-core and Multi-core CPU for Natural Language Applications*, International Journal of Advanced Computer Science and Applications, Vol. 2, No. 5, 2011.

[59] S. Tomov, R. Nath, H. Ltaief, J. Dongarra, *Dense linear algebra solvers for multicore with GPU accelerators*, in proceedings of IPDPS Workshops, 1-8, 2010.

[60] A. Buttari, J. Dongarra, J. Kurzak, J.Langou, P. Luszczek, S. Tomov, *The Impact of Multicore on Math Software*, in proceedings of PARA, 1-10, 2006.

[61] J. Dongarra, P. Beckman, P. Aerts, F. Cappello, T. Lippert, S. Matsuoka, P. Messina, T. Moore, R. Stevens, A. Trefethen, M. Valero et al. *The International Exascale Software Roadmap*, International Journal of High Performance Computer Applications, , Volume 25, Number 1, 2011.

[62] P. Messina, *Workshop Series on Science Grand Challenges Enabled by Extreme Scale Computing*, presentazione a IESP Workshop, Saclay, France, giugno 2009.

[63] P. Messina, *Updates on US Exascale Ac1vi1es*, presentazione a IESP Meeting, Colonia, Germania, ottobre 2011.

[64] R. Wisniewski, *Things to think about: Codesign and IP*, IESP Meeting, San Francisco, CA, USA, aprile 2011.

[65] P. Messina, *Challenges and Opportunities in Exascale Computing*, presentazione a 13th IEEE International Conference on High Performance Computing and Communications (HPCC'11), Banff, Canada, settembre 2011