



Cineca
TRAINING
High Performance
Computing 2017

Code Optimization part II

N. Sanna (n.sanna@cineca.it) V. Ruggiero (v.ruggiero@cineca.it)
Roma, 13 July 2017

SuperComputing Applications and Innovation Department

Outline

Compilatori e ottimizzazione

Floating Point Computing

Makefile

Architectures

Power Performance



Linguaggi

- ▶ Esiste un'infinità di linguaggi differenti
- ▶ <http://foldoc.org/contents/language.html>

20-GATE; 2.PAK; 473L Query; 5lforth; A#; A-0; a1; a56;
 Abbreviated Test Language for Avionics Systems; ABC;
 ABC ALGOL; ABCL/1; ABCL/c+; ABCL/R; ABCL/R2; ABLE;
 ABSET; abstract machine; Abstract Machine Notation;
 abstract syntax; Abstract Syntax Notation 1;
 Abstract-Type and Scheme-Definition Language; ABSYS;
 Accent; Acceptance, Test Or Launch Language; Access;
 ACOM; ACOS; ACT++; Act1; Act2; Act3; Actalk; ACT ONE;
 Actor; Actra; Actus; Ada; Ada++; Ada 83; Ada 95; Ada 9X;
 Ada/Ed; Ada-0; Adaplan; Adaplex; ADAPT; Adaptive Simulated
 Annealing; Ada Semantic Interface Specification;
 Ada Software Repository; ADD 1 TO COBOL GIVING COBOL;
 ADELE; ADES; ADL; AdLog; ADM; Advanced Function Presentation;
 Advantage Gen; Adventure Definition Language; ADVSYS; Aeolus;
 AFAC; AFP; AGORA; A Hardware Programming Language; AIDA;
 AIR MATERIAL COMmand compiler; ALADIN; ALAM; A-language;
 A Language Encouraging Program Hierarchy; A Language for Attributed ..

Linguaggi

▶ Linguaggi interpretati

- ▶ il linguaggio viene "tradotto" statement per statement dall'interprete durante l'esecuzione
- ▶ impossibili ottimizzazioni tra differenti statement
- ▶ permette di verificare la correttezza statement dopo statement seguendo il flusso d'istruzioni (migliore gestione degli errori semantici)
- ▶ Esempi: Python, PHP, Java, MATLAB,...

▶ Linguaggi compilati

- ▶ il programma viene "tradotto" dal compilatore prima dell'esecuzione
- ▶ possibili ottimizzazioni tra differenti statement
- ▶ gestione minimale degli errori semantici
- ▶ Fortran, C, C++

Le CPU

- ▶ È composta di:
 - ▶ registri (operandi delle istruzioni)
 - ▶ unità funzionali (eseguono le istruzioni)
- ▶ Unità funzionali:
 - ▶ aritmetica intera
 - ▶ operazioni logiche bitwise
 - ▶ aritmetica floating-point
 - ▶ calcolo di indirizzi
 - ▶ lettura e scrittura in memoria (load & store)
 - ▶ previsione ed esecuzione di "salti" (branch) nel flusso di esecuzione

Le CPU

- ▶ RISC: Reduced Instruction Set CPU (e.g. IBM Power)
 - ▶ istruzioni semplici
 - ▶ formato regolare delle istruzioni
 - ▶ decodifica ed esecuzione delle istruzioni semplificata
 - ▶ codice macchina molto "verboso"
- ▶ CISC: Complex Instruction Set CPU (e. g. Intel x86)
 - ▶ istruzioni di semantica "ricca"
 - ▶ formato irregolare delle istruzioni
 - ▶ decodifica ed esecuzione delle istruzioni complicata
 - ▶ codice macchina molto "compatto"
- ▶ Differenza non più rilevante quanto a prestazioni: le CPU CISC di oggi convertono le istruzioni in micro operazioni RISC-like

Statement & Istruzione

- ▶ Lo statement è una riga di un codice
 - ▶ `write(*,*)`
 - ▶ `printf();`
 - ▶ `do i = 1,n`
 - ▶ `x(i) = y(i) + 1`
- ▶ L'istruzione è l'operazione che la CPU "realmente" fa, nelle macchine RISC sono essenzialmente:
 - ▶ `load/store`
 - ▶ `somma/prodotto`
 - ▶ `operazioni tra bit ...`
- ▶ Uno statement può tradursi in una sola, poche o tante istruzioni



Architettura vs. Implementazione

- ▶ Architettura:
 - ▶ set di istruzioni
 - ▶ registri architetturali interi, floating point e di stato
- ▶ Implementazione
 - ▶ registri fisici ($2.5 \div 20 \times$ registri architetturali)
 - ▶ frequenza di clock e tempo di esecuzione delle istruzioni
 - ▶ numero di unità funzionali
 - ▶ dimensione, numero, caratteristiche delle cache
 - ▶ Out Of Order execution, Simultaneous Multi-Threading
- ▶ Una architettura, più implementazioni:
 - ▶ Power: Power4, Power5, Power6, ...
 - ▶ x86: Pentium III, Pentium 4, Xeon, Pentium M, Pentium D, Core, Core2, Athlon, Opteron, ...
 - ▶ prestazioni differenti
 - ▶ "regole" diverse per ottenere alte prestazioni



Il compilatore

- ▶ Traduce il codice sorgente in codice macchina
- ▶ Rifiuta codici sintatticamente errati
- ▶ Segnala (alcuni) potenziali problemi semantici
- ▶ Può tentare di ottimizzare il codice
 - ▶ ottimizzazioni indipendenti dal linguaggio
 - ▶ ottimizzazioni dipendenti dal linguaggio
 - ▶ ottimizzazioni dipendenti dalla CPU
 - ▶ ottimizzazioni dell'uso della memoria e della cache
- ▶ È uno strumento
 - ▶ potente: può risparmiare lavoro al programmatore
 - ▶ complesso: a volte può fare cose sorprendenti o controproducenti
 - ▶ limitato: è un sistema esperto, ma non ha l'intelligenza di un essere umano, non può capire pienamente il codice



HPC e Linguaggi

- ▶ Linguaggi ideali per l'High Performance Computing?
 - ▶ adatti all'implementazione di algoritmi "scientifici"
 - ▶ devono permettere elevati livelli di ottimizzazione
 - ▶ integrandosi nelle recenti architetture di supercalcolo
- ▶ Quali sono?
 - ▶ Fortran/C/C++, ci limitiamo a Fortran e C, cenno specifici a C++
 - ▶ il Python è in ascesa, ma per le parti computazionalmente intensive si usa appoggiarsi a C o Fortran
- ▶ I compilatori più usati per l'HPC (non sono molti)
 - ▶ GNU (gfortran, gcc, g++): "libero"
 - ▶ Intel (ifort, icc, icpc)
 - ▶ IBM (xlf, xlc, xlC)
 - ▶ Portland Group (pgf90, pgcc, pgCC)
 - ▶ PathScale, Oracle/Solaris, Fujitsu, Nag, Microsoft,...
- ▶ Linux, Windows or Mac OS X?
 - ▶ discorso complesso, ma la grande maggioranza delle architetture di supercalcolo ad oggi gira su piattaforma Linux

Il compilatore: schema

- ▶ Composto da differenti blocchi
- ▶ Front-end
 - ▶ Controllo sintassi
 - ▶ Analisi semantica
 - ▶ Traduzione rappresentazione intermedia
- ▶ Back-end
 - ▶ Ottimizzazione
 - ▶ Generazione codice eseguibile
- ▶ In genere
 - ▶ Front-end: dipende dal linguaggio
 - ▶ Back-end: indipendente dal linguaggio



Rappresentazione Intermedia

- ▶ Rappresenta gli stessi calcoli del codice di alto livello
 - ▶ in una forma più adatta per l'analisi
 - ▶ include calcoli non presenti nel sorgente, come il calcolo degli indirizzi

▶ Rappresentazione Intermedia

```

A: t1 = j t2 = n t3 = (t1<t2)
   jmp (B) t3
   jmp (C) TRUE
B: t4 = k t5 = j t6 = t5*2
   t7 t4+t6
   k=t7
   t8=j t9=t8*2
   m=t9
   t10=j t11=t10+1
   j=t11
   jmp(A) TRUE

```

▶ C

```

while (j<n) {
k = k+j*2;
m = j*2;
j++
}

```

C:

Assembler

- ▶ La rappresentazione intermedia, dopo l'ottimizzazione, è tradotta in assembly, e eventualmente ulteriormente ottimizzata

- ▶ Assembly

- ▶ C

```
int add (int a, int b)
{
    return a+b;
}
```

```
int add (int a , int b) {
0: 55          push   %ebp
1: 8b ec       mov    %esp,%ebp
/home/marco/Master10/prog/add.c:2
return a+b;
3: 8b 45 0c    mov    0xc(%ebp),%eax
6: 03 45 08    add   0x8(%ebp),%eax
9: c9         leave
a: c3         ret
b: 90         nop
```



Building di un Programma

- ▶ Creare un eseguibile dai sorgenti è in generale un processo a tre fasi
- ▶ Pre-processing:
 - ▶ ogni sorgente è letto dal pre-processore
 - ▶ sostituire (**#define**) MACROs
 - ▶ inserire codice per gli statement **#include**
 - ▶ inserire o cancellare codice valutando **#ifdef**, **#if ...**
- ▶ Compilazione:
 - ▶ ogni sorgente è tradotto in un codice oggetto
 - ▶ un file oggetto è una collezione organizzata di simboli che si riferiscono a variabili e funzioni definite o usate nel sorgente
- ▶ Linking:
 - ▶ file oggetti sono combinati per costruire il singolo eseguibile finale ed ogni simbolo deve essere risolto
 - ▶ i simboli possono essere definiti nei file oggetto
 - ▶ o disponibili in altri codici oggetti (librerie esterne)

Compilare con GNU gfortran

- ▶ Quando si dà il comando:

```
user@$> gfortran dsp.f90 dsp_test.f90
```

vengono eseguiti automaticamente i tre passi

- ▶ Pre-processing (l'opzione `-E`)

```
user@$> gfortran -E dsp.f90  
user@$> gfortran -E dsp_test.f90
```

- ▶ si ha a stdout il file pre-processato

- ▶ Compilazione dei sorgenti (l'opzione `-c`)

```
user@$> gfortran -c dsp.f90  
user@$> gfortran -c dsp_test.f90
```

- ▶ da ogni sorgente viene prodotto un file oggetto `.o`

Linking con GNU gfortran

- ▶ Linkare oggetti tra di loro

```
user@$> gfortran dsp.o dsp_test.o
```

- ▶ Per risolvere i simboli definiti in librerie esterne specificare:
 - ▶ le librerie da usare (opzione `-l`)
 - ▶ le directory in cui stanno (opzione `-L`)
- ▶ Come linkare `libblas.a` nella cartella `/opt/lib`

```
user@$> gfortran file1.o file2.o -L/opt/lib -ldsp
```

- ▶ Come creare e linkare una libreria statica

```
user@$> gfortran -c dsp.f90
ar curv libdsp.a dsp.o
ranlib libdsp.a
gfortran test_dsp.f90 -L. -ldsp
```

- ▶ `ar` archivia in `libdsp.a` `dsp.o`
- ▶ `ranlib` genera l'indice dell'archivio



Prendere confidenza

- ▶ Per il manuale in linea

```
man gcc
```

riporta le opzioni del compilatore C di GNU e il loro significato

- ▶ Attenzione: `man gfortran` dà solo le opzioni ulteriori rispetto a `gcc`, mentre per gli altri compilatori solitamente i `man` sono replicati nelle parti comuni
- ▶ Il numero di opzioni è notevole e purtroppo differisce da compilatore a compilatore
- ▶ Tipologia di opzioni
 - ▶ linguaggio: sullo standard (o sul dialetto) da seguire
 - ▶ ottimizzazione: argomento delle prossime slide...
 - ▶ target: per l'integrazione con l'architettura di calcolo
 - ▶ debugging: la più importante, `-g`, crea i simboli di debugging necessari per l'uso di debugger
 - ▶ warning: per avere informazioni sulla compilazione

Il compilatore cosa sa fare

- ▶ Esegue trasformazioni del codice come:
 - ▶ Register allocation
 - ▶ Register spilling
 - ▶ Copy propagation
 - ▶ Code motion
 - ▶ Dead and redundant code removal
 - ▶ Common subexpression elimination
 - ▶ Strength reduction
 - ▶ Inlining
 - ▶ Index reordering
 - ▶ Loop unrolling/merging
 - ▶ Loop blocking
 - ▶ ...

- ▶ Lo scopo è massimizzare le prestazioni



Il compilatore cosa non sa fare

- ▶ Analizzare ed ottimizzare globalmente codici molto grossi (a meno di abilitare l'IPO, molto costoso in tempo e risorse)
- ▶ Capire dipendenze tra dati con indirizzamenti indiretti
- ▶ Strength reduction di potenze non intere, o maggiori di $2 \div 4$
- ▶ Common subexpression elimination attraverso chiamate a funzione
- ▶ Unrolling, Merging, Blocking con:
 - ▶ chiamate a procedure
 - ▶ chiamate o statement di Input-Output in mezzo al codice
- ▶ Fare inlining di funzioni se non viene detto esplicitamente
- ▶ Sapere a run-time i valori delle variabili per i quali alcune ottimizzazioni sono inibite

Livelli di ottimizzazione

- ▶ I compilatori forniscono dei livelli di ottimizzazione “predefiniti” utilizzabili con la semplice opzione **-O<n>**
 - ▶ **n** accresce il livello di ottimizzazione, da 0 a 3 (a volte fino a 5)
- ▶ Fortran IBM:
 - ▶ **-O0**: nessuna ottimizzazione (utile insieme a **-g** in debugging)
 - ▶ **-O2**, **-O** : ottimizzazioni locali, compromesso tra velocità di compilazione, ottimizzazione e dimensioni dell'eseguibile
 - ▶ **-O3**: ottimizzazioni memory-intensive, può alterare la semantica del programma (da qui in poi da considerare l'uso si **-qstrict** per evitare risultati errati)
 - ▶ **-O4**: ottimizzazioni aggressive (**-qarch=auto**, **-qhot**, **-qipa**, **-qtune=auto**, **-qcache=auto**, **-qsimd=auto**)
 - ▶ **-O5**: come **-O4** con **-qipa=level=2** aggressiva e lenta
- ▶ Alcuni compilatori hanno **-fast**, che include **On** e altro
- ▶ Per GNU una scelta comune insieme a **-O3** è **-funroll-loops**
- ▶ Attenzione all'ottimizzazione di default: per GNU è **-O0**, di solito è **-O2**



Loop con -O3 (Compiler Intel)

`icc (or ifort) -O3`

- ▶ Automatic vectorization (use of packed SIMD instructions)
- ▶ Loop interchange (for more efficient memory access)
- ▶ Loop unrolling (more instruction level parallelism)
- ▶ Prefetching (for patterns not recognized by h/w prefetcher)
- ▶ Cache blocking (for more reuse of data in cache)
- ▶ Loop peeling (allow for misalignment)
- ▶ Loop versioning (for loop count; data alignment; runtime dependency tests)
- ▶ Memcpy recognition (call Intel's fast memcpy, memset)
- ▶ Loop splitting (facilitate vectorization)
- ▶ Loop fusion (more efficient vectorization)
- ▶ Scalar replacement (reduce array accesses by scalar temps)
- ▶ Loop rerolling (enable vectorization)
- ▶ Loop reversal (handle dependencies)



Dal sorgente all'eseguibile

- ▶ La macchina astratta "vista" a livello di sorgente è molto diversa da quella reale
- ▶ Esempio: prodotto di matrici

```
do j = 1, n
do k = 1, n
do i = 1, n
    c(i, j) = c(i, j) + a(i, k) * b(k, j)
end do
end do
end do
```

- ▶ Il "nocciolo"
 - ▶ carica dalla memoria tre valori
 - ▶ fa una moltiplicazione ed una somma
 - ▶ immagazzina il risultato

MatrixMul: performance

- ▶ Prodotto matrice-matrice, 1024×1024 , doppia precisione
- ▶ Ordine dei loop ottimale per la cache
- ▶ Scritto in Fortran
- ▶ Architetture considerate per le prove (per nodo)
 - ▶ FERMI: IBM Blue Gene/Q system, 16 core single-socket PowerA2 a 1.6 GHz di frequenza
 - ▶ PLX: 2 esa-core XEON 5650 Westmere CPUs 2.40 GHz

FERMI - xlf

PLX - ifort

Opzione	secondi	Mflops
-O0	65.78	32.6
-O2	7.13	301
-O3	0.78	2735
-O4	55.52	38.7
-O5	0.65	3311

Opzione	secondi	MFlops
-O0	8.94	240
-O1	1.41	1514
-O2	0.72	2955
-O3	0.33	6392
-fast	0.32	6623

▶ Perché tanta varietà di risultati? Basta passare da -On a -On+1?

Cosa fa il compilatore: report

- ▶ Cosa accade ai diversi livelli di ottimizzazione?
 - ▶ Perché il compilatore IBM su Fermi al livello **-O4** degrada così le performance?
- ▶ Utilizzare le opzioni di report é un buon modo di capire cosa sta facendo il compilatore
- ▶ Su IBM **-qreport** mostra che per **-O4** l'ottimizzazione prende un percorso completamente diverso dagli altri casi
 - ▶ il compilatore riconosce il pattern del prodotto matrice-matrice e sostituisce le righe di codice con la chiamata a una funzione di libreria BLAS **__x1_dgemm**
 - ▶ che però si rivela molto lenta perché non fa parte delle librerie matematiche ottimizzate da IBM (ESSL)
 - ▶ anche il compilatore Intel fa questo per dgemm, ma invoca le efficienti MKL
- ▶ Aumentando il livello di ottimizzazione, solitamente le performance migliorano
 - ▶ ma è bene testare questo miglioramento per il proprio codice

Uno sguardo all'assembler

- ▶ Esempio datato, utile però per capire

Matrix Multiply inner loop code with -qnoot

38 instructions, 31.4 cycles per iteration

```

__L1:
    lwz    r3,160(SP)
    lwz    r9,STATIC_BSS
    lwz    r4,24(r9)
    subfi  r5,r4,-8
    lwz    r11,40(r9)
    mullw  r6,r4,r11
    lwz    r4,36(r9)
    rlwinm r4,r4,3,0,28
    add    r7,r5,r6
    add    r7,r4,r7
    lfdx   fp1,r3,r7
    lwz    r7,152(SP)
    lwz    r12,0(r9)
    subfi  r10,r12,-8
    lwz    r8,44(r9)
    mullw  r12,r12,r8
    add    r10,r10,r12
    add    r10,r4,r10
    lfdx   fp2,r7,r10
    lwz    r7,156(SP)
    lwz    r10,12(r9)
    subfi  r9,r10,-8
    mullw  r10,r10,r11
    rlwinm r8,r8,3,0,28
    add    r9,r9,r10
    add    r8,r8,r9
    lfdx   fp3,r7,r8
    fmadd  fp1,fp2,fp3,fp1
    add    r5,r5,r6
    add    r4,r4,r5
    stfdx  fp1,r3,r4
    lwz    r4,STATIC_BSS
    lwz    r3,44(r4)
    addi   r3,1(r3)
    stw    r3,44(r4)
    lwz    r3,112(SP)
    addic. r3,r3,-1
    stw    r3,112(SP)
    bgt    __L1

```

L'essenziale

Matrix Multiply inner loop code with -qnoot

necessary instructions

```

__L1:
    lwz    r3,160(SP)
    lwz    r9,STATIC_BSS
    lwz    r4,24(r9)
    subfi  r5,r4,-8
    lwz    r11,40(r9)
    mullw  r6,r4,r11
    lwz    r4,36(r9)
    rlwinm r4,r4,3,0,28
    add    r7,r5,r6
    add    r7,r4,r7
    lfdx   fp1,r3,r7
    lwz    r7,152(SP)
    lwz    r12,0(r9)
    subfi  r10,r12,-8
    lwz    r8,44(r9)
    mullw  r12,r12,r8
    add    r10,r10,r12
    add    r10,r4,r10
    lfdx   fp2,r7,r10

    lwz    r7,156(SP)
    lwz    r10,12(r9)
    subfi  r9,r10,-8
    mullw  r10,r10,r11
    rlwinm r8,r8,3,0,28
    add    r9,r9,r10
    add    r8,r8,r9
    lfdx   fp3,r7,r8
    fmadd  fp1,fp2,fp3,fp1
    add    r5,r5,r6
    add    r4,r4,r5
    stfdx  fp1,r3,r4
    lwz    r4,STATIC_BSS
    lwz    r3,44(r4)
    addi   r3,1(r3)
    stw    r3,44(r4)
    lwz    r3,112(SP)
    addic. r3,r3,-1
    stw    r3,112(SP)
    bgt    __L1

```

Loop control

Matrix Multiply inner loop code with -qnoot

necessary instructions loop control

```

__L1:
    lwz    r3,160(SP)
    lwz    r9,STATIC_BSS
    lwz    r4,24(r9)
    subfi  r5,r4,-8
    lwz    r11,40(r9)
    mullw  r6,r4,r11
    lwz    r4,36(r9)
    rlwinm r4,r4,3,0,28
    add    r7,r5,r6
    add    r7,r4,r7
    lfdx   fp1,r3,r7
    lwz    r7,152(SP)
    lwz    r12,0(r9)
    subfi  r10,r12,-8
    lwz    r8,44(r9)
    mullw  r12,r12,r8
    add    r10,r10,r12
    add    r10,r4,r10
    lfdx   fp2,r7,r10

    lwz    r7,156(SP)
    lwz    r10,12(r9)
    subfi  r9,r10,-8
    mullw  r10,r10,r11
    rlwinm r8,r8,3,0,28
    add    r9,r9,r10
    add    r8,r8,r9
    lfdx   fp3,r7,r8
    fmadd  fp1,fp2,fp3,fp1
    add    r5,r5,r6
    add    r4,r4,r5
    stfdx  fp1,r3,r4
    lwz    r4,STATIC_BSS
    lwz    r3,44(r4)
    addi   r3,1(r3)
    stw    r3,44(r4)
    lwz    r3,112(SP)
    addic. r3,r3,-1
    stw    r3,112(SP)
    bgt    __L1
  
```

L'indirizzamento!

Matrix Multiply inner loop code with -qnoopt

necessary instructions loop control addressing code

```

__L1:
  lwz    r3,160(SP)
  lwz    r9,STATIC_BSS
  lwz    r4,24(r9)
  subfi  r5,r4,-8
  lwz    r11,40(r9)
  mullw  r6,r4,r11
  lwz    r4,36(r9)
  rlwinm r4,r4,3,0,28
  add    r7,r5,r6
  add    r7,r4,r7
  lfdx   fp1,r3,r7
  lwz    r7,152(SP)
  lwz    r12,0(r9)
  subfi  r10,r12,-8
  lwz    r8,44(r9)
  mullw  r12,r12,r8
  add    r10,r10,r12
  add    r10,r4,r10
  lfdx   fp2,r7,r10

  lwz    r7,156(SP)
  lwz    r10,12(r9)
  subfi  r9,r10,-8
  mullw  r10,r10,r11
  rlwinm r8,r8,3,0,28
  add    r9,r9,r10
  add    r8,r8,r9
  lfdx   fp3,r7,r8
  fmadd  fp1,fp2,fp3,fp1
  add    r5,r5,r6
  add    r4,r4,r5
  stfdx  fp1,r3,r4
  lwz    r4,STATIC_BSS
  lwz    r3,44(r4)
  addi   r3,1(r3)
  stw    r3,44(r4)
  lwz    r3,112(SP)
  addic. r3,r3,-1
  stw    r3,112(SP)
  bgt    __L1
  
```

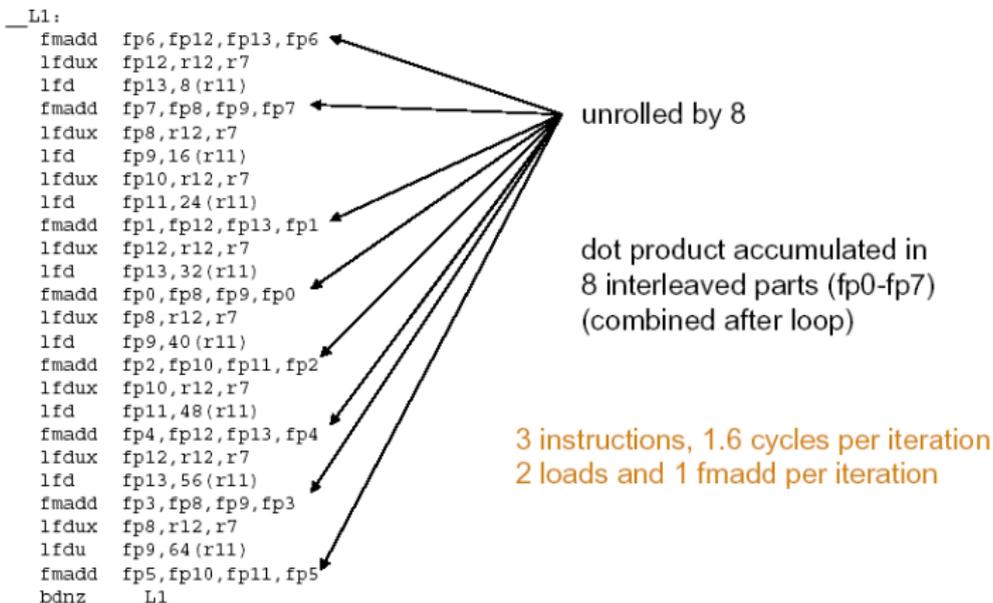


Cosa fare?

- ▶ Le operazioni dominanti sono quelle di conversione indici indirizzo di memoria
- ▶ Osservazioni:
 - ▶ il loop “percorre” la memoria sequenzialmente
 - ▶ gli indirizzi degli elementi successivi sono calcolabili facilmente sommando una costante
 - ▶ sfruttare una conversione indice indirizzo per piú elementi successivi
- ▶ Può essere fatto automaticamente?

Ottimizziamo

Matrix Multiply inner loop code with -O3 -qtune=pwr4



Ottimizziamo di piú

Matrix multiply inner loop code with -O3 -qhot -qtune=pwr4

```

L1:
fmadd fp1, fp4, fp2, fp1
fmadd fp0, fp3, fp5, fp0
lfdx fp2, r29, r9
lfd fp4, 32(r30)
fmadd fp10, fp7, fp28, fp10
fmadd fp7, fp9, fp7, fp8
lfdx fp26, r27, r9
lfd fp25, 8(r29)
fmadd fp31, fp30, fp27, fp31
fmadd fp6, fp11, fp30, fp6
lfd fp5, 8(r27)
lfd fp8, 16(r28)
fmadd fp30, fp4, fp28, fp29
fmadd fp12, fp13, fp11, fp12
lfd fp3, 8(r30)
lfd fp11, 8(r28)
fmadd fp1, fp4, fp9, fp1
fmadd fp0, fp13, fp27, fp0
lfd fp4, 16(r30)
lfd fp13, 24(r30)
fmadd fp10, fp8, fp25, fp10
fmadd fp8, fp2, fp8, fp7
lfdx fp9, r29, r9
lfd fp7, 32(r28)
fmadd fp31, fp11, fp5, fp31
fmadd fp6, fp26, fp11, fp6
lfdx fp11, r27, r9
lfd fp28, 8(r29)
fmadd fp12, fp3, fp26, fp12
fmadd fp29, fp4, fp25, fp30
lfd fp30, -8(r28)
lfd fp27, 8(r27)
bdnz L1
    
```

unroll-and-jam 2x2
inner unroll by 4
interchange "i" and "j" loops

2 instructions, 1.0 cycles per
iteration
balanced: 1 load and 1 fmadd
per iteration



Istruzioni macchina

- ▶ Istruzioni per $c(i, j) = c(i, j) + a(i, k) * b(k, j)$
- ▶ -O0: 24 istruzioni
 - ▶ 3 load/1 store
 - ▶ 1 floating point multiply+add Flop/istruzione 2/24
- ▶ -O2: 9 istruzioni (riuso calcolo indirizzi)
 - ▶ 4 load/1 store
 - ▶ 2 floating point multiply+add Flop/istruzione 4/9
- ▶ -O3: 150 istruzioni (unrolling)
 - ▶ 68 load/ 34 store
 - ▶ 48 floating point multiply+add Flop/istruzione 96/150
- ▶ -O4: 344 istruzioni (unrolling&blocking)
 - ▶ 139 load / 74 store
 - ▶ 100 floating point multiply+add Flop/istruzione 200/344

Fa tutto il compilatore?

- ▶ **-fast** realizza uno speed-up di 30 volte rispetto a -O0 per il caso matrice-matrice (ifort su PLX)
 - ▶ mette in atto una vasta gamma di ottimizzazioni più o meno complicate
- ▶ **Ha senso ottimizzare il codice anche manualmente?**
- ▶ Il compilatore sa fare automaticamente
 - ▶ Dead code removal: per esempio rimuovere un if

```
b = a + 5.0;
if ((a>0.0) && (b<0.0)) {
    .....
}
```

- ▶ Redudant code removal

```
integer, parameter :: c=1.0
f=c*f
```

Loop counters: attenzione ai tipi

- ▶ Usare sempre i tipi corretti
- ▶ Usare un real per l'indice dei loop implica una trasformazione implicita reale \rightarrow intero ...
- ▶ Secondo i recenti standard Fortran si tratta di un vero e proprio errore, ma i compilatori tendono a tollerarlo

```

real :: i, j, k
....
do j=1, n
do k=1, n
do i=1, n
c(i, j)=c(i, j)+a(i, k)*b(k, j)
enddo
enddo
enddo
  
```

Risultati in secondi

Compilazione	integer	real
(PLX) gfortran -O0	9.96	8.37
(PLX) gfortran -O3	0.75	2.63
(PLX) ifort -O0	6.72	8.28
(PLX) ifort -fast	0.33	1.74
(PLX) pgif90 -O0	4.73	4.85
(PLX) pgif90 -fast	0.68	2.30
(FERMI) bgxlf -O0	64.78	104.10
(FERMI) bgxlf -O5	0.64	12.38

Il compilatore può fare tutto?

- ▶ Il compilatore può fare molto . . . ma non è un essere umano
- ▶ È piuttosto facile intralciare il suo lavoro
 - ▶ corpo del loop troppo lungo
 - ▶ loop con i due estremi di iterazione variabili
 - ▶ uso eccessivo di costrutti condizionali (if)
 - ▶ uso eccessivo di puntatori ed indici
 - ▶ uso improprio di variabili intermedie
- ▶ Importante:
 - ▶ due codici semanticamente uguali possono avere prestazioni ben diverse
 - ▶ il compilatore può fare assunzioni erranee ed alterare la semantica

Index reordering

- ▶ Per un loop nest semplice ci pensa il compilatore
 - ▶ a patto di usare un opportuno livello di ottimizzazione

```
do i=1,n
do k=1,n
do j=1,n
  c(i,j) = c(i,j) + a(i,k)*b(k,j)
end do
end do
end do
```

- ▶ Tempi in secondi

Compilazione	j-k-i	i-k-j
(PLX) ifort -O0	6.72	21.8
(PLX) ifort -fast	0.34	0.33

Index reordering / 2

- ▶ Per loop nesting più complicati il compilatore a volte no...
 - ▶ anche al più alto livello di ottimizzazione
 - ▶ conoscere il meccanismo di cache è quindi utile!

```
do jj = 1, n, step
  do kk = 1, n, step
    do ii = 1, n, step
      do j = jj, jj+step-1
        do k = kk, kk+step-1
          do i = ii, ii+step-1
            c(i,j) = c(i,j) + a(i,k)*b(k,j)
          enddo
        enddo
      enddo
    enddo
  enddo
enddo
```

- ▶ Tempi in secondi

Compilazione	j-k-i	i-k-j
(PLX) ifort -O0	10	11.5
(PLX) ifort -fast	1.	2.4

Modularizzazione e cache

```
do i=1,nwax+1
  do k=1,2*nwaz+1
    call diffus (u_1,invRe,qv,rv,sv,K2,i,k,Lu_1)
    call diffus (u_2,invRe,qv,rv,sv,K2,i,k,Lu_2)
    ....
  end do
end do

subroutine diffus (u_n,invRe,qv,rv,sv,K2,i,k,Lu_n)
  do j=2,Ny-1
    Lu_n(i,j,k)=invRe*(2.d0*qv(j-1)*u_n(i,j-1,k)-(2.d0*rv(j-1)
      +K2(i,k))*u_n(i,j,k)+2.d0*sv(j-1)*u_n(i,j+1,k))
  end do
end subroutine
```

- ▶ 5 accessi in memoria contigui in i
- ▶ 3 accessi in memoria contigui in j

Modularizzazione e cache / 2

```
call diffus (u_1,invRe,qv,rv,sv,K2,Lu_1)
call diffus (u_2,invRe,qv,rv,sv,K2,Lu_2)
....

subroutine diffus (u_n,invRe,qv,rv,sv,K2,i,k,Lu_n)
  do k=1,2*nwaz+1
    do j=2,Ny-1
      do i=1,nwax+1
        Lu_n(i,j,k)=invRe*(2.d0*qv(j-1)*u_n(i,j-1,k)-(2.d0*rv(j-1)
          +K2(i,k))*u_n(i,j,k)+2.d0*sv(j-1)*u_n(i,j+1,k))
      end do
    end do
  end do
end subroutine
```

- ▶ Modularizzare così permette di avere l'ordine ottimale dei loop
- ▶ A volte il compilatore trasforma da solo: in questo caso è richiesto l'"inlining"

Inlining

- ▶ Ottimizzazione manuale o effettuata dal compilatore che sostituisce una funzione col suo corpo
 - ▶ elimina il costo della chiamata e potenzialmente l'Instruction Cache
 - ▶ rende più facile l'ottimizzazione interprocedurale
- ▶ In C e C++ la keyword **inline** è un “suggerimento”
- ▶ Non ogni funzione è “inlinable” e in ogni caso dipende dalle capacità del compilatore
 - ▶ oltre che dalle capacità del programmatore
- ▶ Intel (n: 0=disable, 1=secondo la keyword, 2=se opportuno)

```
-inline-level=n
```

- ▶ GNU (n: size, default is 600):

```
-finline-functions  
-finline-limit=n
```



Common Subexpression Elimination

- ▶ Per i calcoli intermedi si riusano spesso alcune espressioni: può essere vantaggioso riciclare quantità già calcolate:

$$A = B + C + D$$

$$E = B + F + C$$

- ▶ Richiede: 4 load, 2 store, 4 somme

$$A = (B + C) + D$$

$$E = (B + C) + F$$

- ▶ Richiede: 4 load, 2 store, 3 somme
- ▶ Attenzione: dal punto di vista numerico il risultato non è necessariamente identico
- ▶ Se la locazione di un array è acceduta più di una volta può convenire effettuare lo “Scalar replacement”
 - ▶ ad opportune ottimizzazioni il compilatore può farlo

Funzioni e Side Effects

- ▶ Lo scopo “primo” di una funzione é in genere dare un valore in ritorno
 - ▶ a volte, per vari motivi, però non é cosí
 - ▶ la modifica di variabili passate, o globali o anche l'I/O si chiamano comunque side effects (effetti collaterali)
- ▶ La presenza di funzioni con side effects può inibire il compilatore dal fare ottimizzazioni

```
function f(x)
  f=x+dx
end
```

allora $f(x) + f(x) + f(x)$ può essere valutato come $3 * f(x)$

```
function f(x)
  x=x+dx
  f=x
end
```

allora la precedente valutazione non é piú corretta

CSE e chiamate a funzione

- ▶ Alterando l'ordine delle chiamate il compilatore non sa se si altera il risultato (possibili effetti collaterali)
- ▶ 5 chiamate a funzioni, 5 prodotti:

```
x=r*sin(a)*cos(b);  
y=r*sin(a)*sin(b);  
z=r*cos(a);
```

- ▶ 4 chiamate a funzioni, 4 prodotti (1 variabile temporanea):

```
temp=r*sin(a)  
x=temp*cos(b);  
y=temp*sin(b);  
z=r*cos(a);
```

CSE: limitazioni

- ▶ Core loop troppo grossi:
 - ▶ il compilatore lavora su finestre di dimensioni finite: potrebbe non accorgersi di una grandezza da riutilizzare
- ▶ Funzioni:
 - ▶ se altero l'ordine delle chiamate ottengo lo stesso risultato?
- ▶ Ordine e valutazione:
 - ▶ solo ad alti livelli di ottimizzazione il compilatore altera l'ordine delle operazioni (**-qnostrict** per IBM)
 - ▶ per inibirla in certe espressioni: mettere le parentesi (il programmatore ha sempre ragione)
- ▶ Aumenta l'uso di registri per l'appoggio dei valori intermedi ("register spilling")

Cosa può fare il compilatore?

```

do k=1,n3m
  do j=n2i,n2do
    jj=my_node*n2do+j
    do i=1,n1m
      acc =1./(1.-coe*aciv(i)*(1.-int(forclo(nve,i,j,k))))
      aci(jj,i)= 1.
      api(jj,i)=-coe*apiv(i)*acc*(1.-int(forclo(nve,i,j,k)))
      ami(jj,i)=-coe*amiv(i)*acc*(1.-int(forclo(nve,i,j,k)))
      fi(jj,i)=qcap(i,j,k)*acc
    enddo
  enddo
enddo
...
...
do i=1,n1m
  do j=n2i,n2do
    jj=my_node*n2do+j
    do k=1,n3m
      acc =1./(1.-coe*ackv(k)*(1.-int(forclo(nve,i,j,k))))
      ack(jj,k)= 1.
      apk(jj,k)=-coe*apkv(k)*acc*(1.-int(forclo(nve,i,j,k)))
      amk(jj,k)=-coe*amkv(k)*acc*(1.-int(forclo(nve,i,j,k)))
      fk(jj,k)=qcap(i,j,k)*acc
    enddo
  enddo
enddo

```

Probabilmente questo

```

do k=1, n3m
  do j=n2i, n2do
    jj=my_node*n2do+j
    do i=1, n1m
      temp = 1.-int(forclo(nve, i, j, k))
      acc = 1./ (1.-coe*aciv(i)*temp)
      aci(jj, i) = 1.
      api(jj, i) = -coe*apiv(i)*acc*temp
      ami(jj, i) = -coe*amiv(i)*acc*temp
      fi(jj, i) = qcacp(i, j, k)*acc
    enddo
  enddo
enddo
...
...
do i=1, n1m
  do j=n2i, n2do
    jj=my_node*n2do+j
    do k=1, n3m
      temp = 1.-int(forclo(nve, i, j, k))
      acc = 1./ (1.-coe*ackv(k)*temp)
      ack(jj, k) = 1.
      apk(jj, k) = -coe*apkv(k)*acc*temp
      amk(jj, k) = -coe*amkv(k)*acc*temp
      fk(jj, k) = qcacp(i, j, k)*acc
    enddo
  enddo
enddo

```

Ma non questo

```

do k=1,n3m
  do j=n2i,n2do
    do i=1,n1m
      temp_fact(i,j,k) = 1.-int(forclo(nve,i,j,k))
    enddo
  enddo
enddo
...
...
do i=1,n1m
  do j=n2i,n2do
    jj=my_node*n2do+j
    do k=1,n3m
      temp = temp_fact(i,j,k)
      acc =1./(1.-coe*ackv(k)*temp)
      ack(jj,k)= 1.
      apk(jj,k)=-coe*apkv(k)*acc*temp
      amk(jj,k)=-coe*amkv(k)*acc*temp
      fk(jj,k)=qcap(i,j,k)*acc
    enddo
  enddo
enddo
...
...
! idem per l'altro loop

```

Array syntax

- ▶ Traslazione dei primi due indici di un array a tre indici (512^3)
- ▶ Il “caro vecchio” loop (stile Fortran 77): **0.19 secondi**
 - ▶ l’ordine dei cicli negli indici che traslano é inverso alla traslazione per evitare di “sporcare” i dati della matrice

```
do k = nd, 1, -1
  do j = nd, 1, -1
    do i = nd, 1, -1
      a03(i, j, k) = a03(i-1, j-1, k)
    enddo
  enddo
enddo
```

- ▶ Array syntax (stile Fortran 90): **0.75 secondi**
 - ▶ secondo lo standard, le cose vanno “come se il membro a destra fosse tutto valutato prima di effettuare le operazioni richieste”

```
a03(1:nd, 1:nd, 1:nd) = a03(0:nd-1, 0:nd-1, 1:nd)
```

- ▶ Array syntax con un hint al compilatore: **0.19 secondi**

```
a03(nd:1:-1, nd:1:-1, nd:1:-1) = a03(nd-1:0:-1, nd-1:0:-1, nd:1:-1)
```

Report di ottimizzazione

- ▶ Per capirne di piú in questo come in altri casi é bene attivare le flag di report di ottimizzazione
- ▶ Con Intel ifort attivare

```
-opt-report [n]          n=0 (none) , 1 (min) , 2 (med) , 3 (max)  
-opt-report-file<file>  
-opt-report-phase<phase>  
-opt-report-routine<routine>
```

- ▶ Le tre modalit  sono alle righe 55,64,69 rispettivamente

```
Loop at line:55 simple MEMOP Intrinsic disabled-->SIMPLE reroll  
Loop at line:64 memcpy generated  
Loop at line:69 simple MEMOP Intrinsic disabled-->SIMPLE reroll
```

- ▶ Tutto questo é ovviamente molto dipendente dal compilatore



Report di ottimizzazione / 2

- ▶ Purtroppo non é presente un'opzione equivalente con compilatori GNU
 - ▶ La migliore alternativa é specificare

```
-fdump-tree-all
```

in modo che vengano stampate tutte le fasi intermedie di compilazione

- ▶ ma la lettura non é decisamente agevole
- ▶ Con il compilatore PGI

```
-Minfo=accel,inline,ipa,loop,lre,mp,opt,par,unified,vect
```

oppure senza opzioni per averle tutte

Dare informazioni al compilatore

- ▶ Estremi del loop noti a compile time o solo a run-time:
 - ▶ può inibire alcune ottimizzazioni, tra cui l'unrolling

```
real a(1:1024,1:1024)
real b(1:1024,1:1024)
real c(1:1024,1:1024)
...
read(*,*) i1,i2
read(*,*) j1,j2
read(*,*) k1,k2
...
do j = j1, j2
do k = k1, k2
do i = i1, i2
c(i, j)=c(i, j)+a(i, k)*b(k, j)
enddo
enddo
enddo
```

- ▶ Tempi in secondi
(Loop Bounds Compile-Time
o Run-Time)

Compilazione	LB-CT	LB-RT
(PLX) ifort -O0	6.72	9
(PLX) ifort -fast	0.34	0.75



Allocazione statica o dinamica ?

- ▶ Potenzialmente l'allocazione statica può dare al compilatore più informazioni per ottimizzare
 - ▶ a prezzo di un codice più rigido
 - ▶ l'elasticità permessa dall'allocazione dinamica è particolarmente utile nel calcolo parallelo

```
integer :: n  
parameter(n=1024)  
real a(1:n,1:n)  
real b(1:n,1:n)  
real c(1:n,1:n)
```

```
real, allocatable, dimension(:, :) :: a  
real, allocatable, dimension(:, :) :: b  
real, allocatable, dimension(:, :) :: c  
print*, 'Enter matrix size'  
read(*, *) n  
allocate(a(n, n), b(n, n), c(n, n))
```



Allocazione statica o dinamica ? / 2

- ▶ Per i compilatori recenti però spesso le prestazioni statica vs dinamica si equivalgono
 - ▶ per il semplice matrice-matrice l'allocazione dinamica gestisce meglio i loop bounds letti da input

Compilazione	statica	dinamica	dinamica-LBRT
(PLX) ifort -O0	6.72	18.26	18.26
(PLX) ifort -fast	0.34	0.35	0.36

- ▶ L'allocazione statica viene fatta nella memoria cosiddetta "stack"
 - ▶ in compilazione possono esserci dei limiti di utilizzo per cui occorre specificare l'opzione **-mmodel=medium**
 - ▶ a run-time assicurarsi che sul nodo la stack non sia limitata (se si usa bash)

```
ulimit -a
```

ed eventualmente

```
ulimit -s unlimited
```

Allocazione dinamica in C

- ▶ C non conosce matrici ma array di array
 - ▶ l'allocazione statica garantisce allocazione contigua di tutti i valori

```
double A[nrows][ncols];
```

- ▶ Con l'allocazione dinamica occorre fare attenzione
 - ▶ “the wrong way” (= non efficiente)

```
/* Allocate a double matrix with many malloc */  
double** allocate_matrix(int nrows, int ncols) {  
    double **A;  
    /* Allocate space for row pointers */  
    A = (double**) malloc(nrows*sizeof(double*) );  
    /* Allocate space for each row */  
    for (int ii=1; ii<nrows; ++ii) {  
        A[ii] = (double*) malloc(ncols*sizeof(double));  
    }  
    return A;  
}
```

Allocazione dinamica in C / 2

- ▶ Si può allocare un array lineare

```
/* Allocate a double matrix with one malloc */
double* allocate_matrix_as_array(int nrows, int ncols) {
    double *arr_A;
    /* Allocate enough raw space */
    arr_A = (double*) malloc(nrows*ncols*sizeof(double));
    return arr_A;
}
```

- ▶ e usarlo come una matrice (linearizzazione dell'indice)

```
arr_A[i*ncols+j]
```

- ▶ le MACROs possono aiutare
- ▶ e, eventualmente aggiungere una matrice di puntatori che puntano all'array allocato

```
/* Allocate a double matrix with one malloc */
double** allocate_matrix(int nrows, int ncols, double* arr_A) {
    double **A;
    /* Prepare pointers for each matrix row */
    A = new double*[nrows];
    /* Initialize the pointers */
    for (int ii=0; ii<nrows; ++ii) {
        A[ii] = &(arr_A[ii*ncols]);
    }
    return A;
}
```

Aliasing e Restrict

- ▶ In C, se due puntatori puntano ad una stessa area di memoria, si parla di “aliasing”
- ▶ Il rischio di aliasing può **molto** limitare l’ottimizzazione del compilatore
 - ▶ difficile invertire l’ordine delle operazioni
 - ▶ particolarmente per gli argomenti passati a una funzione
- ▶ Lo standard C99 introduce la keyword **restrict** per indicare che l’aliasing non é possibile

```
void saxpy(int n, float a, float *x, float* restrict y)
```

- ▶ In C++, si assume che l’aliasing non possa avvenire tra puntatori a tipi diversi (strict aliasing)



Aliasing e Restrict / 2

- ▶ Il Fortran assume che gli argomenti di procedure non possano puntare a identiche aree di memoria
 - ▶ tranne che per gli array per i quali gli indici permettono comunque un'analisi corretta
 - ▶ o per i **pointer** che però vengono usati ove necessario
 - ▶ un motivo per cui il Fortran spesso ottimizza meglio del C!
- ▶ I compilatori permettono di configurare le assunzioni dell'aliasing (vedere il man)
 - ▶ GNU (solo strict-aliasing): **-fstrict-aliasing**
 - ▶ Intel (eliminazione completa): **-fno-alias**
 - ▶ IBM (no overlap per array): **-qalias=noaryovrlp**

Non solo compilatore

- ▶ Il compilatore ha associata una runtime library
- ▶ Contiene funzioni chiamate esplicitamente
 - ▶ funzioni trigonometriche e trascendenti
 - ▶ manipolazioni di bit
 - ▶ funzioni Input Output (C)
- ▶ Contiene funzioni chiamate implicitamente
 - ▶ funzioni Input Output (Fortran)
 - ▶ operatori complessi del linguaggio
 - ▶ routine di utilità generiche, gestione eccezioni, . . .
 - ▶ routine di supporto ad un particolare modello di calcolo (OpenMP, UPC, GAF)
- ▶ Può essere fondamentale per le prestazioni
 - ▶ qualità dell'implementazione
 - ▶ funzioni matematiche accurate vs. veloci



Input/Output

- ▶ È sempre mediato dal sistema operativo
 - ▶ causa chiamate di sistema
 - ▶ comporta lo svuotamento della pipeline
 - ▶ distrugge la coerenza dei dati in cache
 - ▶ può alterare la priorità di scheduling
 - ▶ è lento
- ▶ Regolo d'oro n.1: MAI mescolare calcolo intensivo con I/O
- ▶ Regolo d'oro n.2: leggere/scrivere i dati in blocco, non pochi per volta
- ▶ Attenzione ad I/O nascosti: swapping
 - ▶ avviene quando la RAM è insufficiente
 - ▶ usa il disco come surrogato
 - ▶ unica soluzione: fuggirlo come la peste

Ci sono più modi di fare I/O

```

do k=1,n ; do j=1,n ; do i=1,n
write(69,*) a(i,j,k) ! formattato
enddo ; enddo ; enddo

do k=1,n ; do j=1,n ; do i=1,n
write(69) a(i,j,k) ! binario
enddo ; enddo ; enddo

do k=1,n ; do j=1,n
write(69) (a(i,j,k),i=1,n) ! colonne
enddo ; enddo

do k=1,n
write(69) ((a(i,j,k),i=1),n,j=1,n) ! matrice
enddo

write(69) (((a(i,j,k),i=1,n),j=1,n),k=1,n) ! blocco

write(69) a ! dump

```

Con prestazioni differenti

Opzione	secondi	Kbyte
formattato	81.6	419430
binario	81.1	419430
colonne	60.1	268435
matrice	0.66	134742
blocco	0.94	134219
dump	0.66	134217

- Il file-system e anche il suo utilizzo hanno un notevole impatto sui tempi

I/O riassumendo

- ▶ La lettura/scrittura dei dati formattati è lenta
- ▶ Leggere/scrivere i dati in formato binario
- ▶ Leggere/scrivere in un blocco e non uno per volta
- ▶ Scegliere il file system più efficiente a disposizione
- ▶ I buffer di scrittura possono nascondere latenze
- ▶ Ma l'impatto sul calcolo sarà comunque devastante
- ▶ Attenzione al dump di array in caso di padding
- ▶ Soprattutto per il calcolo parallelo:
 - ▶ usare librerie di I/O: MPI-I/O, HDF5, NetCDF,...

Unità vettoriali

- ▶ Da non confondere con le macchine vettoriali!
- ▶ Le unità vettoriali lavorano con set di istruzioni SIMD e circuiti dedicati a operazioni floating-point simultanee
 - ▶ Intel MMX (1996), AMD 3DNow! (1998), Intel SSE (1999) che aggiungono nuovi registri e possibilità floating point
 - ▶ Nuove istruzioni (packet) SSE2, SSE3, SSE4, AVX
- ▶ Esempio di vettorizzazione: addizione di due array a 4 componenti può diventare una singola istruzione

$$c(0) = a(0) + b(0)$$

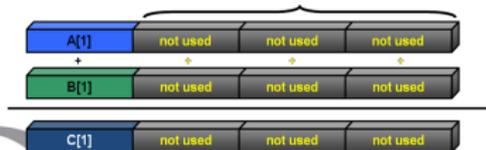
$$c(1) = a(1) + b(1)$$

$$c(2) = a(2) + b(2)$$

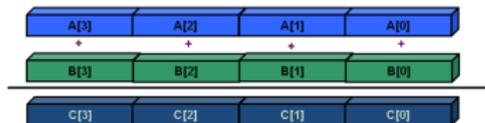
$$c(3) = a(3) + b(3)$$

non vettorizzato

e.g. 3 x 32-bit unused integers



vettorizzato



Evoluzione unità SIMD

- ▶ SSE: registri a 128 bit (Intel Core - AMD Opteron)
 - ▶ 4 operazioni floating/integer in singola precisione
 - ▶ 2 operazioni floating/integer in doppia precisione
- ▶ AVX: registri a 256 bit (Intel Sandy Bridge - AMD Bulldozer)
 - ▶ 8 operazioni floating/integer in singola precisione
 - ▶ 4 operazioni floating/integer in doppia precisione
- ▶ MIC: registri a 512 bit (Intel Knights Corner - 2013)
 - ▶ 16 operazioni floating/integer in singola precisione
 - ▶ 8 operazioni floating/integer in doppia precisione



Vettorizzazione

- ▶ La vettorizzazione dei loop può incrementare drammaticamente le performance
- ▶ Ma per essere vettorizzabili, i loop devono obbedire a certi criteri
- ▶ E il programmatore deve aiutare il compilatore a verificarli
- ▶ Anzitutto, l'assenza di dipendenza tra i dati di diverse iterazioni
 - ▶ circostanza frequente ma non troppo in ambito HPC
- ▶ Altri criteri
 - ▶ Countable (numero delle iterate costante)
 - ▶ Single entry-single exit (nessun break or exit)
 - ▶ Straight-line code (nessun branch)
 - ▶ Deve essere il loop interno di nest
 - ▶ Nessuna chiamata a funzione (eccetto quelle matematiche o quelle inlined)
- ▶ AVX può essere una sorgente di risultati diversi in calcolo numerico (e.g., Fused Multiply Addition)

Algoritmi e vettorizzazione

- ▶ Differenti algoritmi per lo stesso scopo possono comportarsi diversamente rispetto alla vettorizzazione
 - ▶ Gauss-Seidel: dipendenza tra le iterazioni, non vettorizzabile

```
for( i = 1; i < n-1; ++i )  
  for( j = 1; j < m-1; ++j )  
    a[i][j] = w0 * a[i][j] +  
      w1*(a[i-1][j] + a[i+1][j] + a[i][j-1] + a[i][j+1]);
```

- ▶ Jacobi: nessuna dipendenza tra le iterazioni, vettorizzabile

```
for( i = 1; i < n-1; ++i )  
  for( j = 1; j < m-1; ++j )  
    b[i][j] = w0*a[i][j] +  
      w1*(a[i-1][j] + a[i][j-1] + a[i+1][j] + a[i][j+1]);  
for( i = 1; i < n-1; ++i )  
  for( j = 1; j < m-1; ++j )  
    a[i][j] = b[i][j];
```

Ottimizzazione e vettorizzazione

- ▶ Alcuni comuni “coding tricks” possono impedire la vettorizzazione
 - ▶ vettorizzabile

```
for( i = 0; i < n-1; ++i ){  
    b[i] = a[i] + a[i+1];  
}
```

- ▶ **x** a una certa iterazione é necessaria per lo step successivo

```
x = a[0];  
for( i = 0; i < n-1; ++i ){  
    y = a[i+1];  
    b[i] = x + y;  
    x = y;  
}
```

- ▶ Quando si compila é bene controllare se la vettorizzazione é stata attivata
- ▶ In caso contrario, si può provare ad aiutare il compilatore
 - ▶ modificando il codice per renderlo vettorizzabile
 - ▶ inserendo direttive per forzare la vettorizzazione

Direttive di vettorizzazione

- ▶ Se il programmatore ha certezza che una certa dipendenza riscontrata dal compilatore sia in realtà solo apparente può forzare la vettorizzazione con direttive “compiler dependent”
 - ▶ Intel Fortran: **!DIR\$ simd**
 - ▶ Intel C: **#pragma simd**
- ▶ Poichè **inow** è diverso da **inew**, la dipendenza è solo apparente

```

62      do k = 1, n
63!DIR$ simd
        do i = 1, 1
...
66          x02 = a02(i-1, k+1, inow)
67          x04 = a04(i-1, k-1, inow)
68          x05 = a05(i-1, k, inow)
          x06 = a06(i, k-1, inow)
          x11 = a11(i+1, k+1, inow)
          x13 = a13(i+1, k-1, inow)
72          x14 = a14(i+1, k, inow)
73          x15 = a15(i, k+1, inow)
74          x19 = a19(i, k, inow)
75
76          rho =+x02+x04+x05+x06+x11+x13+x14+x15+x19
...
126         a05(i, k, inew) = x05 - omega*(x05-e05) + force
127         a06(i, k, inew) = x06 - omega*(x06-e06)

```

Vettorizzazione manuale

- ▶ È possibile istruire direttamente il codice delle funzioni vettoriali da utilizzare
- ▶ In pratica, si tratta di scrivere un loop che fa quattro iterazioni alla volta usando registri e operazioni vettoriali, ed essere pratici con le “mask”

```
void scalar(float* restrict result,
           const float* restrict v,
           unsigned length)
{
    for (unsigned i = 0; i < length; ++i)
    {
        float val = v[i];
        if (val >= 0.f)
            result[i] = sqrt(val);
        else
            result[i] = val;
    }
}
```

```
void sse(float* restrict result,
         const float* restrict v,
         unsigned length)
{
    __m128 zero = _mm_set1_ps(0.f);

    for (unsigned i = 0; i <= length - 4; i += 4)
    {
        __m128 vec = _mm_load_ps(v + i);
        __m128 mask = _mm_cmpge_ps(vec, zero);
        __m128 sqrt = _mm_sqrt_ps(vec);
        __m128 res =
            _mm_or_ps(_mm_and_ps(mask, sqrt),
                    _mm_andnot_ps(mask, vec));
        _mm_store_ps(result + i, res);
    }
}
```



Parallelizzazione automatica

- ▶ Alcuni compilatori offrono opzioni per sfruttare il parallelismo architetturale delle macchina (e.g., i cores) senza modificare il codice sorgente
- ▶ Shared Memory Parallelism (solo intra-nodo)
- ▶ Simile a OpenMP ma non richiede direttive
 - ▶ performance attese piú limitate
- ▶ GNU:

```
-floop-parallelize-all
-ftree-parallelize-loops=n - n= number of threads
```

- ▶ Intel:

```
-parallel
-par-threshold[n] - set loop count threshold
-par-report{0|1|2|3}
```

- ▶ IBM:

```
-qsmp                la abilita automaticamente
-qsmp=openmp:noauto per disabilitare la
                    parallelizzazione automatica
```

Outline

Compilatori e ottimizzazione

Floating Point Computing

Makefile

Architectures

Power Performance

Aritmetica in virgola mobile

Dato un certo numero $x \in \mathfrak{R}$ si può dimostrare che f , la sua rappresentazione in numero a virgola mobile può scriversi come

$$f = \sigma \cdot s \cdot 2^e$$

dove

$\sigma \in [1, 2)$, $s = 1 + k/2^{N-1}$, $k \in (0, 1, 2, \dots, 2^{N-1} - 1)$,
 $e \in [\epsilon_{min}, \epsilon_{max}] \cap \mathbb{Z}$ con $\epsilon_{min} = -2^{M-1} + 2$, $\epsilon_{max} = 2^{M-1} - 1$ e \mathbb{Z}
l'insieme dei numeri interi definiti positivi, dato un campo degli
esponenti a M-bit, con N-bit significativi.

Aritmetica in virgola mobile

Ma vediamo in maniera più pratica come viene rappresentato in memoria un numero in virgola mobile:

Singola precisione



Doppia precisione



Aritmetica in virgola mobile

In entrambi i casi vediamo che abbiamo un bit di segno (s), un campo (e) e un campo di mantissa o significativo (f), ma con un differente numero di bit per campo.

Abbiamo detto in precedenza la definizione di rappresentazione floating-point, ma un numero a virgola mobile può anche essere scritto come

$$(-1)^s \cdot d_0 \cdot d_1 \cdot d_2 \dots \cdot d_{p-1} \cdot \beta^e$$

dove p rappresenta il massimo numero di cifre significative nel significando e β è la base del sistema numerico usato (generalmente 2 o 16).

È chiaro che rappresentare un numero reale con un sistema a virgola mobile presenta diversi problemi.

Due in particolare sono i più evidenti:

- ▶ Non si può rappresentare in maniera esatta un numero reale in un sistema a virgola mobile.
- ▶ Un numero reale può essere troppo grande o troppo piccolo per poter essere rappresentato con un range di esponente disponibile.



Aritmetica in virgola mobile

Quale esempio del primo problema consideriamo di rappresentare il numero 0.2. In base 10 ($b=10$) con una precisione pari a tre cifre ($p=3$) la rappresentazione è esatta: $2.00 \cdot 10^{-1}$.

Però, in base 2 ($b=2$) con diciamo, 6 cifre significative, abbiamo che 0.2 dovrebbe essere rappresentato come $1.10011 \cdot 2^{-3}$ (ci perdiamo una coda di $1.1001100110011... \cdot 2^{-9}$).

Quindi avremo sempre un errore nel rappresentare i valori che vogliamo usare nei nostri calcoli ed essi saranno contaminati da questi errori di rappresentazione anche se il calcolo in se stesso è esatto.

Per risolvere questa situazione dobbiamo far leva su di un compromesso tra precisione nella traduzione da numero reale a numero in virgola mobile e velocità di esecuzione nella traduzione stessa, funzione del numero di bit usati nella rappresentazione floating-point.

Lo standard IEEE754

Questi ed altri problemi (errori di arrotondamento, relativi alla precisione di macchina, etc.) furono affrontati diverso tempo fa.

Nel 1979, il benemerito Istituto USA di Ingegneria Elettrica ed Elettronica (IEEE) propose uno standard per i calcoli a virgola mobile, l'IEEE 754.

Questo standard che si è concretizzato nella sua forma definitiva nel 1981, come suggerisce il nome, non riguarda la rappresentazione di interi (a parte la conversione in numeri floating point), bensì quella dei numeri in virgola mobile.

Lo standard IEEE754

Lo standard assume che la base di rappresentazione sia 2 e tra le altre cose chiarisce in maniera precisa i seguenti punti:

- ▶ Il formato delle rappresentazioni dei numeri a virgola mobile
- ▶ I risultati delle operazioni di addizione, sottrazione, moltiplicazione, divisione, radice quadrata, resto e comparazione di numeri a virgola mobile
- ▶ La conversione tra interi e numeri floating-point
- ▶ La conversione tra il formato base a virgola mobile e stringhe decimali
- ▶ Le eccezioni (exception) e loro gestione incluso i non-numeri (i cosiddetti, NaN, Not a Number).



Lo standard IEEE754

Lo standard non definisce i termini di applicazione di quanto sopra che può essere realizzato sia in hardware che in software, ma ha permesso di semplificare la gestione dei dati floating-point e di avere un comune standard di riferimento per essi.

Per questo motivo, tutti i computer oggi in commercio adottano le soglie sui dati in virgola mobile definite dallo standard IEEE 754 che sono:

N.B La disposizione in memoria di numeri a virgola mobile a singola e doppia precisione data in precedenza è proprio quella prevista dallo standard IEEE754.



Lo standard IEEE754

Singola precisione

Precisione relativa $2^{-23} \approx 1.1920929 \cdot 10^{-7}$

Numero più piccolo $(-1)^s 2^{-126} \approx 1.1754944 \cdot 10^{-38}$

Numero più grande $(-1)^s (1 - 2^{-24}) 2^{+128} \approx 3.402823510^{+38}$

Doppia precisione

Precisione realtiva $2^{-52} \approx 2.2204460492503 \cdot 10^{-16}$

Numero più piccolo $(-1)^s 2^{-1022} \approx 2.2250738585072 \cdot 10^{-308}$

Numero più grande

$(-1)^s (1 - 2^{-53}) 2^{+1024} \approx 1.7976931348623^{+308}$

Outline

Compilatori e ottimizzazione

Floating Point Computing

Makefile

Architectures

Power Performance

L'utility Make

L'utility *make* è essenzialmente un linguaggio di programmazione per automatizzare il processo di compilazione.

L'idea dietro *make* è che il programmatore non deve ricompilare un certo file se esiste già un file oggetto corrispondente. Il file oggetto si dice *corrente* se è stato compilato più recentemente dell'ultimo cambiamento al suo file sorgente.

Come esempio consideriamo la seguente situazione:

Abbiamo appena modificato il file sorgente *program.f* e vogliamo compilarlo e "linkarlo" ai moduli *inputs.f* e *outputs.f*.

Il comando corrispondente (f77) sarà:

```
f77 program.f inputs.f outputs.f
```

Il quale esegue la compilazione ed il *linking* dei tre moduli correttamente.

L'utility Make

Sei i moduli oggetto *inputs.o* e *outputs.o* esistono già e se essi sono correnti (non modificati) avremmo potuto dare il comando (*):

```
f77 program.f inputs.o outputs.o
```

In questo caso f77 avrebbe compilato *program.f* e fatto solo il link di *program.o*, *inputs.o* e *outputs.o*.

Ovviamente compilando un solo programma invece di tre, il processo di compilazione-linking sarà molto più veloce ed efficiente.

Operare come nel secondo caso, quando si hanno molti file sorgenti e oggetto a disposizione, diviene un'operazione assai laboriosa e da qui, quindi, l'esigenza di una utility come *make* che può fare questo per noi in maniera automatica.

(*) in questo caso, non avendo usato l'opzione **-c**, il compilatore dopo aver generato il/i file oggetto invoca automaticamente il linker **ld** per generare l'eseguibile. In Unix, se non diamo un nome al nostro eseguibile mediante l'opzione **-o**, il nome di default ad esso assegnato è **a.out**.



L'utility Make

Inoltre, *make* riduce la possibilità di errori di compilazione che possono verificarsi sia per errori nella liena di comando che nella scelta dei moduli d'utilizzare.

Per capire come funziona *make*, dobbiamo prima definire alcuni termini:

target	Una operazione da eseguire. Nella maggior parte dei casi esso contiene il nome del file da creare.
dipendenza	Una relazione tra due target: il target A dipende dal target B se un cambiamento in B produce una variazione in A.
up-to-date	Un file che è più recente di ognuno dei file da cui dipende
makefile	Un file che descrive come creare uno o più target. In esso sono listati tutti i file dai quali i target dipendono e da le regole necessarie a compilare questi target correttamente.
make	Il comando che esegue quanto contenuto in un makefile. Per default <i>make</i> cerca un file chiamato <i>makefile</i> o <i>Makefile</i> nella directory corrente. Come opzione si può usare il target da compilare.

L'utility Make

Ma facciamo un esempio pratico, riportando in un *makefile* il caso visto con *f77* con due target diversi: ad es. usiamo due diverse opzioni per *f77*.

```
#
# Un semplice makefile
# (Le righe di commenti cominciano con #)
#
# Primo target:
#
program:
    f77 -o program -O program.f inputs.f outputs.f

#Secondo target:
#
program.db:
    f77 -o program -DDEBUG -g program.f inputs.f outputs.f
```

Se usiamo questo *makefile* con *make* possiamo scegliere tra i due target semplicemente scrivendoli dopo *make* ...



L'utility Make

Ad esempio,

```
make program
```

eseguirà il comando

```
f77 -o program -O program.f inputs.f outputs.f
```

e produrrà l'eseguibile **program**.

È importante ricordare che *make* creerà una nuova shell per ognuno dei comandi che incontra dopo il *target* e che questi vanno dati dopo il carattere **[TAB]** (ogni riga dei comandi per un *target* comincia con un **[TAB]**...)



L'utility Make

Se vogliamo che i nostri *target* dipendano da uno o più file, dobbiamo allora indicare le relative dipendenze. Questo si fa' aggiungendo il nome dei file da cui un certo *target* dipende dopo i [:]

```
program: program.o inputs.o outputs.o
        f77 -o program program.o inputs.o outputs.o
```

La prima riga ci dice che **program** dipende da *program.o*, *inputs.o* e *outputs.o*, mentre la seconda usa `f77` (ovvero `ld`) per fare il link dei file oggetti se essi già esistono.

Se invece i file oggetto non esistono, essi dovranno avere le loro dipendenze nel *makefile* dovrà esserci un *target* per ognuno dei file oggetto. In questo modo *make* prima genera i file oggetto e poi l'eseguibile.

L'utility Make

In definitiva il nostro *makefile* completo sarà:

```
program: program.o inputs.o outputs.o
    f77 -o program program.o inputs.o outputs.o

program.o: program.f
    f77 -c -o program.o program.f

inputs.o: inputs.f header.f
    f77 -c -o inputs.o inputs.f

outputs.o outputs.f
    f77 -c -o outputs.o outputs.f
```

L'utility Make

Nel makefile si possono usare delle abbreviazioni e macro.

Tra le abbreviazioni più usate abbiamo

`$@` Indica il nome completo del target

`$*` Indica il nome del target senza suffisso

Usandole in una riga dell'esempio precedente avremo:

```
program: program.o inputs.o outputs.o
        f77 -o $@ $*.o inputs.o outputs.o
```

Una macro, invece, serve in un *makefile* per raggruppare in un'unica variabile diversi nomi o comandi.

Ad esempio, se poniamo nella macro **DEPENDS** i nostri file oggetto, potremo scrivere la riga precedente come

```
DEPENDS = program.o inputs.o outputs.o
```

```
program: $(DEPENDS)
        f77 -o $@ $ (DEPENDS)
```

Outline

Compilatori e ottimizzazione

Floating Point Computing

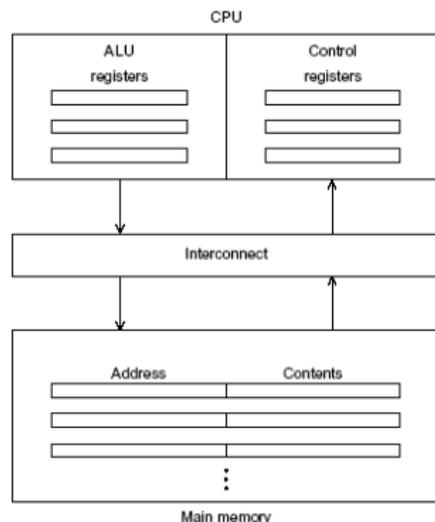
Makefile

Architectures

Power Performance

Von Neumann Architecture

- ▶ Central Processing Unit (CPU)
 - ▶ Arithmetic logic unit (executes instructions)
 - ▶ Control unit
 - ▶ Registers (fast memory)
- ▶ Interconnection CPU RAM (Bus)
- ▶ Random Access Memory (RAM)
 - ▶ Address to access memory locations
 - ▶ Memory content (instructions, data)



Von Neumann Architecture

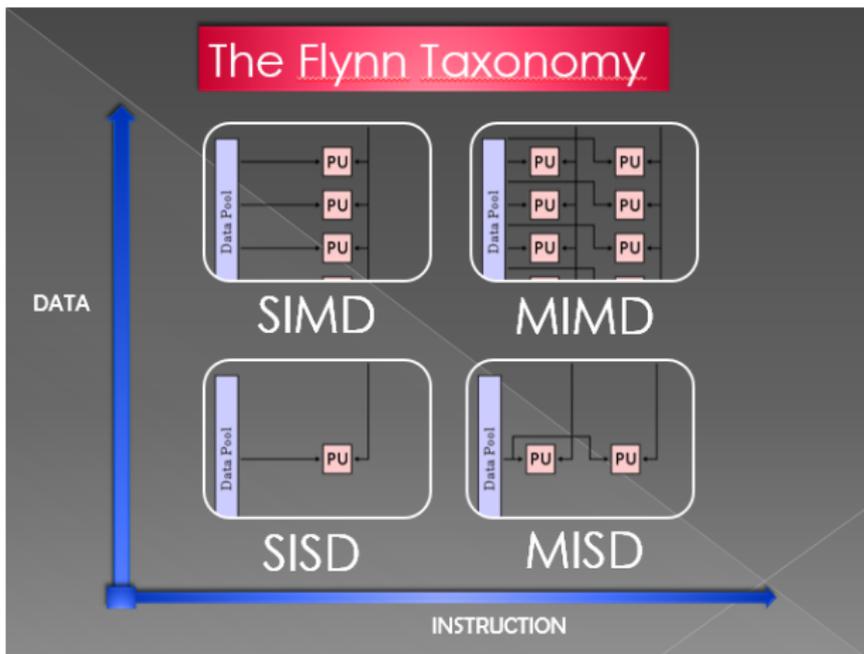
- ▶ Data are sent from memory to CPU (fetch or read)
- ▶ Data are sent from CPU to memory (written to memory or stored)
- ▶ The separation between the CPU and memory leads to what is known as the «von Neumann bottleneck»: the limited throughput (data transfer rate) between the CPU and memory compared to the amount of memory
- ▶ In most modern computers, the throughput is one hundred smaller compared to the rate the CPU can process



Solutions to the von Neumann Bottleneck

- ▶ Caching
Very fast memories integrated directly into the processor chip.
There are first, second or third level caches
- ▶ Virtual memory
The RAM is used as a cache for big data storage.
- ▶ Instruction level parallelism
single CPU core, but multiple functional units to execute
multiple instructions in parallel (pipelining, multiple issues)

Flynn's taxonomy

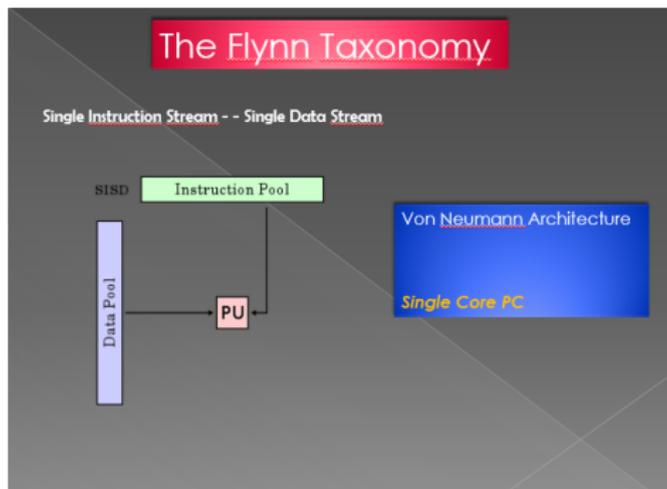


Flynn's taxonomy

- ▶ It classifies computer architectures based on the number of instructions that can be executed and how they operate on data.
 - ▶ **SISD**: single instruction, single data. Classical von Neumann architecture, scalar mono-processor system.
 - ▶ **SIMD**: single instruction, multiple data. Vectorial architectures, vectorial processors, GPU.
 - ▶ **MISD**: multiple instructions, single data. This hardware solution does not exist.
 - ▶ **MIMD**: multiple instructions, multiple data. Multiple processors/cores execute different instructions operating on different data
- ▶ The modern computer architectures combine features from the previous categories

SISD

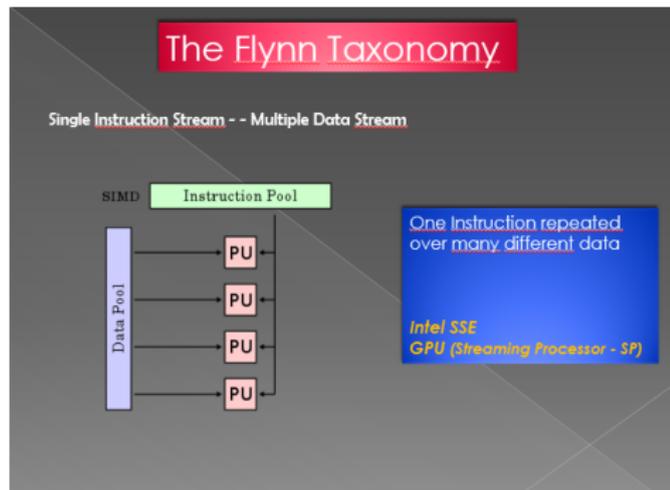
- ▶ Classical von Neumann architecture
- ▶ Scalar mono-processor systems (single core).
- ▶ The execution of the instructions can be pipelined (Cyber 76).
- ▶ Each arithmetic instruction starts an arithmetic operation.



SIMD

A single instruction operates simultaneously on multiple data
Synchronous computational model

- ▶ Vectorial processors
 - ▶ Many ALU
 - ▶ Vectorial registers
 - ▶ Load/Store vector units
 - ▶ Vector instructions
 - ▶ Interleaved memory
 - ▶ OpenMP, MPI
- ▶ Graphical Processing Unit
 - ▶ Programmable GPU
 - ▶ many ALU
 - ▶ many Load/Store units
 - ▶ many SFU
 - ▶ one thousand threads that work in parallel
- ▶ CUDA

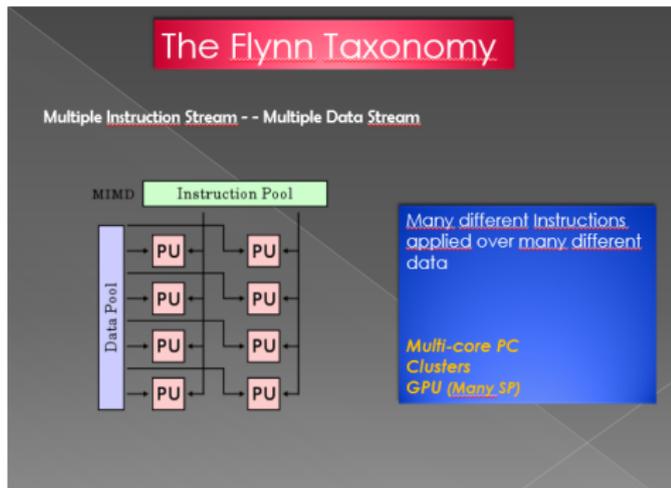


MIMD

Multiple processors execute different instructions operating on different data. Asynchronous computational model

► Cluster

- many nodes (hundreds/thousands)
- more multicore processors per node
- shared RAM node
- distributed RAM between nodes
- hierarchy memory levels
- OpenMP, MPI, MPI+OpenMP



MIMD:Fermi Blue Gene/Q

Model: IBM-BlueGene /Q

Architecture: 10 BGQ Frame with 2 MidPlanes each

Front-end Nodes OS: Red-Hat EL 6.2

Compute Node Kernel: lightweight Linux-like kernel

Processor Type: IBM PowerA2, 16 cores, 1.6 GHz

Computing Nodes: 10.240

Computing Cores: 163.840

RAM: 16GB / node

Internal Network: Network interface
with 11 links ->5D Torus

Disk Space: more than 2PB of scratch space

Peak Performance: 2.1 PFlop/s

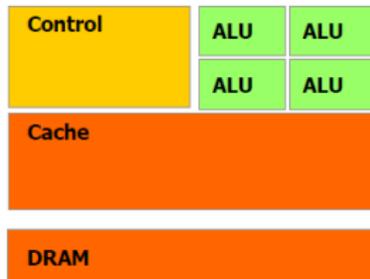
Cluster CPU GPU

- ▶ Hybrid Solution CPU multi-core + GPU many-core:
 - ▶ Each compute node has multi-core processors and graphics card with processors dedicated to GPU Computing.
 - ▶ Theoretical massive amount of power computing on single node.
 - ▶ Additional memory embedded inside GPU
 - ▶ OpenMP, MPI, CUDA and hybrid solutions MPI+OpenMP, MPI+CUDA, OpenMP+CUDA, OpenMP+MPI+CUDA

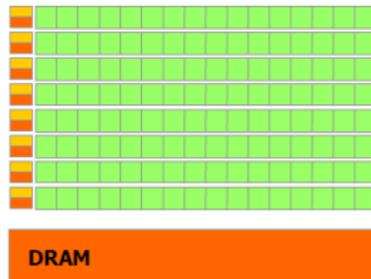


CPU vs GPU

- ▶ The CPU is a general purpose processor
 - ▶ threads dedicated to heavy computations, at most 1 thread per core
- ▶ The GPU is a processor dedicated to intense data-parallel computations
 - ▶ very easy flow control
 - ▶ many light-weight threads that work in parallel



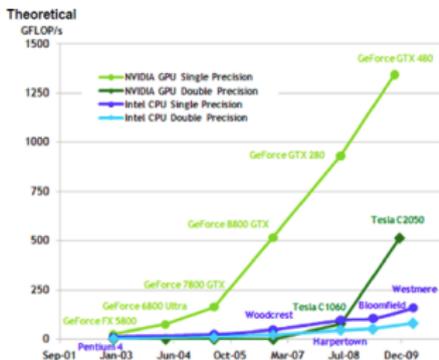
CPU



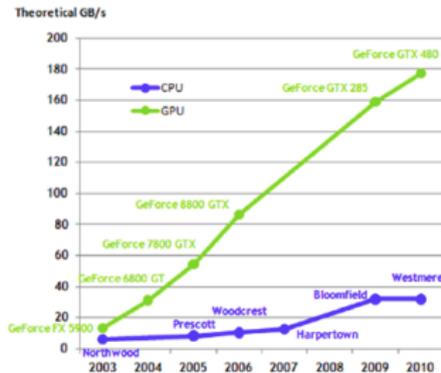
GPU

Multicore vs Manycore

- ▶ Recent trends in the microprocessor architectures:
 - ▶ increasing the overall computing power by increasing the number of cores rather than the computing power of a single core.
 - ▶ the computing power and the bandwidth of a GPU are 10 times larger compared to the CPU ones



Numero di operazioni in virgola mobile al secondo per la CPU e la GPU



Larghezza di banda della memoria

INTEL MIC

- ▶ Intel Xeon Phi Coprocessor
 - ▶ Based on Intel Many Integrated Core (MIC) architecture
 - ▶ 60 core/1,053 GHz/240 threads
 - ▶ 8 GB memory and 320 GB/s bandwidth
 - ▶ Get up to 1 teraFLOPS of double precision peak performance
 - ▶ 512-bit SIMD units
 - ▶ Traditional parallel approach: MPI, OpenMP



Bandwidth

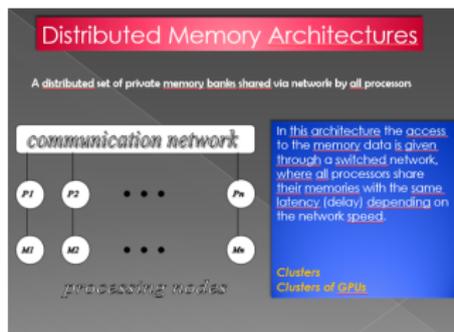
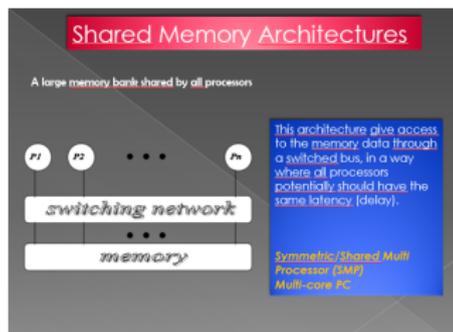
- ▶ Defined as the amount of data transferred per second between memory and processor
- ▶ Measured in number of bytes per second (Mb/s, Gb /s, etc..)
- ▶ $A = B * C$
 - ▶ B: data which is read from memory
 - ▶ C: data which is read from memory
 - ▶ Multiplication $B * C$ is calculated
 - ▶ The result is saved to memory using the same place of the A variable
- ▶ 1 floating-point operation \rightarrow 3 memory accesses

Stream

- ▶ The STREAM benchmark is a simple synthetic benchmark program that measures sustainable memory bandwidth (in MB/s) and the corresponding computation rate for simple vector kernels
- ▶ Time elapsed to:
 - ▶ $a \rightarrow c$ (copy)
 - ▶ $a*b \rightarrow c$ (scale)
 - ▶ $a+b \rightarrow c$ (add)
 - ▶ $a+b*c \rightarrow d$ (triad)
- ▶ It measures the maximum bandwidth
- ▶ <http://www.cs.virginia.edu/stream/ref.html>

Shared and Distributed memory

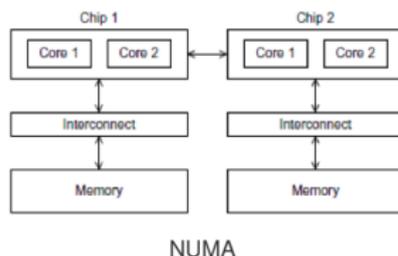
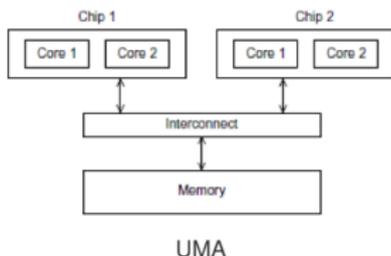
- ▶ The classical MIMD architectures and the hybrid CPU-GPU architectures are classified in two categories.
 - ▶ Shared memory systems: each core can access the entire system memory
 - ▶ Distributed memory systems: each processor can directly access only to its own local memory. The communication among different processors occurs via a specific communication protocol



- ▶ In the modern multicore systems, the memory is shared within a single node and distributed between nodes.

Shared Memory

- ▶ **Uniform Memory Access** where all processors access the main memory at the same speed using the interconnection system.
- ▶ **Non Uniform Memory Access** where a multicore processor can access its own local memory faster than non-local memory. The non-local memory is memory local to another processor or memory shared between processors.



UMA NUMA

- ▶ Main problems:
 - ▶ The efficiency of the cache optimization decreases when a process (or thread) is transferred from one CPU to another
 - ▶ UMA machines: memory intensive processes running on different CPUs may significantly decrease performance due to the bus contention
 - ▶ NUMA machines : Run slow down when CPU accesses non local memory data.
- ▶ Solutions:
 - ▶ binding of processes or binding of threads to CPU
 - ▶ memory affinity
 - ▶ for AIX architecture, set MEMORY_AFFINITY environment variable.
 - ▶ when implemented: use numactl

Networks

- ▶ In high performance systems, different processors are arranged on different nodes. The memory is shared between different processors in the same node, while it is distributed between nodes.
 - ▶ Nodes are connected to different networks.
 - ▶ Gigabit Ethernet: the most common, low cost, low performances
 - ▶ Infiniband: more common, high performances, very expensive
 - ▶ Myrinet: less common
- ▶ old networks
 - ▶ Quadrics
 - ▶ Cray



Rank	System	Cores	Rmax (TFlop/s)	Rpeak (TFlop/s)	Power (kW)
1	Sunway TaihuLight - Sunway MPP, Sunway SW26010 260C 1.45GHz, Sunway , NRPCC National Supercomputing Center in Wuxi China	10,649,600	93,014.6	125,435.9	15,371
2	Tianhe-2 (MilkyWay-2) - TH-IVB-FEP Cluster, Intel Xeon E5-2692 12C 2.200GHz, TH Express-2, Intel Xeon Phi 3151P , NUDT National Super Computer Center in Guangzhou China	3,120,000	33,862.7	54,902.4	17,808
3	Piz Daint - Cray XC50, Xeon E5-2690v3 12C 2.6GHz, Aries interconnect , NVIDIA Tesla P100 , Cray Inc. Swiss National Supercomputing Centre (CSCS) Switzerland	361,760	19,590.0	25,326.3	2,272
4	Titan - Cray XK7, Opteron 6274 16C 2.200GHz, Cray Gemini interconnect, NVIDIA K20x , Cray Inc. DDE/SC/Oak Ridge National Laboratory United States	560,640	17,590.0	27,112.5	8,209
5	Sequoia - BlueGene/Q, Power BQC 16C 1.60 GHz, Custom , IBM DOE/NNSA/LLNL United States	1,572,864	17,173.2	20,132.7	7,890

Rank	System	Cores	Rmax (TFlop/s)	Rpeak (TFlop/s)	Power (kW)
14	Marconi Intel Xeon Phi - CINECA Cluster, Intel Xeon Phi 7250 68C 1.4GHz, Intel Omni-Path , Lenovo CINECA Italy	241,808	6,223.0	10,833.0	1,600.2
64	Fermi - BlueGene/Q, Power BQC 16C 1.60GHz, Custom , IBM CINECA Italy	163,840	1,788.9	2,097.2	821.9
66	Marconi Intel Xeon - Lenovo NeXtScale nx360M5, Xeon E5-2697v4 18C 2.3GHz, Omni-Path , Lenovo CINECA Italy	54,432	1,723.9	2,003.1	1,360.8
281	GALILEO - IBM NeXtScale nx360M4, Xeon E5-2630v3 8C 2.4GHz, Infiniband QDR, Intel Xeon Phi 7120P , IBM/Lenovo CINECA Italy	50,232	684.3	1,103.1	2,825.6

MIMD: Marconi

Partition	Installation	CPU	# nodes	# of Racks	Power
A1 – Broadwell (2.1PFlops)	April 2016	E5-2697 v4	1512	25	700KW
A2 - Knight Landing (11 Pflops)	September 2016	KNL	3600	50	1300KW
A3 – Skylake (4.5PFlops)	June 2017	E5-2680 v5	1512	25	700KW

MIMD: Marconi

Partizione A1

1512 Lenovo NeXtScale Server -> 2PFlops
processore Intel E5-2697 v4 Broadwell
18 cores @ 2.3GHz.
dual socket node: 36 core e 128GByte / nodo

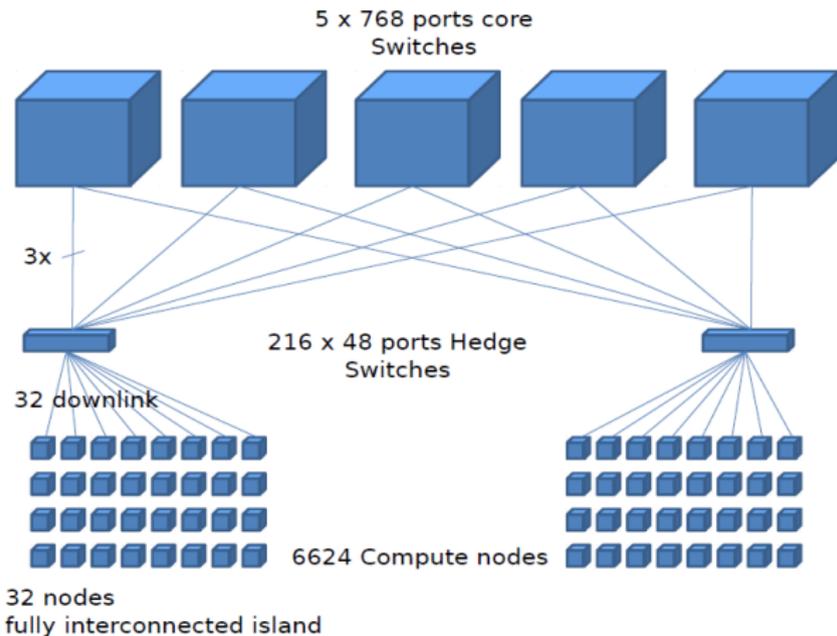
Partizione A2

3600 server Intel AdamPass -> 11PFlops
processore Intel PHI code name Knight Landing
68 cores @ 1.4GHz.
single socket node: 96GByte DDR4 + 16GByte MCDRAM

Partizione A3

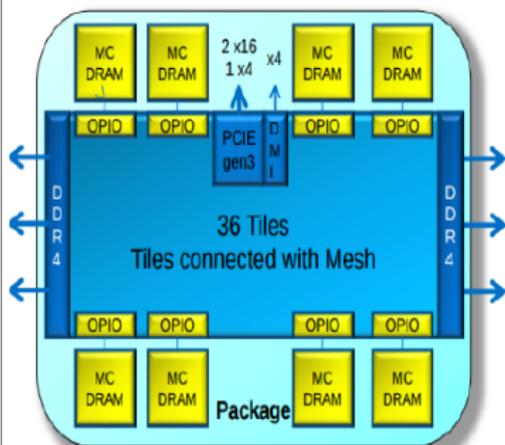
1512 Lenovo Stark Server -> 4.5PFlops
processore Intel E5-2680v5 SkyLake
20 cores @ 2.??GHz
dual socket node: 40 core e 196GByte /nodo

MIMD: Marconi



MIMD: Marconi

Knights Landing Overview



- Stand-alone, Self-boot CPU
- Up to 72 new Silvermont-based cores
- 4 Threads per core. 2 AVX 512 vector units
- Binary Compatible¹ with Intel® Xeon® processor
- 2-dimensional Mesh on-die interconnect
- MCDRAM: On-Package memory: 400+ GB/s of BW²
- DDR memory
- Intel® Omni-path Fabric
- 3+ TFlops (DP) peak per package
- ~3x ST performance over KNC

MIMD: Marconi

	KNC (Galileo)	KNL (Marconi)
#cores	61 (pentium)	68 (Atom)
Core frequency	1.238 GHz	1.4 Ghz
Memory	16GB GDDR5	96GB DDR4 +16Gb MCDRAM
Internal network	Bi-directional Ring	Mesh
Vectorisation	512 bit /core	2xAVX-512 /core
Usage	Co-processor	Standalone
Performance (Gflops)	1208 (dp)/2416 (sp)	~3000 (dp)
Power	~300W	~200W



MIMD: Marconi

Modules/Marconi6.PNG

Outline

Compilatori e ottimizzazione

Floating Point Computing

Makefile

Architectures

Power Performance



PERFORMANCE



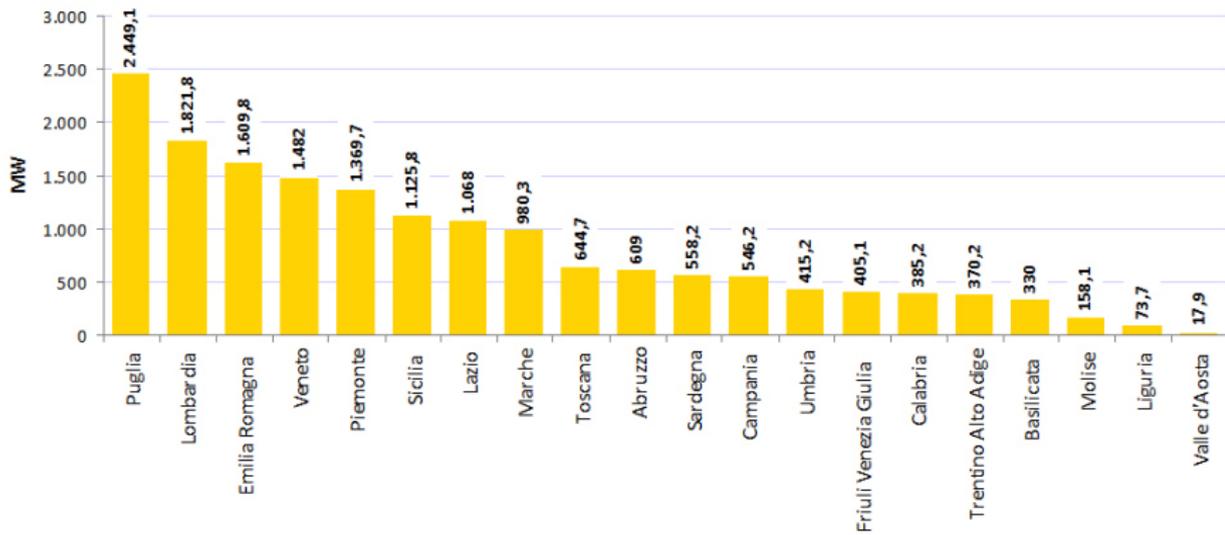
Power Performance

$$\frac{\text{PERFORMANCE}}{\text{POWER}}$$

Power Performance

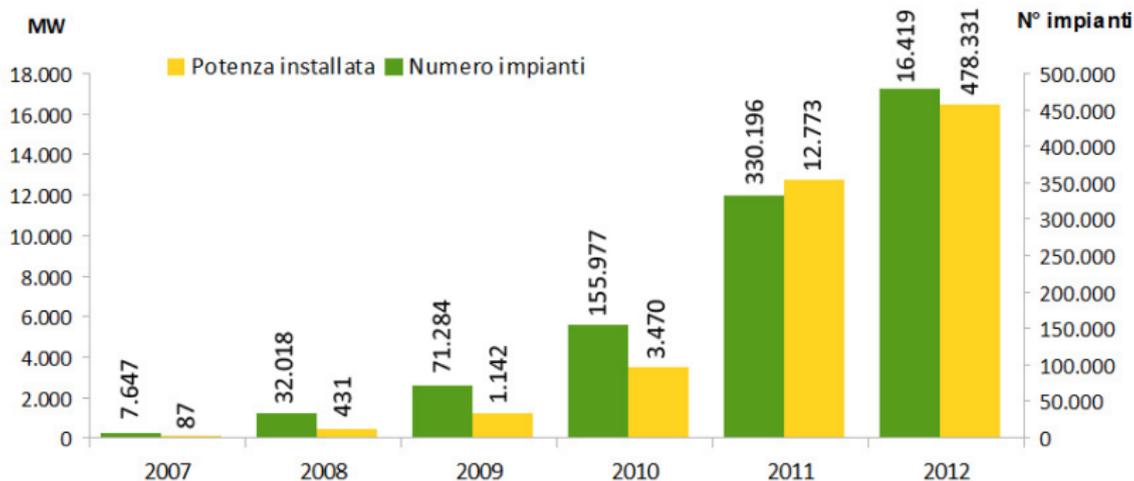


FOTOVOLTAICO: POTENZA INSTALLATA PER REGIONE IN ITALIA (ANNO 2012)



Power Performance

Evoluzione della potenza e della numerosità degli impianti fotovoltaici in Italia





Power Performance

$$\frac{\text{PERFORMANCE}}{\text{POWER}} > 1 \text{ EXAFLOP} < 20 \text{ MWATT}$$



Power Performance

$$\frac{\text{NODE PERFORMANCE} * \#\text{NODES} * \text{CROSS NODE EFICIENCY}}{\text{NODE POWER} * \#\text{NODES} + \text{NETWORK POWER} * \#\text{NODES}}$$



Power Performance

$$\frac{NPERF * NN * CNE}{NODEW * NN + NETWORK * NN}$$



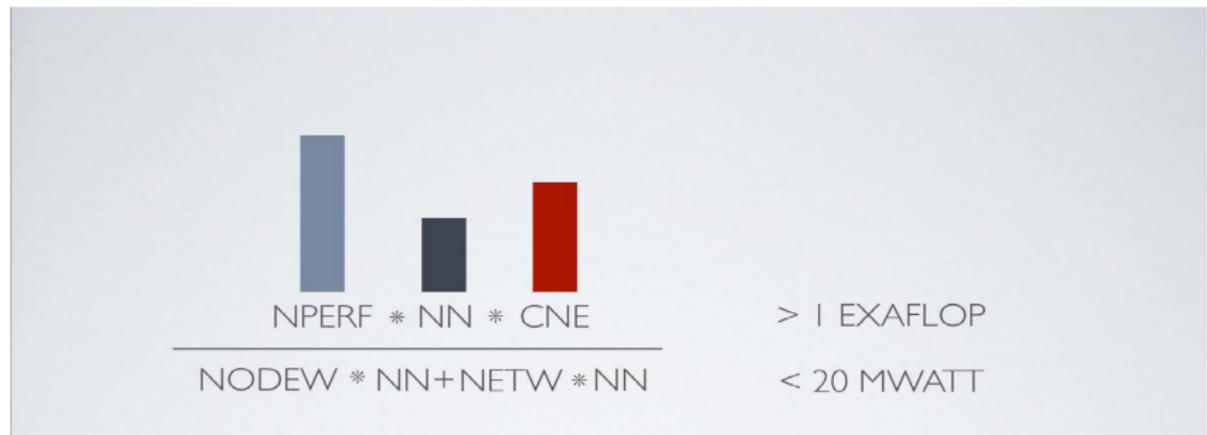
Power Performance

$$\frac{\text{NPERF} * \text{NN} * \text{CNE}}{\text{NODEW} * \text{NN} + \text{NETW} * \text{NN}} \quad > 1 \text{ EXAFLOP}$$

$$\quad \quad \quad < 20 \text{ MWATT}$$



Power Performance

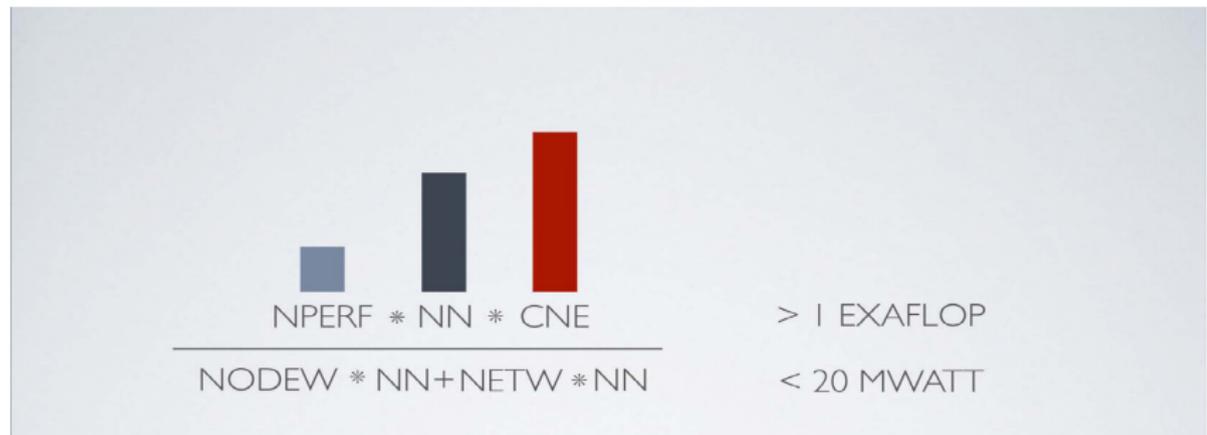


Power Performance





Power Performance

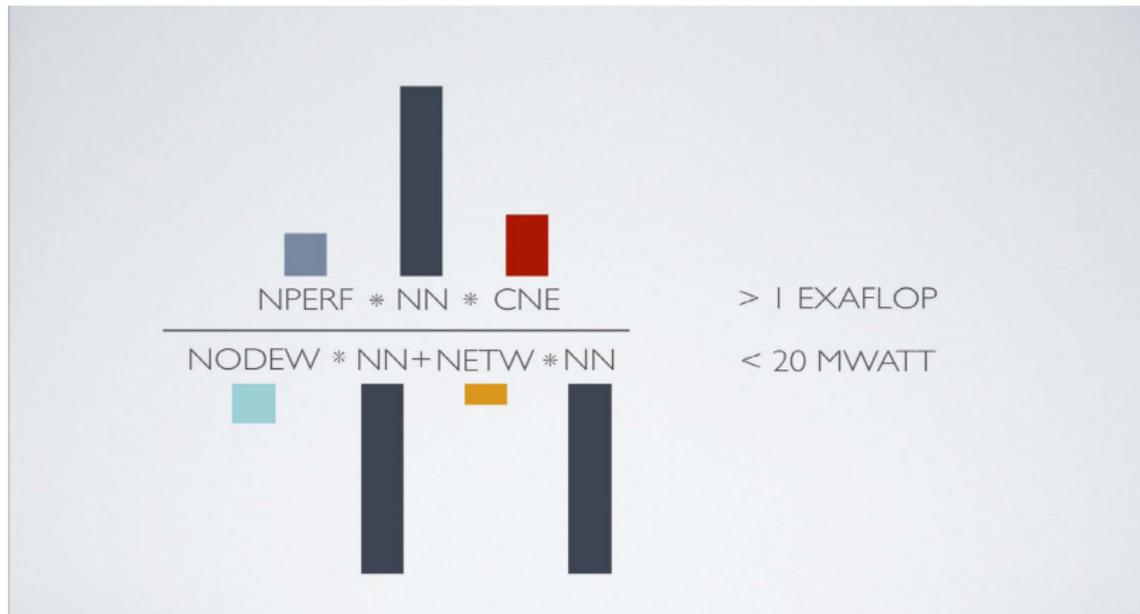




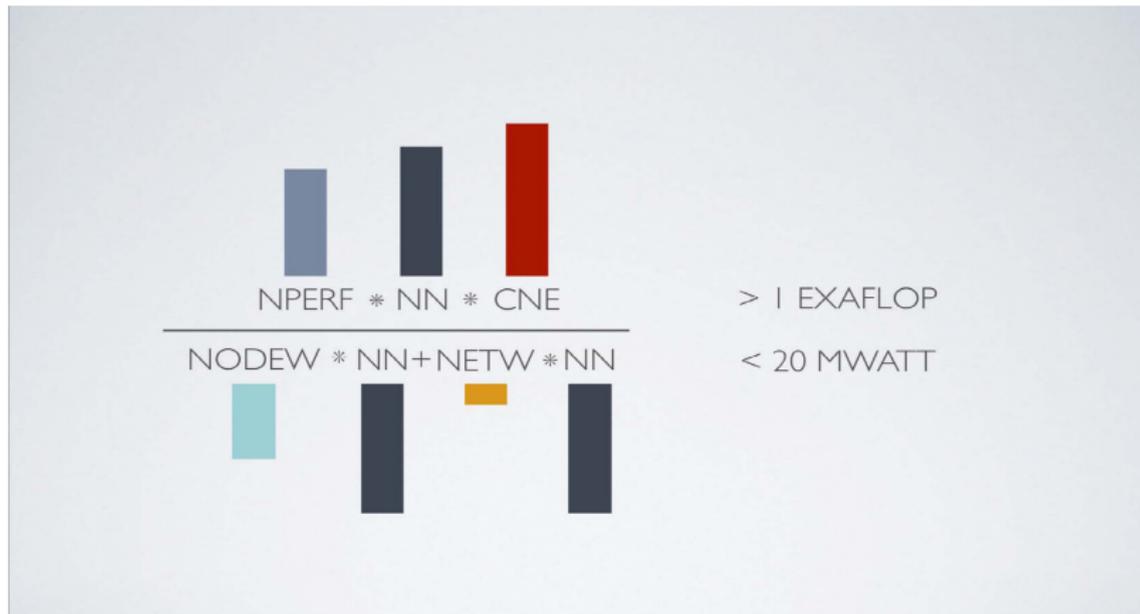
Power Performance



Power Performance



Power Performance





Power Performance

$$\frac{NPERF * NN * CNE}{NODEW * NN + NETWORK * NN} \quad \begin{array}{l} > 1EXAFLOP \\ < 20MWATT \end{array}$$



Power Performance

$$\frac{NPERF * NN * CNE}{NODEW * NN + NETWORK * NN}$$



Power Performance

$NPERF = CORE\ PERF * NUM\ CORES * CROSS\ CORE\ EFFICIENCY$

$$\frac{NPERF * NN * CNE}{NODEW * NN + NETWORK * NN}$$

$NODEW = COREW + MEMW + UNCOREW$



Power Performance

$$NPERF = CPERF * NC * CCE$$

$$\frac{NPERF * NN * CNE}{NODEW * NN + NETWORK * NN}$$

$$NODEW = CW + MW + UCW$$



Power Performance

$$\frac{(C_{PERF} * NC * CCE) * NN * CNE}{(CW + MW + UCW) * NN + NETW * NN}$$



Power Performance

$CPERF = TREAD\ PERF * NUM\ THREADS * CROSS\ THREAD\ EFFICIENCY$

$$\frac{(CPERF * NC * CCE) * NN * CNE}{(CW + MW + UCW) * NN + NETW * NN}$$

$CW = PEAKW * PEAK\% + IDLEW * IDLE\%$

Power Performance

Given:

- $n \in \mathbb{N}$ the number of **blocks of execution**.
- $B \in [0, 1]$ the fraction of the algorithm that is **serial**.

The time $T(n)$ of an algorithm takes to finish when being executed on n threads of execution is:

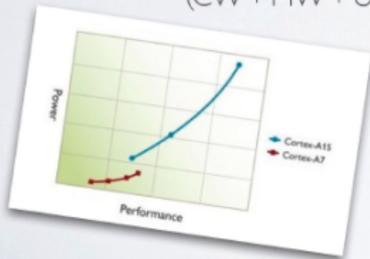
$$T(n) = T(1) \left(B + \frac{1}{n} (1 - B) \right)$$

Therefore, the theoretical speedup that can be had by executing a given algorithm on a system capable of executing in threads is:

$$S(n) = \frac{T(1)}{T(n)} = \frac{T(1)}{T(1) \left(B + \frac{1}{n} (1 - B) \right)} = \frac{1}{B + \frac{1}{n} (1 - B)}$$

$S(P) = P - \alpha \cdot (P - 1)$
where P is the number of processors, S is the **speedup**, and α the non-parallelizable fraction of any parallel process.

$$\frac{(CPERF * NC * CCE) * NN * CNE}{(CW + MW + UCW) * NN + NETW * NN}$$



* PEAK % + ID

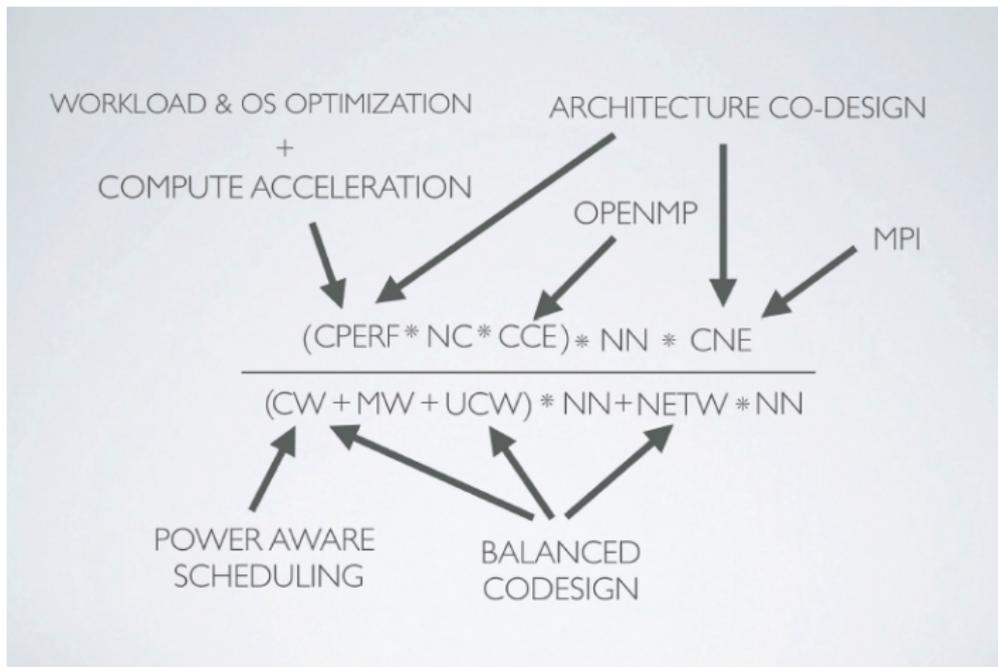




Power Performance

$$\frac{(C_{PERF} * NC * CCE) * NN * CNE}{(CW + MW + UCW) * NN + NETW * NN}$$

Power Performance



Thanks to Chris Adeniyi-Jones (ARM Ltd)