Programmazione Avanzata e parallela

Lezione 06

Luca Manzoni

Ancora branch

E comprendere quando sono utili

- Ricerca binaria con e senza branch
- Ricerca binaria e prefetch
- Sequenza di compilazione
- Loop unrolling
- Function inlining
- Likeliness di un branch

Come costruire la ricerca binaria

Con e senza branch

- La ricerca binaria è il tipico caso in cui la sequenza dei branch è difficile da prevedere (dipende dai valori nel vettore e dal vettore cercato)

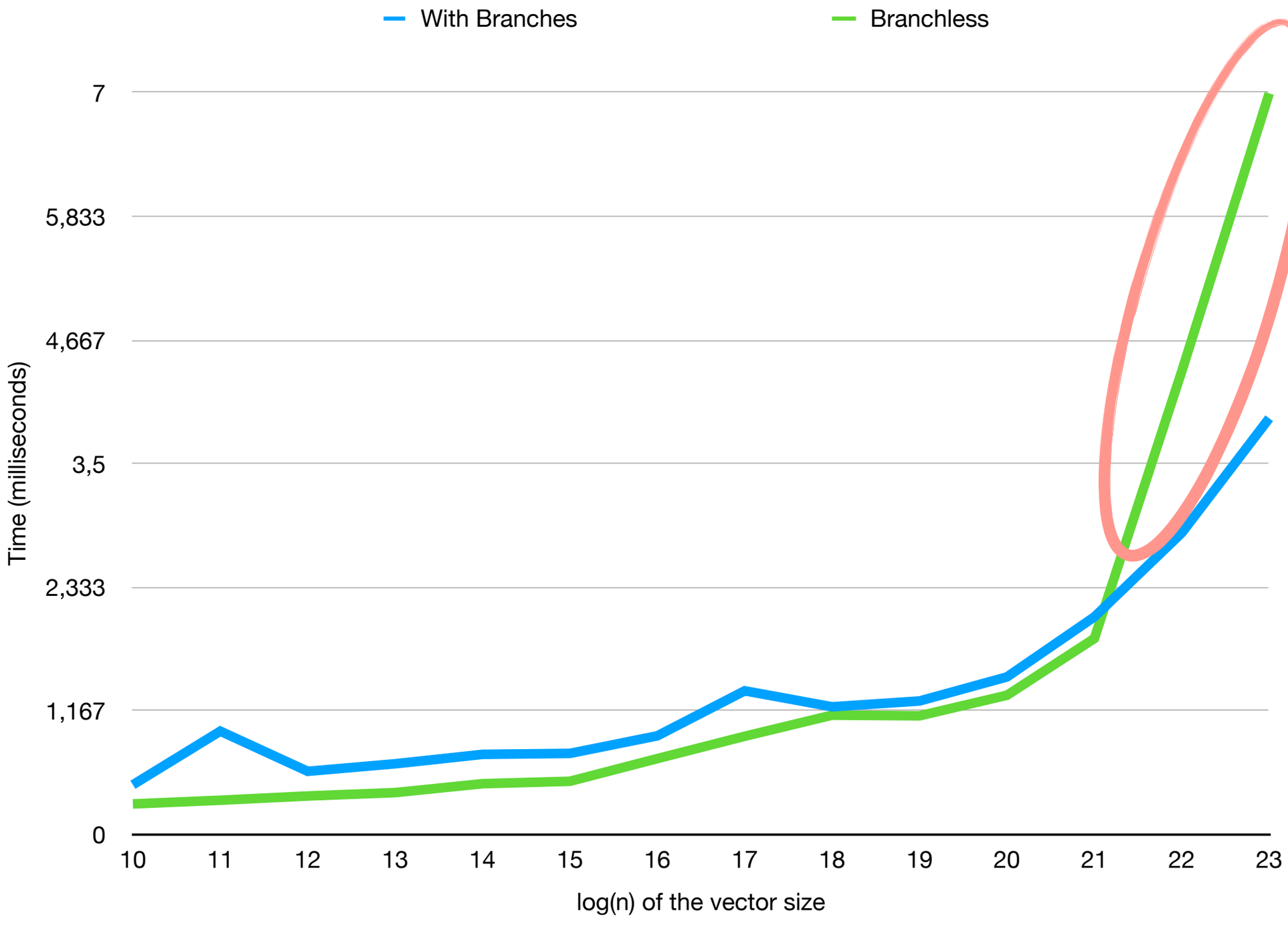
- Possiamo provare a scrivere la ricerca binaria “normale” e una versione senza branch

- **IMPORTANTE**
  il branch dato dall’if nella ricerca binaria viene ottimizzata in modo che sia branchless da parte del compilatore.
  Serve compilare usando gcc (non clang) e l’opzione
  -fno-if-conversion

Ricerca binaria

Con e senza branch

Perché?



Più lenta per array grandi?

Quando i branch aiutano

- Per array grandi inizia a sentirsi un effetto dell'esecuzione speculativa
- Iniziamo ad accedere a v[(l+m)/2] o v[(m+r)/2] prima ancora di aver fatto la scelta se il prossimo indice sarà l'uno o l'altro. Quando indoviniamo lo abbiamo già caricato dalla memoria
- Nel caso branchless invece dobbiamo attendere di aver trovato il prossimo indice per poter proseguire (è un tipo diverso di hazard, quindi niente esecuzione speculativa)
- Per questioni di gerarchia di memoria questo effetto diventa importante su array grandi

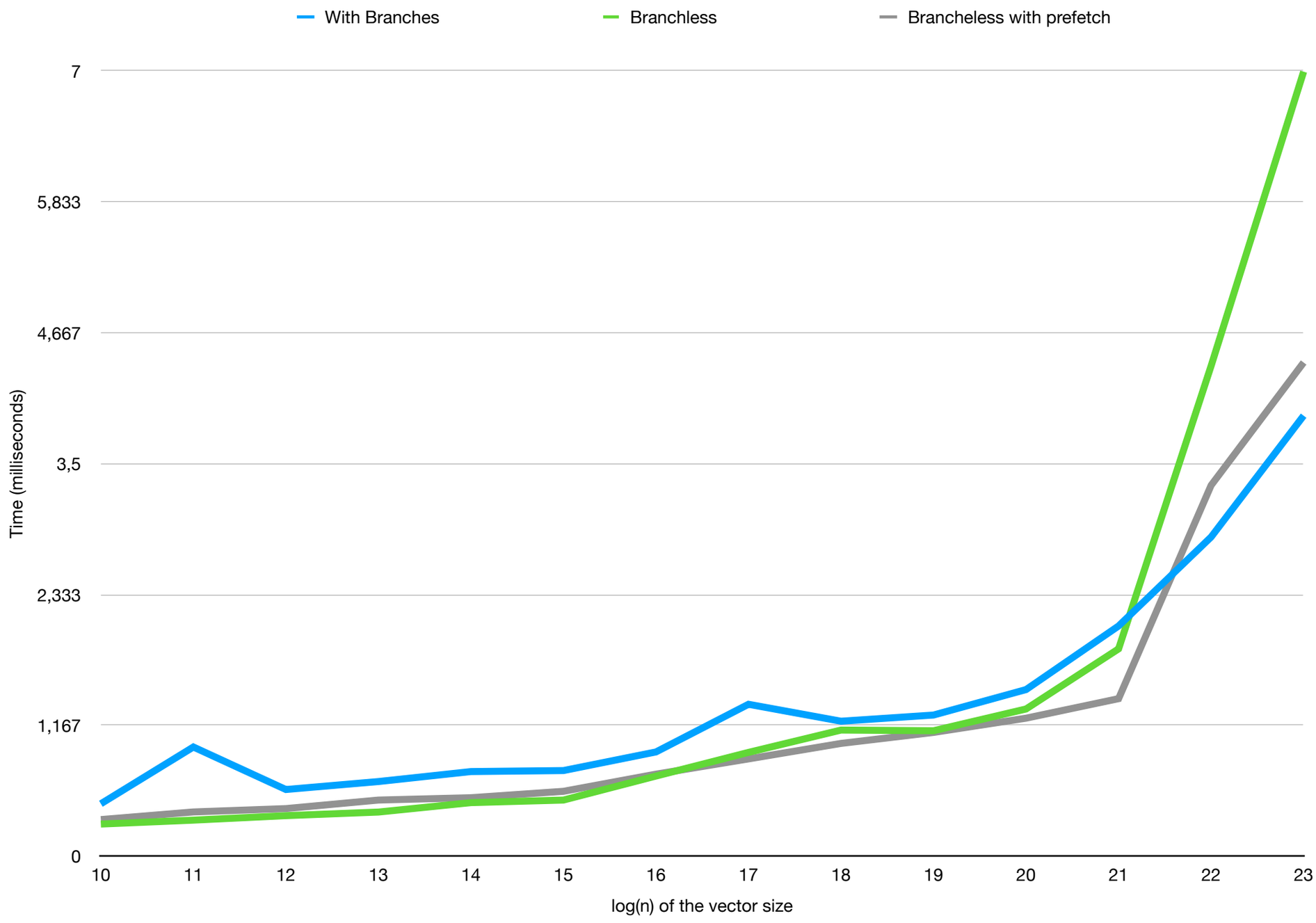
Prefetch

Quando i branch aiutano

- Come possiamo fare?
- Possiamo iniziare a dire al sistema che alcuni indirizzi di memoria è probabile che vengano richiesti successivamente
- Questo è tramite una funzione “builtin” del compilatore: *__builtin_prefetch(void *)*
- Questa direttiva informa il sistema di memoria che gli indirizzi passati come argomento potrebbero essere necessari a breve
- Questo solitamente implica che inizierà a richiedere il loro contenuto

Ricerca binaria

Con e senza branch (+ prefetch)



Sequenza di compilazione

Parte 1

- **Preprocessing**. Espansione delle macro, inserimento dei file header, rimozione dei commenti
  - `gcc –E source.c`
    stampa su standard output
- **Compilazione**. Parsing del sorgente, controllo degli errori di sintassi, trasformazione in una rappresentazione intermedia, ottimizzazione, generazione dell'assembly.
  - `gcc –S source.c`
    genera un file source.s in assembly dell'architettura per cui si compila

Sequenza di compilazione

Parte 2

- **Assembly**. Trasforma l'assembly in codice macchina, tutte le chiamate a funzioni esterne sono però sostituite da dei placeholders.
  - `gcc -c source.c`
    genera il file source.o

- Linking. Genera il file eseguibile sostituendo ai placeholders gli indirizzi corretti e inserendo il codice nell'eseguibile (assumendo di non usare librerie dinamiche)

  - `gcc -o binary source1.o source2.o`
    genera il file eseguibile

Compilazione: check dell'output

Comprendere cosa viene generato

- Spesso è utile comprendere quale sia il codice generato dal compilatore per vedere se:
  - Alcune ottimizzazioni che pensiamo di fare siano già date dal compilatore (e.g., il codice viene già generato senza branch)
  - Il codice che viene generato sia corrispondente a quello che ci aspettiamo
- È possibile passare -fverbose-asm per ottenere del codice assembly con commenti del compilatore
- Compiler explorer (https://godbolt.org/) è molto utile per comprendere come viene generato l'assembly

Ottimizzazioni

Comprendere cosa viene generato

- Molte ottimizzazioni sono “automatiche” (vengono fatte specificando semplicemente al compilatore di ottimizzare)

- Alcuni esempi:

  - **Propagazione delle costanti**: se è possibile computare un valore durante la compilazione posso direttamente inserire il risultato

  - **Rimozione del dead-code**: rimozione del codice che non va a influenzare il comportamento del programma
    (ricordate che è per questo che negli esempi stampiamo, per esempio, il valore di alcune variabili)

Ottimizzazioni

Comprendere cosa viene generato

- Esistono decine (almeno) di altre ottimizzazioni
- Spesso sono applicate usando euristiche, dato che comportano dei compromessi (e.g., velocità vs dimensione in memoria)
- Alcune di queste ottimizzazioni possono essere abilitate o disabilitate con opzioni del compilatore
  - Esempio: *-fno-if-conversion* per disabilitare la trasformazione del code in branchless (solo gcc, non clang)
- Per altre ottimizzazioni possiamo provare a ottimizzare noi

Ottimizzazioni

Loop unrolling

- Per cicli il cui numero di iterazioni è piccolo (e noto a priori) è possibile rimpiazzare il branch per tornare all'inizio del ciclo ripetendo il corpo del ciclo tante volte quanto il numero di iterazioni
- Alternativamente è comunque possibile "srotolare" parzialmente un ciclo:
  - E.g., un ciclo di 1000 iterazioni nella forma *v[i]++;* è sostituito da 100 iterazioni della forma *v[i]++; v[i+1]++; … v[i+9]++;*
- Questa ottimizzazione può (o no) rendere l'esecuzione più rapida

Ottimizzazioni

Loop unrolling

- Questa ottimizzazione può essere abilitata con "*-funroll-loops*" e disabilitata con "*-fno-unroll-loops*"

- Questo non obbliga il compilatore a "srotolare" (dipende da una serie di euristiche) un ciclo ed è una ottimizzazione che può essere già abilitata

- All'interno del codice si può utilizzare un comando specifico del compilatore:
  #pragma GCC unroll 16
  per forzare a "srotolare" il ciclo 16 volte.

- La pragma può essere specifica per i diversi cicli e forza il compilatore anche contro le euristiche che usa

Ottimizzazioni

Function inlining

- Il “function inlining” consiste nel sostituire una chiamata a funzione col corpo della funzione
- Questo evita di dover fare una chiamata a funzione (con salvataggio di registri, modifica dello stack pointer, etc)
- Ma ha lo svantaggio che se ho più punti in cui la funzione è chiamata il codice è duplicato
- Il compilatore può decidere di fare inlining…
- … ma possiamo dare degli hint con la keyword “inline” prima della definizione della funzione

Ottimizzazioni

Likeliness di un branch

- Se per qualche motivo sappiamo che un branch è probabile che sia preso (o non preso) possiamo indicarlo al compilatore
- Gcc fornisce *__builtin_expect()*, che prende due argomenti:
  - Il primo è una espressione da valutare (e.g., $i < n$)
  - Il secondo è il valore atteso dell'espressione (e.g., 1 per vero e 0 per falso)
- Il compilatore può provare ad applicare ottimizzazioni diverse assumendo che il valore dell'espressione sia spesso quello indicato