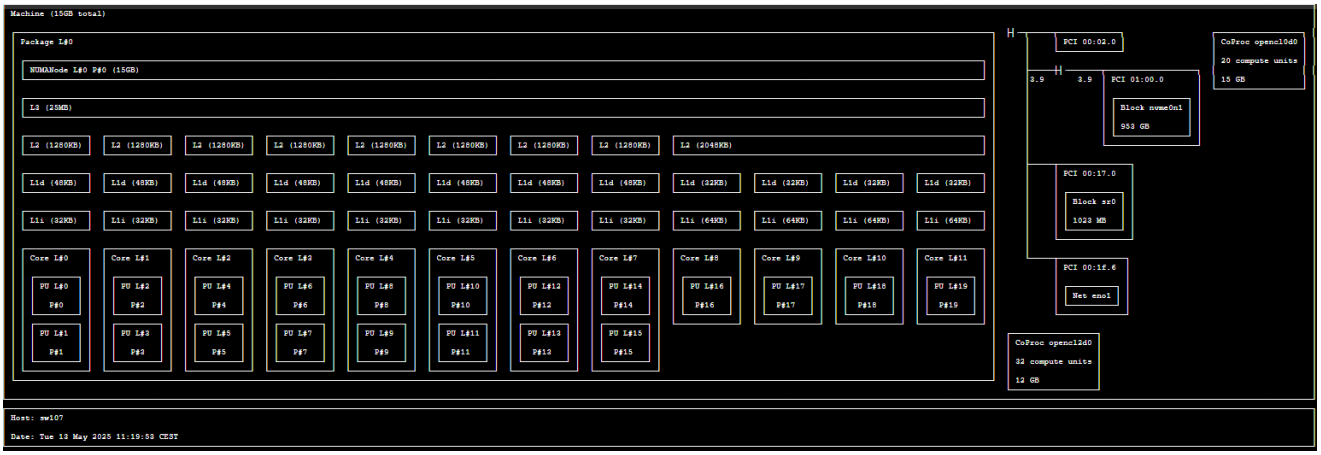
Tramonta Emanuele 7837041

AdC Compito 3



- Abbiamo un totale di 12 core con la cache L3 condivisa tra tutti, tra questi:
  - 8 hanno cache L2 dedicata, e cache L1 divisa per istruzioni e dati, e 2 Processing Units l'uno. Sono Performance Core.
  - 4 hanno una singola L2 condivisa, e L1 individuale divisa per istruzioni e dati, con 1 PU l'uno. Sono Efficiency Core.
  - Abbiamo quindi un totale di $(8 \times 2) + 4 = 20$ thread

Esecuzioni

- `gcc Driver.c Multiply.c  -o Mult -lm`
  - Exec time: 80.893 seconds
  - GigaFlops: 0.998875
  - Sum of results: 781703.999799
- `gcc -O3 Driver.c Multiply.c  -o Mult -lm`
  - Exec time: 18.088
  - GigaFlops: 4.467122
  - Sum of results: 781703.999799
    - con `-O3` il compilatore compila con tecniche come loop unrolling e modifiche all'ordine delle istruzioni per evitare tempi morti come abbiamo visto a lezione, oltre ad altre tecniche più avanzate, risultando in tempi di esecuzione minori.
    - Di conseguenza, abbiamo tempi di esecuzione drasticamente minori, quindi più GigaFlops, mantenendo la correttezza del risultato.

- `icc -diag-disable=10441 Driver.c Multiply.c  -o Mult`
  - Exec time: 16.864
  - GigaFlops: 4.791348
  - Sum of results: 781703.999799
    - In questo caso stiamo usando il compilatore Intel, quindi ottimizzato per CPU Intel, che anche senza flag di ottimizzazione ottiene risultati migliori di `gcc -O3`.
- `icc -diag-disable=10441 -g -O2`
  - Exec time: 16.844
  - GigaFlops: 4.797010
  - Sum of results: 781703.999799
    - Aggiungendo `-O2` abbiamo un piccolo aumento in prestazioni. Evidentemente la maggior parte delle ottimizzazioni rilevanti al nostro codice erano già implementate su `icc` base, o meglio `icx` dato il flag `-diag-disable=10441`
    - `-g` è solo per avere informazioni sui simboli del programma messi nell'eseguibile per debugging con tool esterni.

Analisi tool Intel e .optrpt con vettorizzazione

- **Pre-vettorizzazione**
  - Roofline:
    - L'hotspot è il loop in Multiply.c:55, operando a 4.94GFLOPs in 16.276 secondi.
    - Un hotspot minore è in linea 45, operando a 0.739 GFLOPS in 0.544.
    - Il programma riporta che il loop è scalare e non vettorizzato, presumibilmente data una dipendenza data dal possibile aliasing tra i vettori/matrici.
  - VTune
    - L'utilizzazione dei core è 8.2%, meno di un processore su 12, ovvero usa un singolo processore, e neanche completamente
    - Analogamente, la Logical Core Utilization è 2.079 (10.4%) su 20, 2 core logici equivalenti presumibilmente al singolo core fisico di cui prima.
    - La Vectorization è allo 0%, ovviamente, e nei DP FLOPs il 100% è scalare. Nei SP FLOPs c'è un 7.9% packed 128-bit.
    - L'utilizzo della microarchitettura è 22.1%, presumibilmente in quanto non abbiamo ancora specificato `-march=alderLake` e non può usare tutte le istruzioni a disposizione.
    - Abbiamo un misero 10.4% di threading.

- **Post-vettorizzazione**
  - Roofline:
    - Hotspot in Driver.c:151 vettorizzato con istruzioni AVX, con performance 25.443 GFLOPS, molto meglio di prima, in 3.16 secondi
    - Hotspot in Driver.c:149, con performance 2.033 GFLOPS, meglio del 2° hotspot di prima, in 0.89 secondi
  - VTune
    - Ho perso i file del VTune post-vect perchè sono incompetente, ma mi aspetto di avere simile utilizzo dei core in quanto la vettorizzazione riguarda le singole CPU, per usare più CPU dovremmo usare parallelizzazione.
    - La Vectorization e il threading li immaginerei molto più alti.
    - L'utilizzo della microarchitettura anche, grazie al flag `-march=alderLake`.
  - .optrpt
    1. Primo .optrpt:
       - Il loop in Multiply:55, precedentemente hotspot senza vettorizzazione non è stato vettorizzato
         - `remark #15344: loop was not vectorized: vector dependence prevents vectorization. First dependence is shown below. Use level 5 report for details`
         - `remark #15346: vector dependence: assumed FLOW dependence between b[i] (56:4) and b[i] (56:4)`
           - Per quanto riesca a ragionare sul codice sorgente, siccome non è stato usato NOALIAS come flag nella compilazione, il compilatore pensa che `b` possa essere alias di `a` o `x`, dunque in maniera conservativa evita di vettorizzare. Presumo che usando `-DNOALIAS` lo si convinca che sono distinti e vettorizzerebbe a meno di altre ambiguità.
       - Il loop in Multiply:45, precedentemente hotspot minore, neanche, presumibilmente in quanto loop esterno, oltre a:
         - `remark #15541: outer loop was not auto-vectorized: consider using SIMD directive`
         - Sembra suggerire di usare una direttiva Single Instruction Multiple Data
       - In Driver.c invece, tutti i loop sono vettorizzati tranne:
         - `LOOP BEGIN at Driver.c(54,2) inlined into Driver.c(141,2)`
           - `remark #15542: loop was not vectorized: inner loop was already vectorized`
           - Driver.c:54 in quanto loop esterno
         - `LOOP BEGIN at Driver.c(146,2)`
           - `remark #15543: loop was not vectorized: loop with function call not considered an optimization candidate.`

        - Driver.c:146 in quanto contiene chiamata di funzione, quindi loop considerato non vettorizzabile
2. `-DNOFUNCCALL` .optrpt:
    - Multiply.c:
        - Stesso comportamento, Multiply.c non ha alcun `#ifdef NOFUNCCALL`, e comunque eventuali cambiamenti sarebbero irrilevanti in quanto `matvec(args)` non viene chiamato
    - Driver.c:
        - Come vediamo nel sorgente, con `NOFUNCCALL` il codice di `matvec(args)` è eseguito direttamente inline invece che con chiamata di funzione a `matvec(args)`:
            - `LOOP BEGIN at Driver.c(146,2)`
                - `remark #15542: loop was not vectorized: inner loop was already vectorized`
                - inner loop, non viene vettorizzato
                - `LOOP BEGIN at Driver.c(149,3)`
                    - `remark #15542: loop was not vectorized: inner loop was already vectorized`
                    - idem
                    - `LOOP BEGIN at Driver.c(151,4)`
                        - `remark #15300: LOOP WAS VECTORIZED`
                        - loop vettorizzato, al contrario di prima in quanto chiamava funzione
3. `-march=alderLake` .optrpt:
    - Multiply.c:
        - Nessuna differenza, il problema del potenziale aliasing (e dunque dipendenza dei dati) non è risolto specificando l'architettura. Inoltre, `-DNOFUNCCALL` previene la chiamata di `matvec(args)`.
    - Driver.c:
        - Noto che le linee `vector cost` riportano valori più bassi, data la compilazione ad hoc per istruzioni supportate da estensioni dell'architettura Alder Lake come AVX2 (secondo `lscpu`), invece che limitandosi alle generiche istruzioni x86-64. In particolare, sembrano essere circa dimezzate, denotando, presumibilmente, un dimezzamento dei tempi di esecuzione, e un (circa) raddoppio dello speedup potenziale.

Parallelizzazione

1. Tempo di esec dopo cambio `ROW`, `COL`:
   - Time 77.435
   - GigaFlops = 16.571298
   - Sum of res: 12486803.999199
     - Il tempo di esecuzione è maggiore dato l'incremento bidimensionale delle dimensioni della matrice, comunque ottimo.
     - Abbiamo quasi quadruplicato i GigaFlop, che è ottimo.
     - Non posso commentare sul risultato in quanto confermare la validità di così tanti calcoli manualmente è poco pratico, è però importante ricordare la possibilità di errore dato dal parallelismo, nonostante il comportamento conservativo del compilatore.
     - .optrpt:
       - Multiply.c:
         - Nessuna differenza, aumentare dimensioni matrice non risolve ne peggiora dipendenze di dati.
       - Driver.c:
         - Leggero aumento `potential speedup` nella maggior parte delle vettorizzazioni. Immagino che avendo più righe e colonne abbia più "spazio" o "occasioni", per ignoranza di terminologia adeguata, per loop-unrolling e tecniche di vettorizzazione.
2. Tempo di esec finale con `#pragma omp parallel for private(j)`:
   - 31.347sec
   - GF: 40.935387
   - Sum: 12486803.999199
     - Idem come prima per il risultato.
     - .optrpt:
       - Multiply.c:
         - `-DNOFUNCCALL` previene la chiamata a `matvec(args)`, quindi ogni cambiamento sarebbe irrilevante, non che ce ne sia alcuno
       - Driver.c:
         - Gli `speedup` diminuiscono, per esempio nei loop inline che sostituiscono `matvec(args)`, dove va da 4.410 a 3.770. Ciò mi fa pensare che questo speedup consideri solo la vettorizzazione (SIMD) e non la parallelizzazione, quindi lo speedup finale potrebbe essere $\approx 3.770 \times \text{core in uso}$, ma dubito il calcolo sia così semplice.