

LoopFusion

Matteo Lugli, Carlo Uguzzoni

May 22, 2023

Contents

1	Requisiti per la loop fusion	2
1.1	Adiacenza	2
1.2	Trip count	2
1.3	Dipendenze	2
2	Esecuzione della loop fusion	2
3	Attività sperimentali supplementari	6
3.1	Misura delle performance	6
3.2	Determinazione empirica della dimensione delle cache	7

1 Requisiti per la loop fusion

Si supponga che si stiano prendendo in considerazione i due loop CFG equivalenti L_i e L_k .

1.1 Adiacenza

L'adiacenza dei due loop viene verificata controllando (i) che l'**exit block** di L_i corrisponda all'**header** di L_k (ii) e che il suddetto blocco di intersezione contenga soltanto un'istruzione di tipo **branch**, la cui destinazione deve essere necessariamente il **body** di L_k . Esiste anche il caso in cui i due loop non sono immediatamente adiacenti, ma lo potrebbero diventare applicando adeguate operazioni di trasformazione. Questo non viene gestito per semplicità, dato che richiederebbe un'analisi piuttosto avanzata.

```
if (L1->getExitBlock() == L2->getLoopPreheader()) {
    int instruction_count = 0;
    BasicBlock *MiddleBlock = L1->getExitBlock();

    for (auto iter_block = MiddleBlock->begin();
         iter_block != MiddleBlock->end();
         ++iter_block) {
        ++instruction_count;
    }
    if (instruction_count != 1) {
continue;
    }
    Instruction *I = dyn_cast<Instruction>
(MiddleBlock->begin());
    BranchInst *BI = dyn_cast<llvm::BranchInst> (I);
    if (!(BI && BI->getSuccessor(0) == L2->getHeader())) {
continue;
    }
}
```

1.2 Trip count

Serve inoltre verificare che L_i e L_k eseguano in ogni caso lo stesso numero di iterazioni, quindi che abbiano lo stesso trip count. Per eseguire questo controllo è necessaria un'analisi preliminare, che in LLVM prende il nome di **ScalarEvolutionAnalysis**. Si può ottenere chiamando l'apposito metodo del **FunctionAnalysisManager**:

```
auto &SE = AM.getResult<ScalarEvolutionAnalysis>(F);
```

¹Basic Block

1.3 Dipendenze

2 Esecuzione della loop fusion

Se L_i e L_k hanno superato tutti i controlli, si può procedere ad effettuare la loop fusion. In LLVM il modo più semplice per modificare il *control flow* del programma è cambiare la direzione degli archi del grafo. Le operazioni da eseguire sono le seguenti: (i) collegare l'ultimo BB¹ del *body* di L_i al primo BB del *body* di L_k ,

```
// save this final block for later
BasicBlock *FINAL = L2->getExitBlock();

BasicBlock *LL1 = L1->getLoopLatch();
BasicBlock *LB1 = LL1->getPrevNode();
BasicBlock *HB2 = L2->getHeader();
Instruction* L2PHI = dyn_cast<Instruction>(HB2->begin());
Instruction *I1 = HB2->getTerminator();
BranchInst *BI1 = dyn_cast<llvm::BranchInst>(I1);
BasicBlock *LB2 = BI1->getSuccessor(0);
Instruction *I2 = LB1->getTerminator();
BranchInst *BI2 = dyn_cast<llvm::BranchInst>(I2);
BI2->setSuccessor(0, LB2);
```

(ii) il *body* di L_k al *latch* di L_i ,

```
BasicBlock *LL2 = L2->getLoopLatch();
BasicBlock *LB2E = LL2->getPrevNode();
Instruction *LB2E_branch = LB2E->getTerminator();
LB2E_branch->setSuccessor(0, LL1);
```

(iii) il ramo *false* dell'*header* di L_i con l'*exit block* di L_k .

```
BasicBlock *HB1 = L1->getHeader();
Instruction *I4 = HB1->getTerminator();
BranchInst *BI4 = dyn_cast<llvm::BranchInst>(I4);
BI4->setSuccessor(1, FINAL);
```

Le trasformazioni sono ben visibili in Figura 1 e in Figura 2. Infine si sostituiscono tutti gli usi della *induction variable* di L_k con l'*induction variable* di L_i , in modo da usarne soltanto una per gestire le operazioni di entrambi i loop, che ora possono essere considerati uno solo.

```

Instruction* L1PHI = dyn_cast<Instruction>(HB1->begin());
Value *CastedL1PHI = dyn_cast<Value>(L1PHI);
L2PHI->replaceAllUsesWith(CastedL1PHI);
L2PHI->eraseFromParent();

```

A seguito della qui discussa ottimizzazione, alcuni BB rimangono scollegati dal resto del CFG, risultando quindi inaccessibili. In Figura 2 compaiono nella categoria "Garbage Blocks". Da notare in particolare il *latch* di L_k , che se lasciato avrebbe comportato il doppio incremento della *induction variable*.

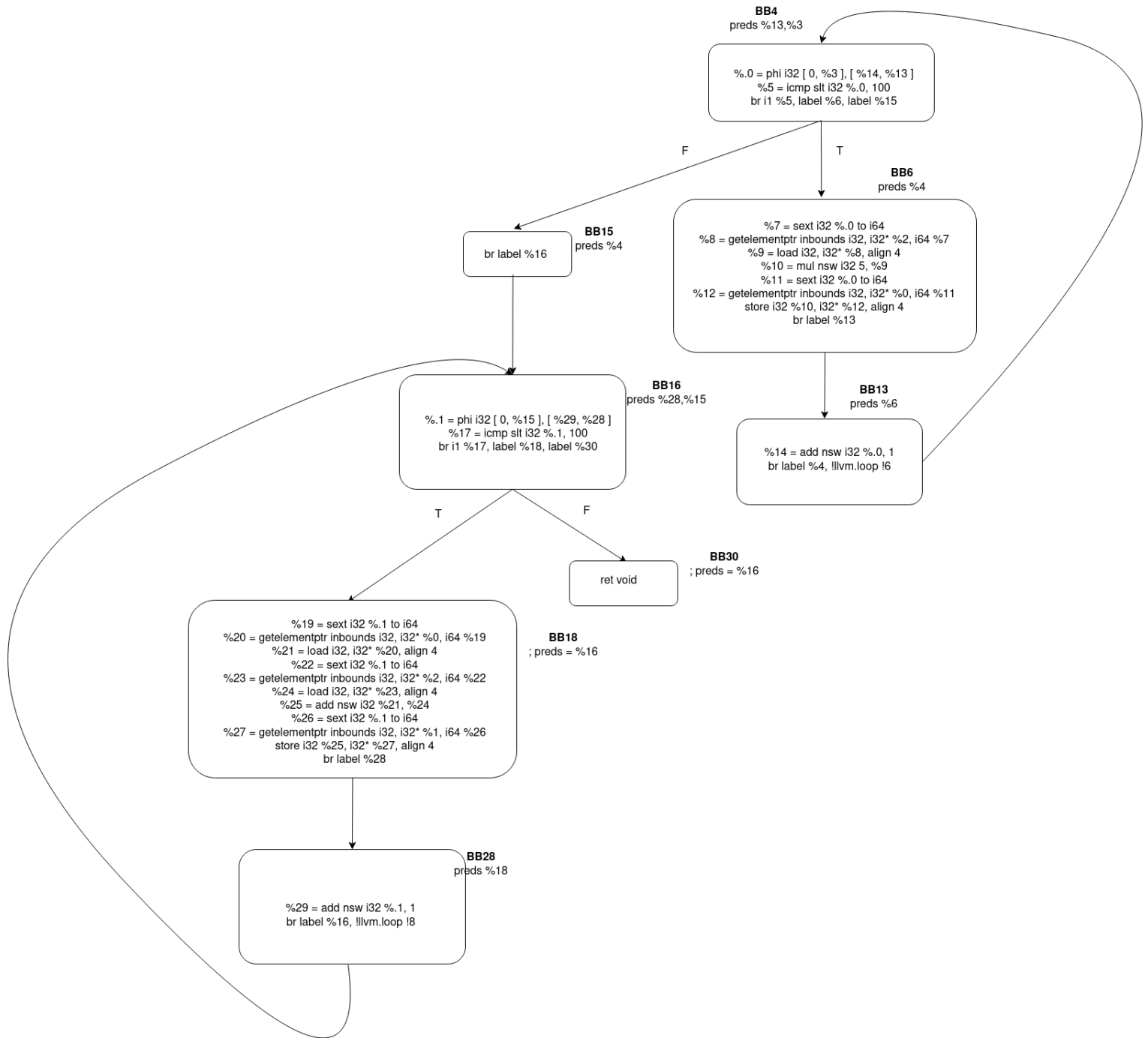


Figure 1: CFG prima del passo di ottimizzazione

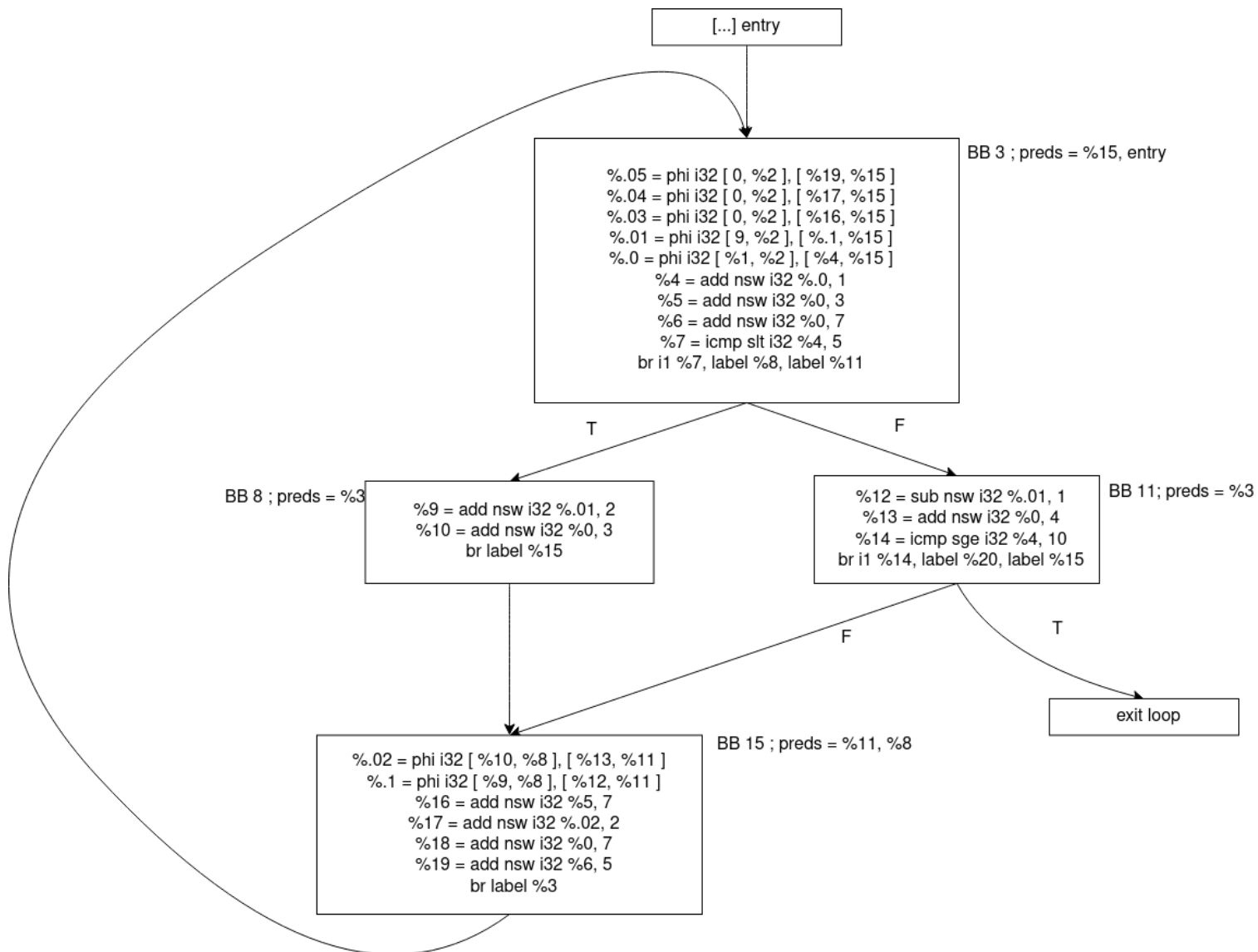


Figure 2: CFG dopo il passo di ottimizzazione

3 Attività sperimentali supplementari

Al termine dell'esperienza, si sono poi svolti i seguenti lavori aggiuntivi:

- Confronto delle performance di uno stesso programma, ottimizzato tramite il passo realizzato e non
- Sperimentazione volta a determinare la dimensione delle linee di cache sul medesimo calcolatore usato per svolgere l'esperienza

3.1 Misura delle performance

In primo luogo, si sono ottenuti due file oggetto da *Loop.c*, l'uno servendosi del passo di loop fusion appositamente creato, e l'altro con il comando *llc*, senza fornire particolari flag (dunque non richiedendo esplicitamente ottimizzazioni).

Tali file oggetto sono stati poi linkati in successione ad un semplice script di profiling scritto in C. Il profiler, richiamando la funzione *populate* definita in *Loop.c* in loop esterni per un numero crescente i di iterazioni, andava poi a misurare il tempo di esecuzione, per ogni i . I valori di i sono stati impostati come potenze del 10, per esponente intero $\in [0, 9]$.

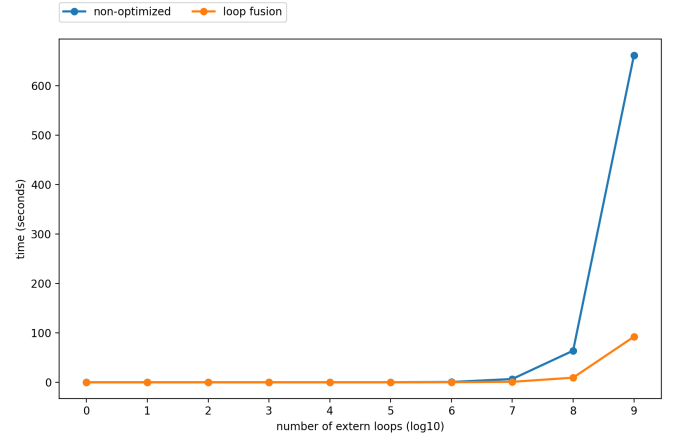


Figure 3: Profiling - confronto tra loop fusion e non-ottimizzazione, $ODG \in [1, 10]$

Pur essendo chiara la tendenza di miglioramento delle performance nel caso di loop fusion per $i = 10^j, j \in [7, 9], j \in \mathbb{N}$, si è reso necessario verificare l'andamento per valori di i più bassi, non visibile di fatto su quest'ultimo grafico.

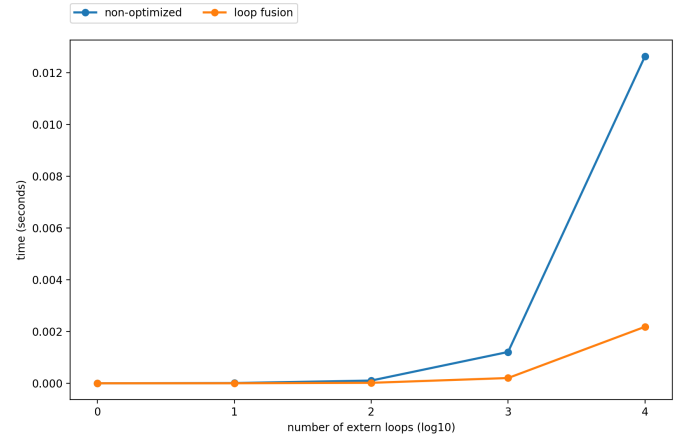


Figure 4: Profiling - confronto tra loop fusion e non-ottimizzazione, $ODG \in [1, 5]$

Infine, si è osservato il comportamento del profiler tra i primi 3 ordini di grandezza, confermando quanto già visto.

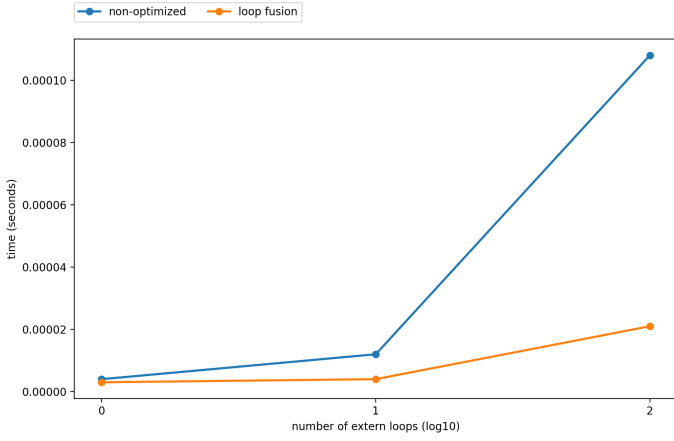


Figure 5: Profiling - confronto tra loop fusion e non-ottimizzazione, $ODG \in [1, 3]$

Il tempo di esecuzione ha una tendenza di crescita esponenziale per entrambe le linee. Si è quindi deciso di eseguire un ulteriore plot, riportando sull'asse y il logaritmo del tempo di esecuzione, per meglio osservare il rapporto tra i tempi di esecuzione ad uguali valori di i .

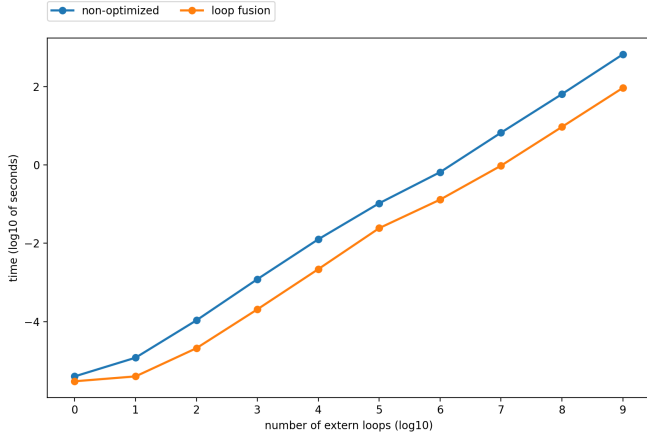


Figure 6: Profiling - confronto tra loop fusion e non-ottimizzazione, $\log_{10} t$ su asse y

Dopo un iniziale assestamento, ed al crescere di i , il rapporto tra le prestazioni con e senza loop fusion si conserva circa stabile (esecuzione $\sim 5 - 7$ volte più rapida con loop fusion).

3.2 Determinazione empirica della dimensione delle cache

Seguendo la proposta del docente, ci siamo proposti di stimare in modo empirico la dimensione della cache del calcolatore su cui abbiamo eseguito l'esperienza (CPU Ryzen 5 2500U).

Sostanzialmente, ci siamo serviti di un profiler analogo al precedente, linkandolo con la versione ottimizzata di *Loop.c*. La funzione *populate* è stata richiamata dal profiler, eseguendo 10^4 volte ad ogni misura del tempo, per uno stride s crescente ($s = 2^i, i \in [0, 16], i \in \mathbb{N}$). La dimensione degli array di partenza è stata impostata come $size = base_size * stride$, così da avere un numero di accessi uguale, ed un tempo di esecuzione simile al netto delle cache miss, per ogni valore di s .

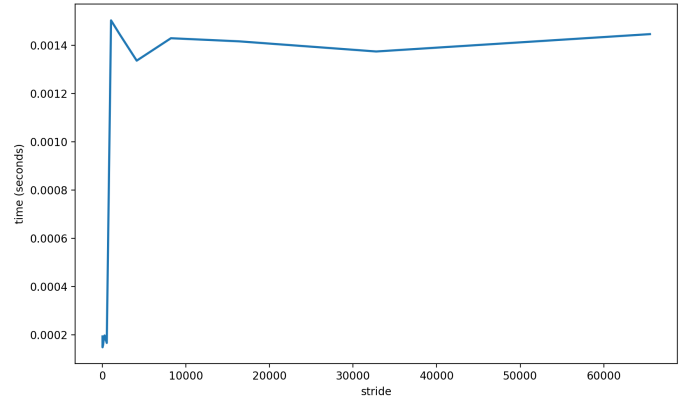


Figure 7: Cache test - stride vs execution time

Lo "scalino" presente in corrispondenza di $s = 1024$ sembrerebbe indicare che di lì in poi si abbiano sole cache miss, dunque la dimensione della singola linea di cache sarebbe 4kb. Ciò non di meno, l'incremento di execution time in $s = 4, 8, \dots, 64$ sembra decisamente irrilevante per quello che ci si aspetterebbe dalle cache penalties, come si può vedere nel seguente tabulato.

Stride: 1	Array size: 4	Exec time: 0.000193
Stride: 2	Array size: 8	Exec time: 0.000173
Stride: 4	Array size: 16	Exec time: 0.000155
Stride: 8	Array size: 32	Exec time: 0.000148
Stride: 16	Array size: 64	Exec time: 0.000162
Stride: 32	Array size: 128	Exec time: 0.000155
Stride: 64	Array size: 256	Exec time: 0.000156
Stride: 128	Array size: 512	Exec time: 0.000174
Stride: 256	Array size: 1024	Exec time: 0.000197
Stride: 512	Array size: 2048	Exec time: 0.000166
Stride: 1024	Array size: 4096	Exec time: 0.001504
Stride: 2048	Array size: 8192	Exec time: 0.001447
Stride: 4096	Array size: 16384	Exec time: 0.001337
Stride: 8192	Array size: 32768	Exec time: 0.001430
Stride: 16384	Array size: 65536	Exec time: 0.001417
Stride: 32768	Array size: 131072	Exec time: 0.001375
Stride: 65536	Array size: 262144	Exec time: 0.001447

Figure 8: Cache test - tabulato dati raccolti

Non si esclude che vengano svolte ulteriori ottimizzazioni per sulle cache, delle quali siamo completamente all'oscuro.