Loop Fusion

Al fine di eseguire il passo di ottimizzazione Loop Fusion **tra due loop** è necessario effettuare le seguenti verifiche:

1) Verificare che i due loop siano adiacenti:

   Per verificare l'adiacenza abbiamo controllato che l'exit block del primo loop coincidesse con il preheader del secondo loop. Tuttavia, questa non una condizione sufficiente alla dimostrazione dell'adiacenza, in quanto il Basic Block che i due loop condividono potrebbe contenere istruzioni esterne ai loop.

   Al fine di assicurarsi che tra i due loop non ci siano altre istruzioni è necessario verificare che la prima (e quindi l'unica) istruzione presente nel Basic Block condiviso tra i due loop sia un'istruzione branch (Br) che porti all'header del secondo loop.

```
bool LoopFusionPass::areLoopsAdjacent(BasicBlock * exitBlockL1, BasicBlock * preheaderL2, BasicBlock * headerL2)
{
    // Se l'exit Block del Loop1 coincide con il preheader del Loop2
    if (exitBlockL1 == preheaderL2)
    {
        // Se la PRIMA istruzione dell'exit Block del Loop1 è una branch, e se la branch porta al Loop2
        if (((*exitBlockL1).front()).getOpcode() == Instruction::Br
            && ((*exitBlockL1).front()).getOperand(0) == headerL2)
        {
            outs()<<"La prima istruzione dell'exit block del loop1 è una branch che porta al Loop2\n";
            return true;
        }
    }
    outs()<<"I due loop NON sono adiacenti\n";
    return false;
}
```

2) Verificare che i due loop abbiano lo stesso Trip Count:
   Al fine di assicurarsi che i due loop siano compatibili è necessario controllare che il numero di iterazioni dei due loop sia uguale. Per questo fine abbiamo utilizzato l'analisi ScalarEvolution, con la quale siamo in grado di calcolare e confrontare il numero di iterazioni

```
ScalarEvolution &SE = FAM.getResult<ScalarEvolutionAnalysis>(F);
```

```cpp
bool LoopFusionPass::sameTripCount(Loop * L1, Loop * L2, ScalarEvolution &SE)
{
    const SCEV * tripCountL1 = SE.getBackedgeTakenCount(L1);
    const SCEV * tripCountL2 = SE.getBackedgeTakenCount(L2);

    outs()<<"TRIP COUNT L1: "<<*tripCountL1<<"\n";
    outs()<<"TRIP COUNT L2: "<<*tripCountL2<<"\n";

    // Controlla che il trip count tra i due loop sia calcolabile e che sia uguale
    if ((!isa<SCEVCouldNotCompute>(tripCountL1)) && (!isa<SCEVCouldNotCompute>(tripCountL2)) && (tripCountL1 == tripCountL2))
        return true;

    return false;
}
```

Si precisa inoltre che per effettuare un passo di Loop Fusion, la sola analisi del Trip Count potrebbe non essere sufficiente, in quanto i due loop potrebbero condividere il numero di iterazioni ma potrebbero avere bound e/o step differenti.

*Si noti che in certi casi sarebbe possibile effettuare un'ottimizzazione con bound/step differenti:

```cpp
int i, a[100], b[200];
for (i = 0; i < 100; i++)
{
    a[i] = 1;
}
for (i = 100; i < 200; i++)
{
    b[i] = 2;
}
```

```cpp
int i, a[100], b[200];
for (i = 0; i < 100; i++)
{
    a[i] = 1;
    b[i+100] = 2;
}
```

Tuttavia, al fine di semplificare, si assume che i loop in input abbiano lo stesso bound e step.

3) Verificare che i due loop siano Control-Flow Equivalent:
Per assicurarsi che due loop siano Control-Flow Equivalent si utilizza l'analisi dei dominatori e dei post-dominatori.

```cpp
DominatorTree * DT = &FAM.getResult<DominatorTreeAnalysis>(F);
PostDominatorTree *PDT = &FAM.getResult<PostDominatorTreeAnalysis>(F);
```

Dominazione: un blocco x domina un blocco y se ogni percorso dall'Entry Block a y contiene x.

Post-Dominazione: un blocco y post-domina un blocco x se ogni percorso da x all'uscita contiene y.

Se il primo loop **domina** il secondo loop e se il secondo loop **post-domina** il primo loop, allora sono Control-Flow Equivalent.

```
bool LoopFusionPass::areLoopsControlFlowEquivalent(Loop * L1, Loop * L2, DominatorTree * DT, PostDominatorTree *PDT)
{
    // Se IterLoop1 domina IterLoop2 e se IterLoop2 post-domina IterLoop1 sono Control-Flow Equivalent
    if ((*DT).dominates((*L1).getLoopPreheader(), (*L2).getLoopPreheader())
        && (*PDT).dominates((*L2).getLoopPreheader(), (*L1).getLoopPreheader()))
        return true;

    return false;
}
```

L'effettiva fusione dei loop viene eseguita secondo i seguenti step:

1) Il successore del Body del Loop 1 diventa il Body del Loop 2

```
(*bodyTerminatorL1).setSuccessor(0, beginBodyL2);
```

2) Rimpiazza tutti gli usi della istruzione PHI dell'header del Loop 2 al fine di far utilizzare al secondo loop la stessa variabile di induzione del primo loop.

```
(*headerPhiL2).replaceAllUsesWith(headerPhiL1);
```

3) Il successore del Body del Loop 2 diventa il Latch del Loop 1

```
(*bodyTerminatorL2).setSuccessor(0, latchL1);
```

4) L'Exit Block del Loop 1 diventa l'Exit Block del Loop 2

```
(*headerTerminatorL1).setSuccessor(1, exitBlockL2);
```

5) Il successore dell'header del Loop 2 diventa il Latch del loop 2

```
(*headerTerminatorL2).setSuccessor(0, latchL2);
```

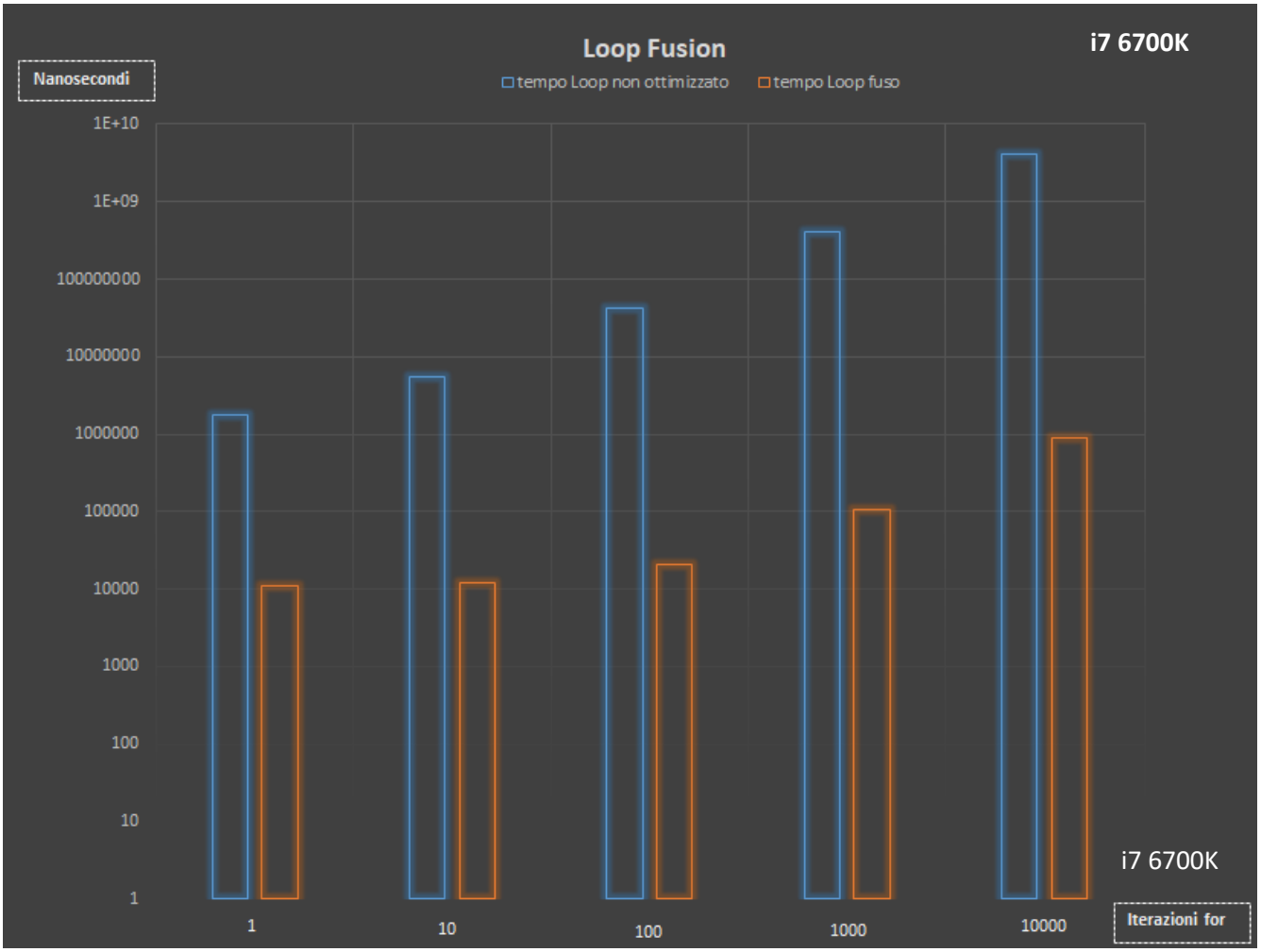
Misurazioni effettuate prima e dopo il passo di loop fusion

Le misurazioni sono state effettuate con dimensione degli array a, b e c in ingresso alla funzione populate di dimensione **100000**.

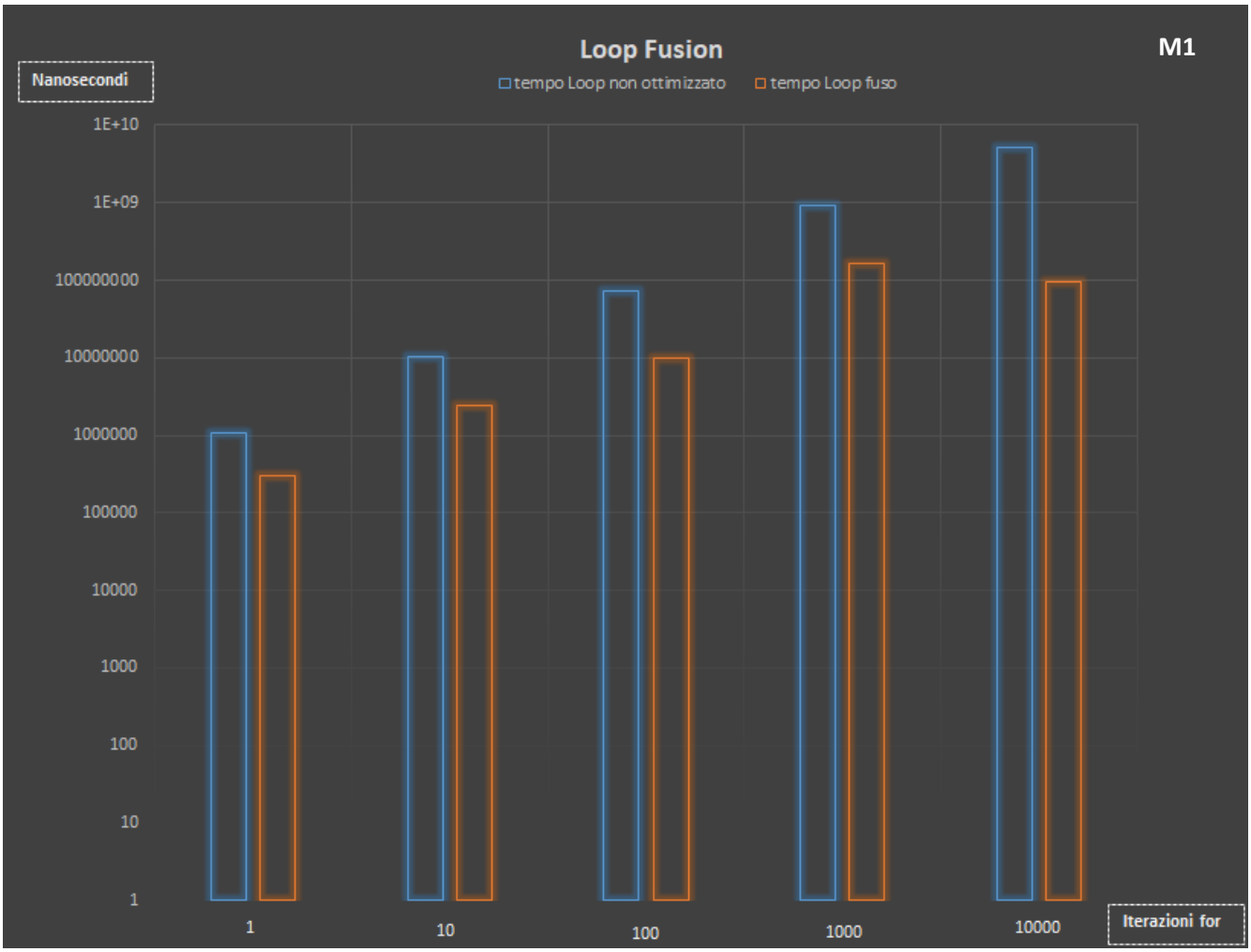
I dati sono stati visualizzati su scala logaritmica per permettere una migliore percezione della differenza.

Al fine di ottenere delle misurazioni più precise abbiamo realizzato uno script (script.sh) che calcola la media di **100** esecuzioni prima e dopo il passo di loop fusion.

Misurazione 1 – effettuata su i7 6700K (WSL):



Misurazione 2 – effettuata su Apple M1 (su macchina virtuale):



In entrambe le misurazioni si percepiscono miglioramenti significativi delle performance di almeno un'unità di grandezza indipendentemente dall'architettura su cui viene effettuata.