Comparison of the performance of scientific calculation codes

Lin Sinan

Supervisors: Prof. Sid Tuoati

Université Cote d'Azur
EIT Digital

June 25, 2020

UNIVERSITÉ CÔTE D'AZUR

Table of Contents

UNIVERSITÉ CÔTE D'AZUR

Table of Contents

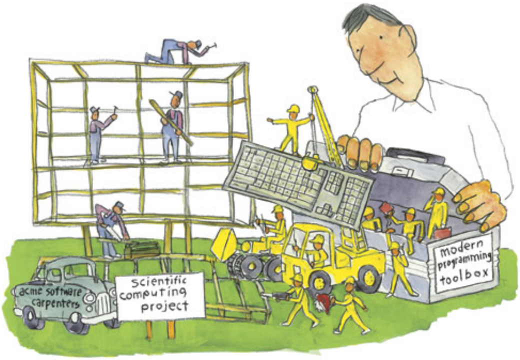

Figure: Where's the Real Bottleneck in Scientific Computing?
source: American Scientist

UNIVERSITÉ CÔTE D'AZUR

A great amount of execution time of a scientific calculation is spent on loops, such as, matrix multiplication. Modern compilers have been developed to make them faster by using loop optimization techniques:

Loop Optimization

- Loop Unrolling
- Loop Interchange
- Loop Blocking

Procedure

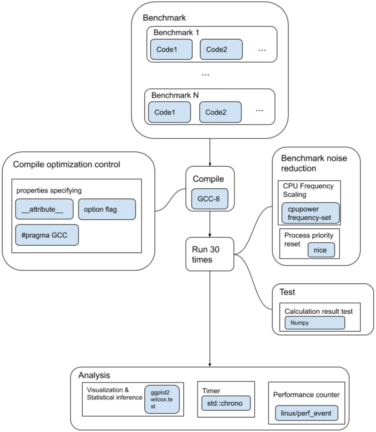

Figure: Benchmarking Procedure

UNIVERSITÉ CÔTE D'AZUR

Table of Contents

UNIVERSITÉ CÔTE D'AZUR

Loop Unrolling

Algorithm 1 Array Addition

$A \leftarrow$ an 32-bit float array with length 100
$b \leftarrow$ a 32-bit float scalar
i = 0
for $i < 100$ **do**
    $A[i] \leftarrow A[i] + b$
    $i \leftarrow i + 1$
EndFor

Algorithm 1 Array Addition with loop unrolling

$A \leftarrow$ an 32-bit float array with length 100
$b \leftarrow$ a 32-bit float scalar
i = 0
for $i < 100$ **do**
    $A[i] \leftarrow A[i] + b$
    $A[i+1] \leftarrow A[i+1] + b$
    $i \leftarrow i + 2$
EndFor

Loop unrolling replicates the body of the loop to reduce loop overhead.

Loop Unrolling

Algorithm 3 Assembly Instructions of Array Addition

```
procedure
    r1 ← address of A array
    r2 ← address of A plus offset 400 (we assume A is a 32-bit float array)
    r3 ← address of b
    r4 ← temporary registers
    loop:
        load r4, 0(r1)
        add r4, r4, r3
        save r4, 0(r1)
        add r1, r1, 4
        bne r1, r2, loop
```

Algorithm 4 Assembly Instructions with unrolling factor 1

```
procedure
    r1 ← address of A array
    r2 ← address of A plus offset 400 (we assume A is a 32bit int array)
    r3 ← address of b
    r4 ← temporary registers
    loop:
        load r4, 0(r1)
        add r4, r4, r3
        save r4, 0(r1)
        load r4, 4(r1)
        add r4, r4, r3
        save r4, 4(r1)
        add r1, r1, 8
        bne r1, r2, loop
```

Algorithm1 has 100 loops and 5 instructions in the loop body, so the number of instructions is 500. As for Algorithm2, the the number of instructions is 400, since it has 8 instructions in loop body, and only 50 loops.

Unrolling

```
// benchmarking starts
    int i,j,k;
    for (i = 0; i < N; i++){
        for (j = 0; j < M; j++){
#ifdef UNROLL_OPTION
#pragma GCC unroll 7
#endif
            for (k = 0; k< P; k++){
                C[i][j] = C[i][j] + A[i][k] * B[k][j];
            }
        }
    }
```

Figure: Common implementation

```
int i,j,k;
for (i = 0; i < N; i++){
    for (j = 0; j < M; j++){
        for (k = 0; k + 7 - 1 < P; k+=7) {
            C[i][j] = C[i][j] + A[i][k] * B[k][j];
            C[i][j] = C[i][j] + A[i][k+1] * B[k+1][j];
            C[i][j] = C[i][j] + A[i][k+2] * B[k+2][j];
            C[i][j] = C[i][j] + A[i][k+3] * B[k+3][j];
            C[i][j] = C[i][j] + A[i][k+4] * B[k+4][j];
            C[i][j] = C[i][j] + A[i][k+5] * B[k+5][j];
            C[i][j] = C[i][j] + A[i][k+6] * B[k+6][j];
        }
        // make sure the remaining loop are used.
        for (; k < P; k++){
            C[i][j] = C[i][j] + A[i][k] * B[k][j];
        }
    }
}
```

Figure: Manually unroll 6 times

Benchmarking code of orginal, manual unrolling and unroll with -funroll-loops

UNIVERSITÉ CÔTE D'AZUR

Result

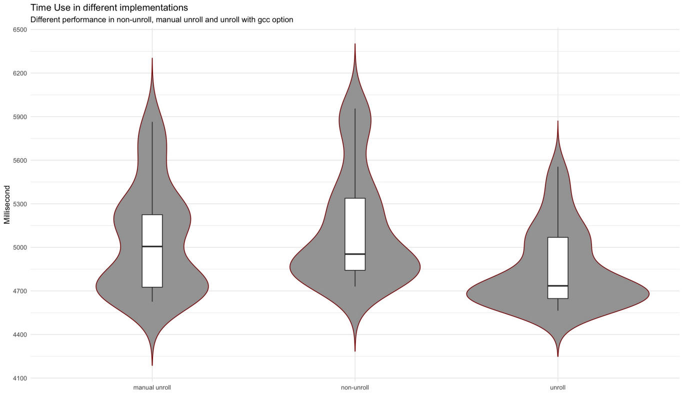

Figure: Violin plot of manually unroll, non-unroll and unroll with compiler option

Result

Table: Unrolling Benchmark Result and Statistical inference

	avg time (*ms*)	**std**	**# of instructions** (*million*)
unroll	4848.607	262.2337	12552.76
no unroll	5126.94	374.7673	23645.26
manually unroll	5049.016	365.0488	12071.26

UNIVERSITÉ CÔTE D'AZUR

Result

Paired samples Wilcoxon test result

Null Hypothesis	P value	Result
Unroll with compiler option is not faster than non-unroll	<0.0001	Refuse
Unroll with compiler option is not faster than manual-unroll	<0.0001	Refuse
Manual-unroll with compiler option is not faster than non-unroll	0.1998	Accept

UNIVERSITÉ CÔTE D'AZUR

Unrolling

```
 ::|||   ; CODE XREF from entry0 (0x100001afd)
.------> 0x100001a70      f30f1012        movss xmm2, dword [rdx]
:::|||   0x100001a74      4981c310a400.   add r11, 0xa410
:::|||   0x100001a7b      4883c21c        add rdx, 0x1c
:::|||   0x100001a7f      f3410f5993f0.   mulss xmm2, dword [r11 - 0xa410]
:::|||   0x100001a88      f30f105ae8      movss xmm3, dword [rdx - 0x18]
:::|||   0x100001a8d      f3410f599b60.   mulss xmm3, dword [r11 - 0x8ca0]
:::|||   0x100001a96      f30f1062ec      movss xmm4, dword [rdx - 0x14]
:::|||   0x100001a9b      f30f58c2        addss xmm0, xmm2            ; '|'
:::|||   0x100001a9f      f30f106af0      movss xmm5, dword [rdx - 0x10]
:::|||   0x100001aa4      f3410f59a3d0.   mulss xmm4, dword [r11 - 0x7530]
:::|||   0x100001aad      f3410f59ab40.   mulss xmm5, dword [r11 - 0x5dc0]
:::|||   0x100001ab6      f30f58c3        addss xmm0, xmm3            ; '}'
:::|||   0x100001aba      f30f1072f4      movss xmm6, dword [rdx - 0xc]
:::|||   0x100001abf      f3410f59b3b0.   mulss xmm6, dword [r11 - 0x4650]
:::|||   0x100001ac8      f30f107af8      movss xmm7, dword [rdx - 8]
:::|||   0x100001acd      f30f58c4        addss xmm0, xmm4            ; '~'
:::|||   0x100001ad1      f3440f1042fc    movss xmm8, dword [rdx - 4]
:::|||   0x100001ad7      f3410f59bb20.   mulss xmm7, dword [r11 - 0x2ee0]
:::|||   0x100001ae0      f3450f598390.   mulss xmm8, dword [r11 - 0x1770]
:::|||   0x100001ae9      4939cb          cmp r11, rcx
:::|||   0x100001aec      f30f58c5        addss xmm0, xmm5
:::|||   0x100001af0      f30f58c6        addss xmm0, xmm6
:::|||   0x100001af4      f30f58c7        addss xmm0, xmm7
:::|||   0x100001af8      f3410f58c0      addss xmm0, xmm8
`======< 0x100001afd      0f856dffffff    jne 0x100001a70
```

Figure: unroll with compiler option

```
 ::|||   ; CODE XREF from entry0 (0x100001ae3)
.------> 0x100001a60      f30f1002        movss xmm0, dword [rdx]
:::|||   0x100001a64      480510a40000    add rax, 0xa410
:::|||   0x100001a6a      4883c21c        add rdx, 0x1c
:::|||   0x100001a6e      f30f5980f05b.   mulss xmm0, dword [rax - 0xa410]
:::|||   0x100001a76      f30f58c1        addss xmm0, xmm1            ; '{'
:::|||   0x100001a7a      f30f104ae8      movss xmm1, dword [rdx - 0x18]
:::|||   0x100001a7f      f30f59886073.   mulss xmm1, dword [rax - 0x8ca0]
:::|||   0x100001a87      f30f58c1        addss xmm0, xmm1            ; '{'
:::|||   0x100001a8b      f30f104aec      movss xmm1, dword [rdx - 0x14]
:::|||   0x100001a90      f30f5988d08a.   mulss xmm1, dword [rax - 0x7530]
:::|||   0x100001a98      f30f58c1        addss xmm0, xmm1            ; '{'
:::|||   0x100001a9c      f30f104af0      movss xmm1, dword [rdx - 0x10]
:::|||   0x100001aa1      f30f598840a2.   mulss xmm1, dword [rax - 0x5dc0]
:::|||   0x100001aa9      f30f58c1        addss xmm0, xmm1            ; '{'
:::|||   0x100001aad      f30f104af4      movss xmm1, dword [rdx - 0xc]
:::|||   0x100001ab2      f30f5988b0b9.   mulss xmm1, dword [rax - 0x4650]
:::|||   0x100001aba      f30f58c1        addss xmm0, xmm1            ; '{'
:::|||   0x100001abe      f30f104af8      movss xmm1, dword [rdx - 8]
:::|||   0x100001ac3      f30f598820d1.   mulss xmm1, dword [rax - 0x2ee0]
:::|||   0x100001acb      f30f58c1        addss xmm0, xmm1            ; '{'
:::|||   0x100001acf      f30f104afc      movss xmm1, dword [rdx - 4]
:::|||   0x100001ad4      f30f598890e8.   mulss xmm1, dword [rax - 0x1770]
:::|||   0x100001adc      4839c1          cmp rcx, rax
:::|||   0x100001adf      f30f58c8        addss xmm1, xmm0            ; 'z'
`======< 0x100001ae3      0f8577ffffff    jne 0x100001a60
```

Figure: Manually unroll

Registers are not allocated well for manual version and increase stalls in instruction pipeline.

Cache

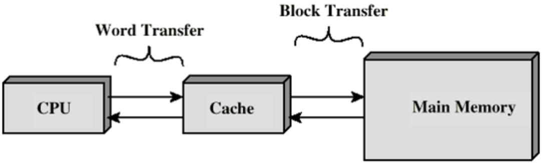

Figure: Data transfer among cpu, cache and RAM

When the computer read from or write to a location in main memory, it first checks whether a copy of that data is in the cache. If so, the processor immediately reads from or writes to the cache, which is much faster than reading from or writing to main memory.

Loop Interchange

Loop Interchange : change the order of loop to improve the cache locality.

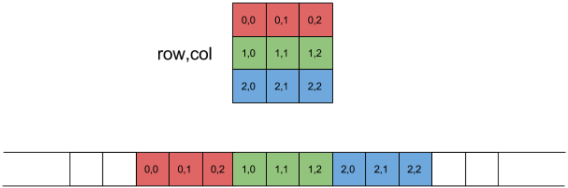

Figure: Layout of 2D array in memory(row-major)

Loop Interchange

Algorithm 7 Matrix multiplication in ijk order

```
int C[N][M];
int A[N][P];
int B[P][M];
for (int i = 0; i < n; i++)
    for (int j = 0; j < M; j++)
        for (int k = 0; k < P; k++)
            C[i][j] += A[i][k] * B[k][j];
```

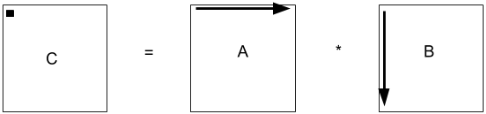

Figure: Matrix multiplication in ijk order

Loop Interchange

Algorithm 8 Matrix multiplication in ikj order

```
int C[N][M];
int A[N][P];
int B[P][M];
for (int i = 0; i < n; i++)
    for (int k = 0; k < P; k++)
        for (int j = 0; j < M; j++)
            C[i][j] += A[i][k] * B[k][j];
```

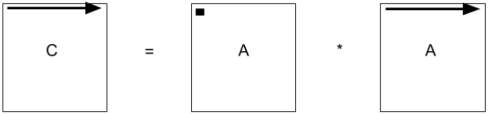

Figure: Matrix multiplication in ikjk order

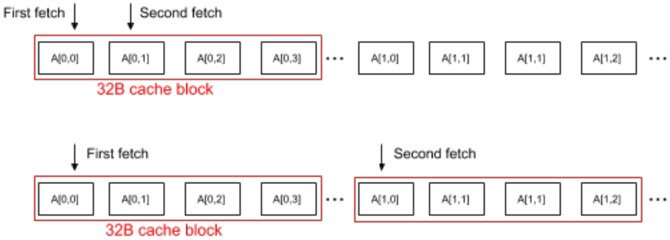

Figure: Different loop orders lead to different cache access pattern

UNIVERSITÉ CÔTE D'AZUR

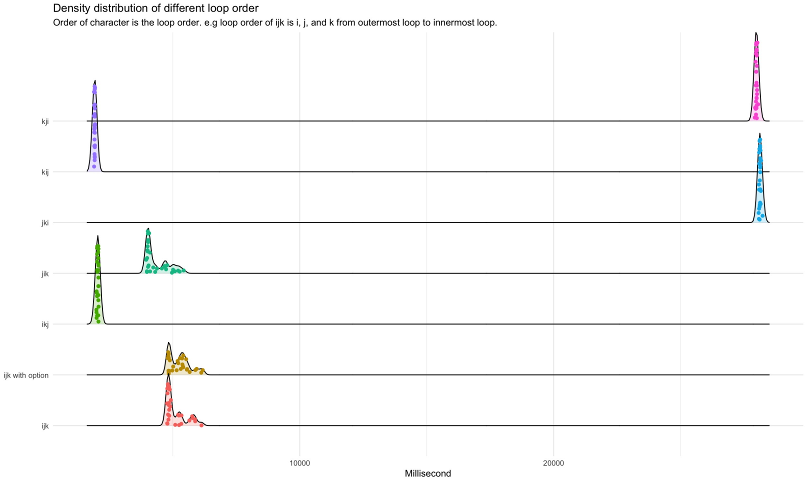

Figure: Ridge plot of different loop orders result

Table: Interchange Benchmark Result

	avg time (*ms*)	**std**	**LLC cache miss**[1] (*million*)
ijk with option	5250.45	411.0194	244.04
ijk	5118.148	374.77	229.61
jik	4390.417	469.84	170.75
ikj	2044.175	29.38	136.03
kij	1922.871	12.56	141.40
kji	27982.27	25.30	364.27
jki	28122.32	30.75	301.49

UNIVERSITÉ CÔTE D'AZUR

```
int i,j,k;
for (i = 0; i < N; i++){
    for (j = 0; j < M; j++){
        for (k = 0; k< P; k++){
            C[i][j] = C[i][j] + A[i][k] * B[k][j];
        }
    }
}
```

Figure: Common implementation

```
int i0,j0,k0,i,j,k;
for (i0 = 0; i0 < N; i0 += BLOCK_SIZE)
    for (k0 = 0; k0 < P; k0 += BLOCK_SIZE)
        for (j0 = 0; j0 < M; j0 += BLOCK_SIZE)
            for (i = i0; i < std::min(i0 + BLOCK_SIZE, N); i++)
                for (k = k0; k < std::min(k0 + BLOCK_SIZE, P); k++)
                    for (j = j0; j < std::min(j0 + BLOCK_SIZE, M); j++)
                        C[i][j] = C[i][j] + A[i][k] * B[k][j];
```

Figure: Matrix multiplication with Blocking

Loop Blocking

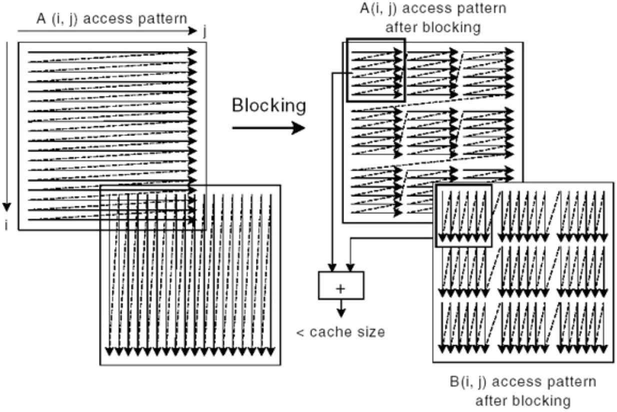

Figure: Loop Blocking visualization(from intel corp)

UNIVERSITÉ CÔTE D'AZUR

result

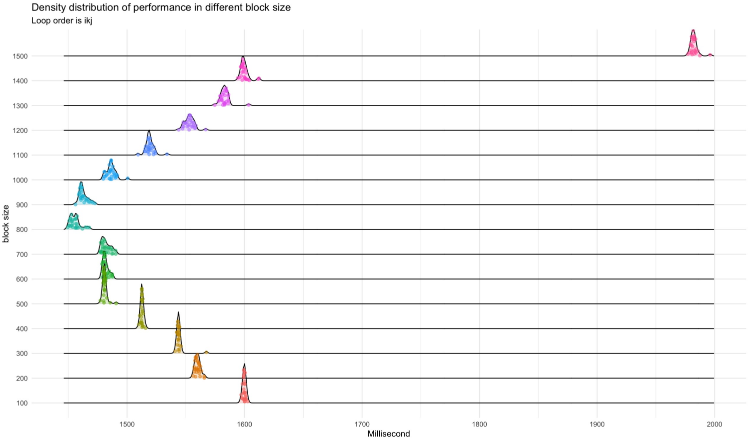

Figure: Ridge plot of performance in different block size

Block Size	avg time (ms)	LLC Cache miss (million)	Branch miss (million)
100	1599.766	4.01	34.43
300	1544.466	9.97	11.29
500	1481.069	4.05	6.76
700	1482.241	5.85	6.76
900	1463.007	12.66	4.51
1100	1519.416	44.01	4.51
1300	1583.119	90.78	4.51
1500	1982.382	135.07	2.25

Figure: Loop Blocking result)

Table of Contents

summary

SUMMARY

- Compilers can help to optimize your code well.
- Compilers cannot give an optimal solution sometimes.
- The property of cache is important for program performance.