Tips and tricks for optimization of Fortran codes

M. R. Hadizadeh*
Institute of Nuclear and Particle Physics, and Department of Physics, Ohio University, Athens, OH 45701, USA
(Dated: August 5, 2013)

In this short report, we have shown the simple optimization tips and tricks which can be used in general scientific programming with focus on Fortran.

I.

Compiler Options;

- Substantial gain can be easily obtained by playing with compiler options
- Optimization options are a must. The first and second level of optimization will rarely give no benefits!
- Optimization options can range from -O1 to -O5 with some compilers. -O3 to -O5 might lead to slower code, so try them independently on each subroutine.
- Always check your results when trying optimization options.
- Compiler options might include hardware specifics such as accessing vector units for example.

Intel Fortran and C compiler Options:
ifort, ifc and icc
    -O0 -O1 -O2 -O3 -ip -xW -tpp7(for P4) -ip ...

Vectorizing of DO loop; a DO loop can be vectorized when each array calculation is independent of another one

```
DO ix=1, Nx
    A(ix)=B(ix)×C(ix)+D(ix)
END DO
```

→

```
A=B×C+D
```

```
DO i=1, 1000
    DO j=1, 1000
        DO k=1, 500
            A(i,j,k) = X(i) × Y(j)
                       +Z(i,j,k)
                       +2.0/X(i)**2
        END DO
    END DO
END DO
```

→

```
DO j=1, 1000
    DO k=1, 500
        A(:,j,k) = X × Y(j)
                   +Z(:,j,k)
                   +2.0/X**2
    END DO
END DO
```

2000 Millions of operations **2 Millions of operations**

SUM function; summation by using SUM function instead of DO Loops

*Electronic address: hadizade@ift.unesp.br

```
Sumx=0
DO ix=1, Nx
    Sumx=Sumx+W(ix)× F(ix)
END DO
```

→

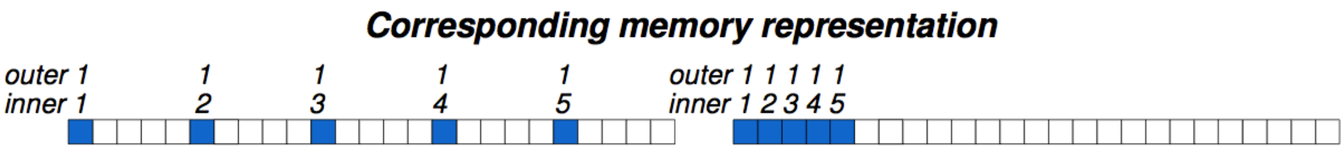
```
Sumx=SUM(W×F)
```

Array Considerations; try to minimize the memory jumps, they could be very costly because of cache and TLB misses

```
DO i=1, Ni
    DO j=1, Nj
        A(i,j) = ···
    END DO
END DO
```

→

```
DO j=1, Nj
    DO i=1, Ni
        A(i,j) = ···
    END DO
END DO
```



Minimizing the number of operations; one of the first thing for optimization is reducing the number of unnecessary operations performed by the CPU!

```
DO k=1, 10
    DO j=1, 5000
        DO i=1, 5000
            A(i,j,k) = 3.0 × m × D(k)+
                       C(j) × 23.5−
                       B(i)
        END DO
    END DO
END DO
```

→

```
DO k=1, 10
    Dtmp(k) = 3.0 × m×D(k)
    DO j=1, 5000
        Ctmp(j) = C(j) × 23.5
        DO i=1, 5000
            A(i,j,k) = Dtmp(k)+
                       Ctmp(j)−
                       B(i)
        END DO
    END DO
END DO
```

1250 Millions of operations **500 Millions of operations**

Complex Numbers; look for operations on complex numbers that have Imaginary or Real part equal to zero. This is again a question of minimizing the number of operations.

```
! Real part of A elements = 0
COMPLEX*16 A(1000,1000), B, C(1000,1000)

DO j=1, 1000
    DO i=1, 1000
        C(i,j) = A(i,j) × B
    END DO
END DO
```

→

```
REAL*8        AI(1000,1000)
COMPLEX*16 B, C(1000,1000)

DO j=1, 1000
    DO i=1, 1000
        C(i,j) = ( -IMAG(B) × AI(i,j),
                   AI(i,j) × REAL(B) )
    END DO
END DO
```

6 Millions of operations

2 Millions of operations

Loop Overhead and Objects declarations and instanciations; in Object-Oriented Languages AVOID objects declarations and instanciations within the most inner loops

```
DO j=1, 1000000
    DO i=1, 100000
        DO k=1, 2
            A(i,j,k) = B(i,j) + C(k)
        END DO
    END DO
END DO
```

→

```
DO j=1, 1000000
    DO i=1, 100000
        A(i,j,1) = B(i,j) + C(1)
        A(i,j,2) = B(i,j) + C(2)
    END DO
END DO
```

Function Call Overhead;

```
DO k=1, 10000
    DO j=1, 10000
        DO i=1, 5000
            A(i,j,k) = F1( C(i), B(j), k )
        END DO
    END DO
END DO

FUNCTION F1(x,y,m)
  REAL*8 x,y,tmp
  INTEGER m
  tmp=x*m - y
  RETURN tmp
END FUNCTION
```

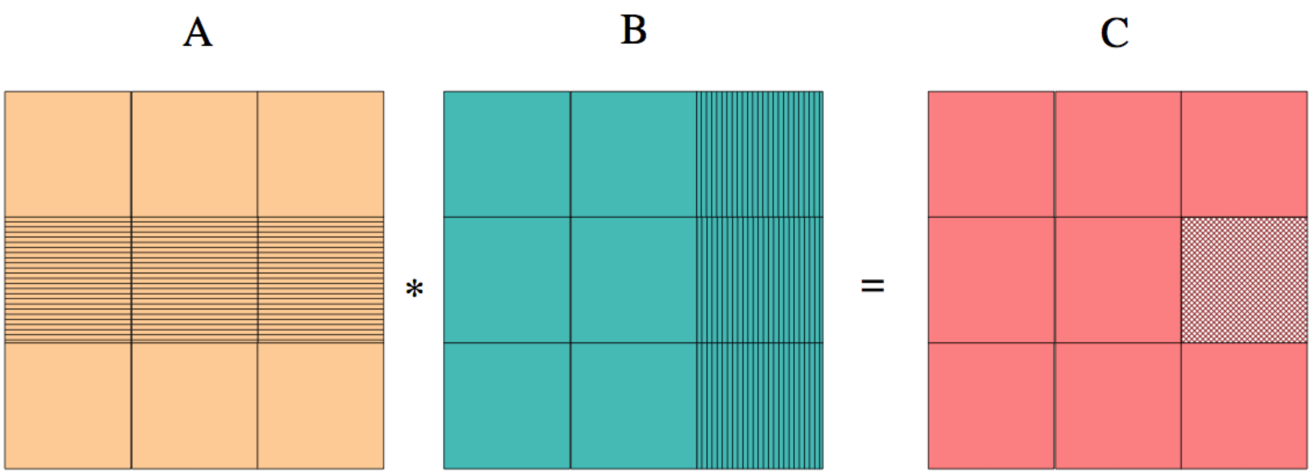
→

```
DO k=1, 10000
    DO j=1, 10000
        DO i=1, 5000
            A(i,j,k) = C(i) * k - B(j)
        END DO
    END DO
END DO
```

Blocking; Blocking is used to reduce cache and TLB misses in nested Matrix operations. The idea is to process as much as possible the data that is brought in the cache.

```
DO i=1, N
    DO j=1, N
        DO k=1, N
            C(i,j) = C(i,j)
                   +A(i,k)*B(k,j)
        END DO
    END DO
END DO
```

```
DO ib=1, N, bsize
    DO jb=1, N, bsize
        DO kb=1, N, bsize
            DO i=ib, min(N,ib+bsize-1)
                DO j=jb,min(N,jb+bsize-1)
                    DO k=kb,min(N,kb+bsize-1)
                        C(i,j) = C(i,j)
                               +A(i,k)*B(k,j)
                    END DO
                END DO
            END DO
        END DO
    END DO
END DO
```




```
C(i,j)=C(i,j)+A(i,k)*B(k,j)
```

Loop Fusion; The main advantage of Loop Fusion is the reduction of cache misses when the same array is used in both loops. It also reduces loop overhead and allow a better control of multiple instructions in a single cycle, when hardware allows it (2 FMA or 2 vector operations for example).

```
DO i=1, 10000
    A = A + X(i) + 2.0*Z(i)
END DO

DO i=1, 10000
    B = 3.0×X(i) - 5.0
END DO
```

```
DO i=1, 10000
    A = A + X(i) + 2.0*Z(i)
    B = 3.0×X(i) - 5.0
END DO
```

Loop Unrolling; the main advantage of Loop Unrolling is to reduce or eliminate data dependencies in loops. This is particularly useful when using an architecture with 2 FMA Units (IBM Power3-4) or a Vector unit (SSE2 extensions)

```
DO i=1, 1000
      A = A + X(i) × Y(i)
END DO
```

→

```
DO i=1, 1000, 4
      A = A + X(i) × Y(i)
            +X(i + 1) × Y(i + 1)
            +X(i + 2) × Y(i + 2)
            +X(i + 3) × Y(i + 3)
END DO
```

Sum Reductions; sum reductions is another way of reducing or eliminating data dependencies in loops. It is more explicit than the Loop Unrolling method.

```
DO i=1, 1000
      A = A + X(i) × Y(i)
END DO
```

→

```
DO i=1, 1000, 4
      A1 = A1 + X(i) × Y(i)
      A2 = A2 + X(i + 1) × Y(i + 1)
      A3 = A3 + X(i + 2) × Y(i + 2)
      A4 = A4 + X(i + 3) × Y(i + 3)
END DO
A = A1 + A2 + A3 + A4
```

Replace divisions by multiplications; Contrary to Floating Point multiplications or additions or subtractions, divisions are very costly in terms of clock cycles.

1 multiplication = 1 cycle
1 division = 14-20 cycles

```
DO j=1, 10000
      DO i=1, 10000
            A(i, j) = (B(i) − C(j))/D
      END DO
END DO
```

→

```
D = 1.0/D
DO j=1, 10000
      DO i=1, 10000
            A(i, j) = (B(i) − C(j)) × D
      END DO
END DO
```

Repeated multiplications for exponentials; exponentiation with a small exponent should be done manually. Like divisions exponential operations use many cycles.

```
A = B ∗ ∗3.0
```

→

```
tmpc = B ∗ B
A = tmpc ∗ B
```

Breaking Interpolations; the multi-dimensional interpolations should be considered as few one dimensional interpolations

Branching (proper use of IFs); try to minimize as much as possible the use of IFs within the inner loops. The CPU will first assume a YES when it encounters a IF statement while filling up the instruction pipline.