The Storage Hierarchy
CPU speed is much faster than the transfer speed between RAM and CPU; hence, if all data were only in RAM, the CPU would be idle most of the time.
Cache

Cache speed is nearly the same as CPU speed;
Data should be continuously prefetched/pipelined to cache which then supplies CPU as needed.
Hence, the data flow between RAM/Cache/CPU should be predicted.
The instructions for the data flow are usually supplied by a compiler.
But you need to help it too!

Cache
Can have levels. L1, L2, L3
Each subsequent level is faster (but smaller - expensive!) than the previous

RAM

Bottleneck
(Slow)

Fast

CPU
Strategies for efficiency

• **Registers**: do maximum work at data already at registers before requesting new data

• **Cache**: maximum use of data already in cache; understand how data are stored in cache

• **Data locality**: priority access the data bits that already close to each other in memory before the bits that are far

• **Input/Output**: generally avoid at all costs; if must, do as much as possible at once rather than little bit at a time
Avoid Data Dependency

\[
x = y + b
\]
\[
z = x + c
\]

Down

\[
x = y + b \quad \text{z depends on } x
\]
\[
z = y + b + c \quad \text{no dependency}
\]

Usually automatically done by a compiler when optimization is on
Avoid explicit integer power

\[ x = y^{**3.0} \quad \text{expensive power-function may be called} \]

\[ \downarrow \]

\[ x = y*y*y \]

Can be automatically done by a ‘smart’ compiler
Avoid divisions

\[ x = y / 2. \]

division can be much more expensive than multiplication

\[ x = 0.5 \times y \]

Can be automatically done by a ‘smart’ compiler
Reduce computation of common subexpressions

\[ x = y \times \frac{b}{c} \]
\[ z = f \times \left(\frac{b}{c}\right)^{2.5} \]

\[ t = \frac{b}{c} \]
\[ x = y \times t \]
\[ z = f \times t^{2.5} \]

Can be automatically done by a ‘smart’ compiler
Watch out for Loop Invariants

do i=1,n
    x(i) = y(i) + z(i) * a/c(i)/b
    h(n) = f * a
end do

a/b does not depend on i

h(n) does not depend on i

tmp = a/b

do i=1,n
    x(i) = y(i) + z(i) * tmp/c(i)
end do

h(n) = f * a
Look for conditions that dependent on loop index

Before

```plaintext
do i=1,nx
  do j=1,ny
    if(x(j).gt.0) then
      y(i,j) = x(i) + z(i,j)*c(i)
    else
      y(i,j) = z(i,j)*b(i)
    end if
  end do
end do
```

After

```plaintext
do j=1,ny
  if(x(j).gt.0) then
    do i=1,nx
      y(i,j) = x(i) + z(i,j)*c(i)
    end do
  else
    do i=1,nx
      y(i,j) = z(i,j)*b(i)
    end do
  end if
end do
```
Get boundary conditions outside the loop

\[
\begin{align*}
do & \ i=1,n \\
& \quad \text{if } (i==1 \ \text{or} \ i==n) \ \text{then} \\
& \quad \quad x(i) = y(i) \\
& \quad \text{else} \\
& \quad \quad x(i) = y(i)*d(i) \\
& \quad \text{end if} \\
& \end{align*}
\]

Before

\[
\begin{align*}
x(1) & = y(1) \\
do & \ i=1,nx \\
& \quad x(i) = y(i)*d(i) \\
& \end{do} \\
x(n) & = y(n) \\
\end{align*}
\]

After
Index Splitting

do i=1,n
    if (i<m) then
        x(i) = y(i)*c(i)
    else
        x(i) = y(i)*d(i)
    end if
end do

Before

After

do i=1,m-1
    x(i) = y(i)*c(i)
end do
do i=m,n
    x(i) = y(i)*d(i)
end do
Loop order exchange

Make ‘slower’ changing loop-index to be in outer loop, fast index - inner loop

In Fortran, array a(3,3) is stored as: a(1,1) a(2,1) a(3,1) a(1,2) a(2,2) a(3,2) a(1,3) a(2,3) a(3,3)

In C the order is reversed: a(1,1) a(1,2) a(1,3) a(2,1) a(2,2) a(2,3) a(3,1) a(3,2) a(3,3)
Loop fusion
(register reuse)

Only if all arrays in the loop fit in cache (small)

Before

\[
\begin{align*}
&\text{do } i=1,n \\
&\quad x(i) = y(i) + 1
\end{align*}
\]

\[
\begin{align*}
&\text{do } i=1,n \\
&\quad a(i) = x(i) + c
\end{align*}
\]

\[
\begin{align*}
&\text{do } i=1,n \\
&\quad d(i) = a(i) + y(i)
\end{align*}
\]

end do

After

\[
\begin{align*}
&\text{do } i=1,n \\
&\quad x(i) = y(i) + 1 \\
&\quad a(i) = x(i) + c \\
&\quad d(i) = a(i) + y(i)
\end{align*}
\]

end do
Loop fission
(effective cache use)

Only if all arrays are large, so all don’t fit in cache

Before

\[
\text{do } i=1,n \\
\quad x(i) = y(i)+1 \\
\quad a(i) = x(i)+c \\
\quad d(i) = a(i)+y(i) \\
\text{end do}
\]

After

\[
\text{do } i=1,n \\
\quad x(i) = y(i)+1 \\
\text{end do} \\
\text{do } i=1,n \\
\quad a(i) = x(i)+c \\
\text{end do} \\
\text{do } i=1,n \\
\quad d(i) = a(i)+y(i) \\
\text{end do}
\]
Avoid loop index dependencies

do i=1,n
  x(i) = x(i-1)*a(i)
  b(i) = x(i)*c(i)
end do

Before

doi=0,n-1
  y(i) = x(i)*a(i+1)
end do
doi=1,n
  x(i) = y(i)*c(i)
end do

After
Avoid if’s when can

```plaintext
if (x >= 1.) then
  x = 1.
elseif (x <= 0.) then
  x = 0.
end if
```

Before

```plaintext
x=min(1.,max(0.,x))
```

After
avoid if’s when can

if (u(i) >= 0.) then
  x(i) = u(i)*(x(i)-x(i-1))
else
  x(i) = u(i)*(x(i+1)-x(i))
end if

v=max(0.,u(i)) * (x(i)-x(i-1))+ &
min(0.,u(i)) * (x(i+1)-x(i))

Before

After
Never call a subroutine/function within large loop

```
real function fun (x)
  x = x*cos(x)
end function fun

Before

do j=1,ny
  do i=1,nx
    ...
    y= fun(x(i,j))
  end do
end do
```

After

```
real function fun (x)
  x = x*cos(x)
end function fun

Inlining

do j=1,ny
  do i=1,nx
    ...
    y= x(i,j)*cos(x(i,j))
  end do
end do
```
Steps in code development:
Never call a subroutine/function within large loop

Before

```
do j=1,ny
  do i=1,nx
    ... 
    call micro()
    ...
  end do
end do
```

After

```
subroutine micro()
  ...
  do j=1,ny
    do i=1,nx
      ...
    end do
  end do
end subroutine micro
```
Some advise

• new code should go through initial test compiling with optimization off, uninitialized memory and array bound checks on;
• If the program fails after ported to another system or after not using the code for awhile, try running first with optimization off;
• Always try to use maximum compiler optimization; but make sure the results are similar (not the same) with lower optimization
• Use comments, sensible names for variables, indenting;
• One file per subroutine/module;
• Never use ‘magic numbers’.