Coloquio Programación en FORTRAN

Día 4 (clases 7 y 8).

- Pointers.
- Tipos de datos derivados.
- Aspectos de la arquitectura
- Trabajo con bibliotecas.
- Operaciones de punto flotante.

Arreglos de dimensión automática

```
subroutine swap(a,b)
  real, dimension(:) :: a,b
  real, dimension(size(a)):: work
                  !  la dimension de a y b se determina
                  !  automaticamente en la llamada a la subrutina
  work=a
  a=b
  b=work
end subroutine swap
```

Son arreglos dinámicos, se almacenan en el stack.
Ver el comando ulimit -s
Los arreglos definidos con allocate se almacenan en el heap.

Stack vs heap

pila vs montículo

Stack

- Acceso muy rápido.
- Asignación y liberación automática de memoria.
- No fragmentación de memoria.
- Capacidad limitada (depende del SO, en linux se controla con ulimit , pero puede no estar no configurado por el administrador)

Heap

- Acceso menos rápido.
- Requiere manejo explícito: allocate y deallocate.
- No garantizado el uso eficiente.
- Fragmentación de memoria.
- Memoria mayor, limitada sólo por la RAM.

Pointers

Un pointer en Fortran contiene la ubicación en la memoria de un objeto, pero también information del tipe, rango y extensión.

Declaración, ejemplos:

```
integer, pointer :: p                          ! pointer to integer
real, pointer, dimension (:) :: rp             ! pointer to 1-dim real array
real, pointer, dimension (:,:) :: rp2          ! pointer to 2-dim real array
```

Asignación de Pointers

```
program pointer1

implicit none
integer, pointer :: p1
allocate (p1)
p1 = 1
end program pointer
```

implícitamente, existe una vartiable objetivo (target) es la que toma el valor “1”.

Asignación de Pointers

```
program pointer1

implicit none
integer, pointer :: p1
allocate (p1)
p1 = 1
end program pointer
```

implícitamente, existe una vartiable objetivo (target) es la que toma el valor “1”.

```
program pointer2
implicit none
integer, pointer :: p1
integer, target :: t1
p1 => t1
p1 = 1
end program pointer2
```

En esta forma, el target está explícito (t1).

Asociación de pointers

- associated (ptr, trgt) : devuelve .true. Si ptr está asociado al puntero trgt.

  associated(p1) o associated(p1,t1) dan .true. despúes de la sentencia p1=>t1.

- nullify (ptr) . Disocia el pointer ptr.
- Nullify() no afecta al target, puede haber mas de un pointer asociado el mismo target

  Un pointer se puede asociar y disociar más de una vez en el programa, a distintas variables. Permite flexibilidad y facilita integración de varios programas.

Tipos de datos derivados

Son combinaciones de los tipos de dato intrínsecos: integer, real, complex, logical, and character.

Ejemplo:

```
type :: atom
character (len=2) :: label
real :: x, y, z
end type atom
type(atom) :: carbon1        ! declaración de variable de tipo atom


……...
carbon1%label = “C”
carbon1%x = 0.0000        ! noten que no se permite espacio en blanco
carbon1%y = 1.3567        ! antes o después de %
carbon1%z = 2.5000
….
```

Arreglos de tipos de datos derivados

Ejemplo:

```
type :: atom
character (len=2) :: label
real :: x, y, z
end type atom
type(atom),dimension(10) :: molecule   !declara de arreglo de tipo atom

……..
molecule(1)%label = “C”
molecule(1)%x = 0.0000        ! noten que no se permite espacio en blanco
molecule(1)%y = 1.3567        ! antes o después de %
molecule(1)%z = 2.5000
….
```

El CPU y la memoria

Problema:

La velocidad de lectura/escritura de la memoria RAM es inferior a la velocidad de los CPU.

Solución paliativa: Memorias caché y Non Uniform Memory Access (NUMA).

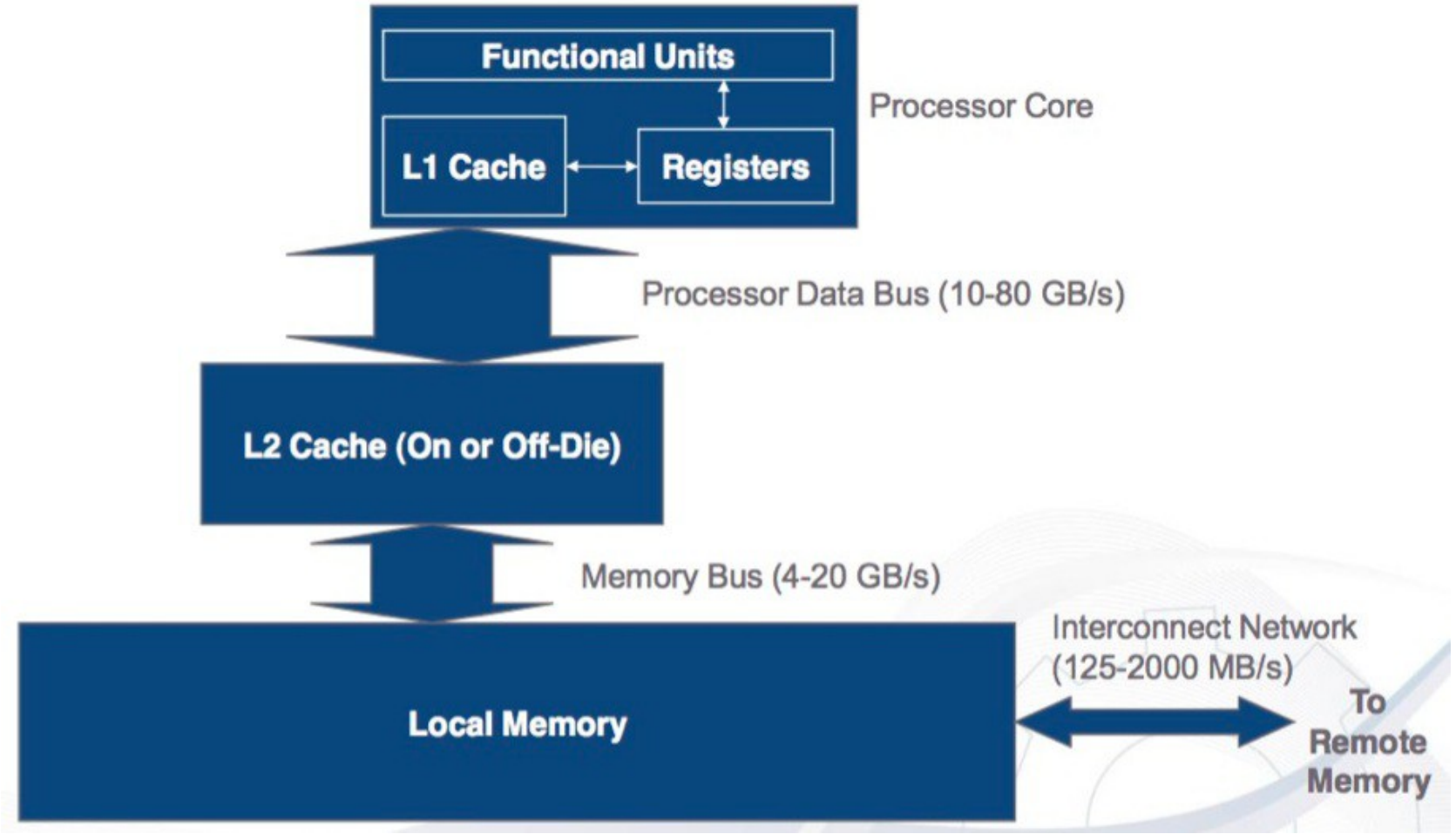
Memoria caché

Es un búfer especial de memoria que poseen las computadoras, funciona de manera similar a la memoria principal, pero es de menor tamaño y de acceso más rápido. Es usada por el microprocesador para reducir el tiempo de acceso a datos ubicados en la memoria principal que se utilizan con más frecuencia.

Se sitúa entre la unidad central de procesamiento (CPU) y la memoria de acceso aleatorio (RAM) para acelerar el intercambio de datos.

procede de la voz inglesa cache; “escondite secreto” para guardar mercancías, habitualmente de contrabando.

Jerarquía de memoria

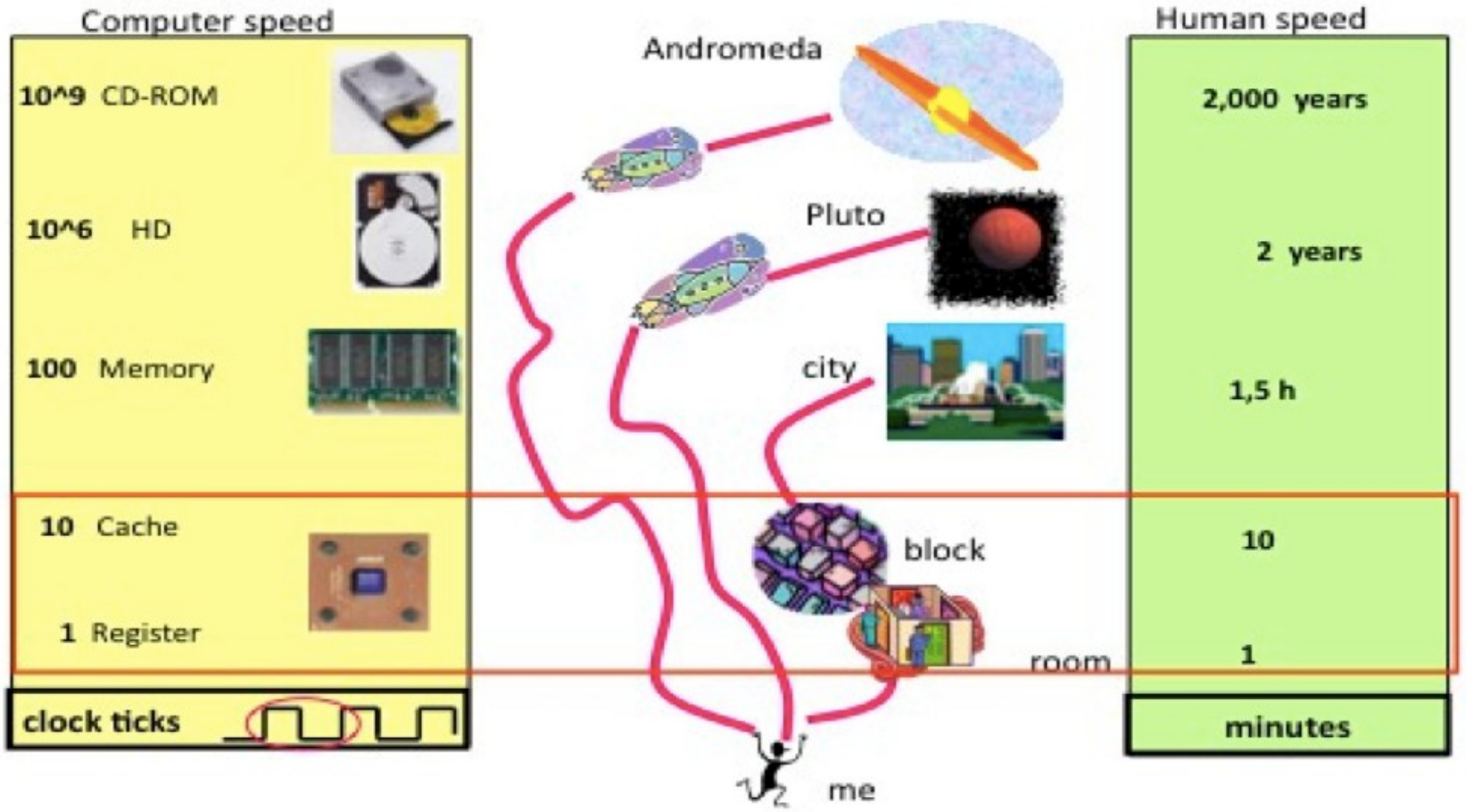


Fuente: S. Cozzini, LSCMS 2009.

Componentes de la memoria

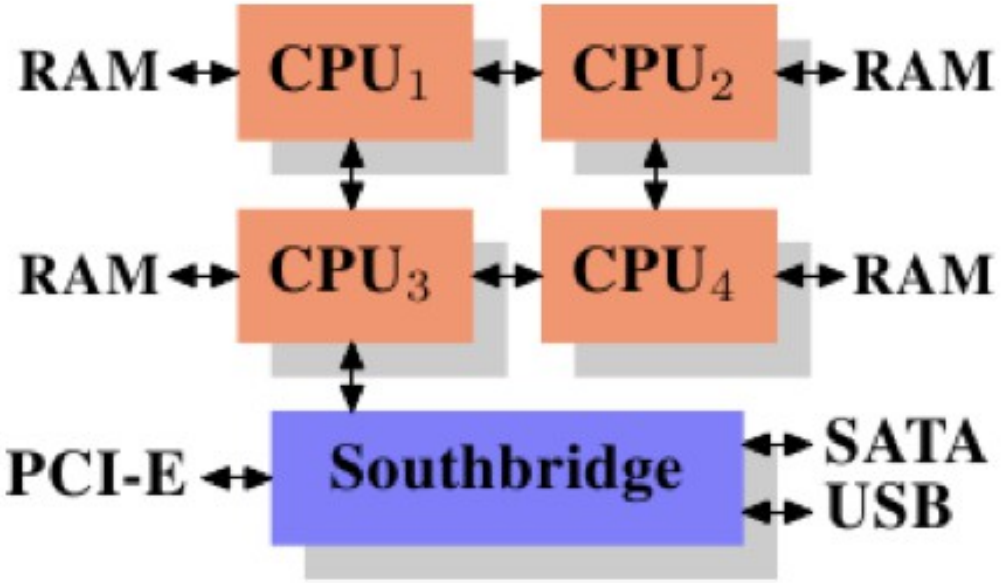
- Registros: Circuitos on-chip. Guardan operandos y resultados de operaciones binarias del CPU.
- Cache L1 (Primaria). Memoria pequeña on-chip.
- L2 y L3 (Secundarias) Cache: Memoria mayor, (on-or off-chip) usada para guardar datos e instrucciones obtenidos de la RAM.
- Memoria Local: RAM y HDD en el mismo nodo.
- Memoria remota: Memoria en otro nodo de un cluster. .

Analogía de tiempos de acceso



Fuente: S. Cozzini, LSCMS 2009.

Non Uniform Memory Access (NUMA)



Consecuencias

- Programar para una máquina con distribución jerárquica de la memoria requiere una optimización para esta estructura de memoria.

Localidad

La mayoría de los programas tienen un alto grado de localidad en sus accesos. Esto explota la jerarquía de memoria.

- Localidad temporal:
  Los últimos productos referenciados (instr o datos) probablemente serán usados de nuevo en un futuro próximo: bucles iterativos, subrutinas, variables locales.

- Localidad espacial:
  Acceso a los datos que está cerca el uno al otro: arreglos

- Localidad secuencial:

El procesador ejecuta las instrucciones en el orden del programa: - La relación ramificaciones/secuencia de relación es típicamente de 1 a 5.

¿Cómo optimizar?

1. Compruebe si hay respuestas correctas (el programa debe ser correcto!)

2. Profiling, determine donde están los cuellos de botella, por ejemplo, la mayoría de las rutinas que requieren mucho tiempo.

3. Optimizar el uso de estas rutinas de opciones del compilador, compilador directivas (pragmas), y modificaciones de código fuente.

Repita 1-3.

Técnicas de optimización

Mejorar el rendimiento de la memoria (el más importante)

- Mejor patrón de acceso a memoria
- El uso óptimo de líneas de caché (mejorar la localidad espacial)
- Re-uso de los datos almacenados en caché (mejorar la localidad temporal)

nivel)

Mejorar el Rendimiento del CPU

- Crar oportunidades de escalamiento (alto nivel), mejorar el plan del programa (de bajo nivel)

Utilice bibliotecas / subrutinas optimizadas .

Optimización para el manejo de memoria

- Intercambio de bucles (loop interchange)
- Desarrollo de bucles.
- División de punto flotante
- Fusión de bucles
- Reuso de los datos de caché.

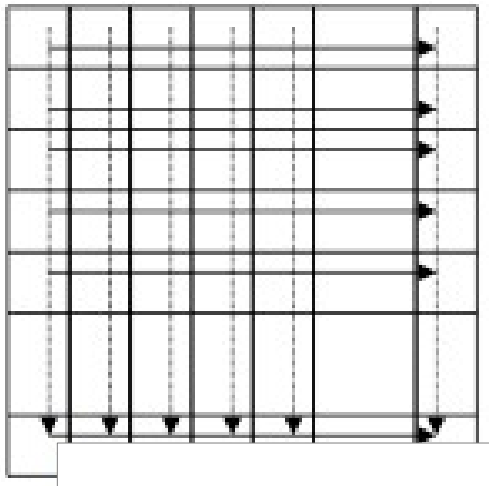
Intercambio de bucles

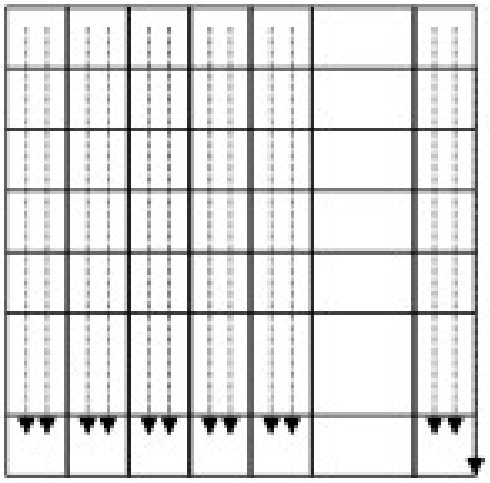
Original

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

Bucle intercambiado

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





Orden de acceso
Orden de almacenamiento

Ejemplo 1: Intercambio de bucles

```
DO i=1,300
  DO j=1,300
    DO k=1,300
      A (i,j,k) = A (i,j,k)+ B (i,j,k)*C (i,j,k)
    END DO
  END DO
END DO
```

```
cd  /home/emenendez/CursoFortran/Programas/Optimizacion
gfortran -o arreglos.x -O3 arreglos.f90
echo 500 | ./arreglos.x      (echo 500 le da el dato 500 al programa)
```

Orden	Gfortan -O3	Gfortran -O0
i j k	8.74	11.64
k j i	0.54	1,48
k i j	2.84	3.29

Intercambio de bucles por compilador

```
MacBook-Air-de-Eduardo:Optimizacion emenendez$ gfortran -o
arreglos.x  -floop-interchange  arreglos.f90

f951: sorry, unimplemented: Graphite loop optimizations cannot be
used (-fgraphite, -fgraphite-identity, -floop-block, -floop-
interchange, -floop-strip-mine, -floop-parallelize-all, and
-ftree-loop-linear)
```

Aunque no emita mensajes, compruebe que el compilador realmente hace lo que promete.

Prefetching

Prefetching es la obtención de datos desde la RAM hacia la cache antes de que sean requeridos por el próximo cálculo.

GNU:

`-fprefetch-loop-arrays`

```
If supported by the target machine, generate instructions to prefetch
memory to improve the performance of loops that access large arrays.
```

Compruebe que el compilador realmente lo hace

Desenrollado de bucles (loop unrolling)

- Bucle normal

```
do i=1,N
   a(i)=b(i)+x*c(i)
end do
```

- Bucle desarrollado

```
do i=1,N,4
  a(i)=b(i)+x*c(i)
  a(i+1)=b(i+1)+x*c(i+1)
  a(i+2)=b(i+2)+x*c(i+2)
  a(i+3)=b(i+3)+x*c(i+3)
end do
```

También se logra con opciones de compilación

gfortran -funroll-loops

Compruebe que el compilador realmente lo hace

División de punto flotante

- Es una operación costosa (20-60 ciclos CPU).
- Estandar IEEE: no debe ser reemplazada por multiplicación por el recíproco.
- Los compiladores ofrecen relajar el estándar IEEE.

  `gfortran -funsafe-math-optimizations`

- Puede que sea aplicada automaticamente en las opciones `-O3` o `-Ofast .`
-

Ejemplo

Optimizacion/inversearray.f90

```
call cpu_time(start)
 do j=1,N
     do i=1,N
     A(i,j)=A(i,j)/B(i)
     enddo
 Enddo
call cpu_time(finish)
```

```
! multiplicacion por inverso
 do i=1,ndim
  C(i)=1/B(i)
enddo
 call cpu_time(start)
 do j=1,N
     do i=1,N
     A(i,j)=A(i,j)*C(i)
     enddo
 enddo
call cpu_time(finish)
```

Peligro: Ejemplo de division → multiplicación x recíproco

```
program inverse
!a simple program to check how many inverse are not accurate.
real*4 :: X,Y,Z
integer:: I
i=0
do j=1,100,1
    X=real(j)
    Y=1.0/X
    Z=Y*X
    IF (Z.ne.1.0) then
        write(*,'(A,F4.1,F8.4,F28.24)')  &
              "this is not correct=", X,Y,Z
        i=i+1
    end if
 end do
 write(*,*)"found", I
end program inverse
```

Ejercicios : a) Cambiar real*4 por real*8 y1.0 por 1.d0 , b) gfortran -Ox with x=1,2,3 , c) gfortran -ffloat-store (estandar IEEE estricto) . Los resultados dependen del compilador.

Blocking

Optimización consistente en regular el procesamiento de datos para que entren por bloques a la cache.

Trapsosicion de una matriz.

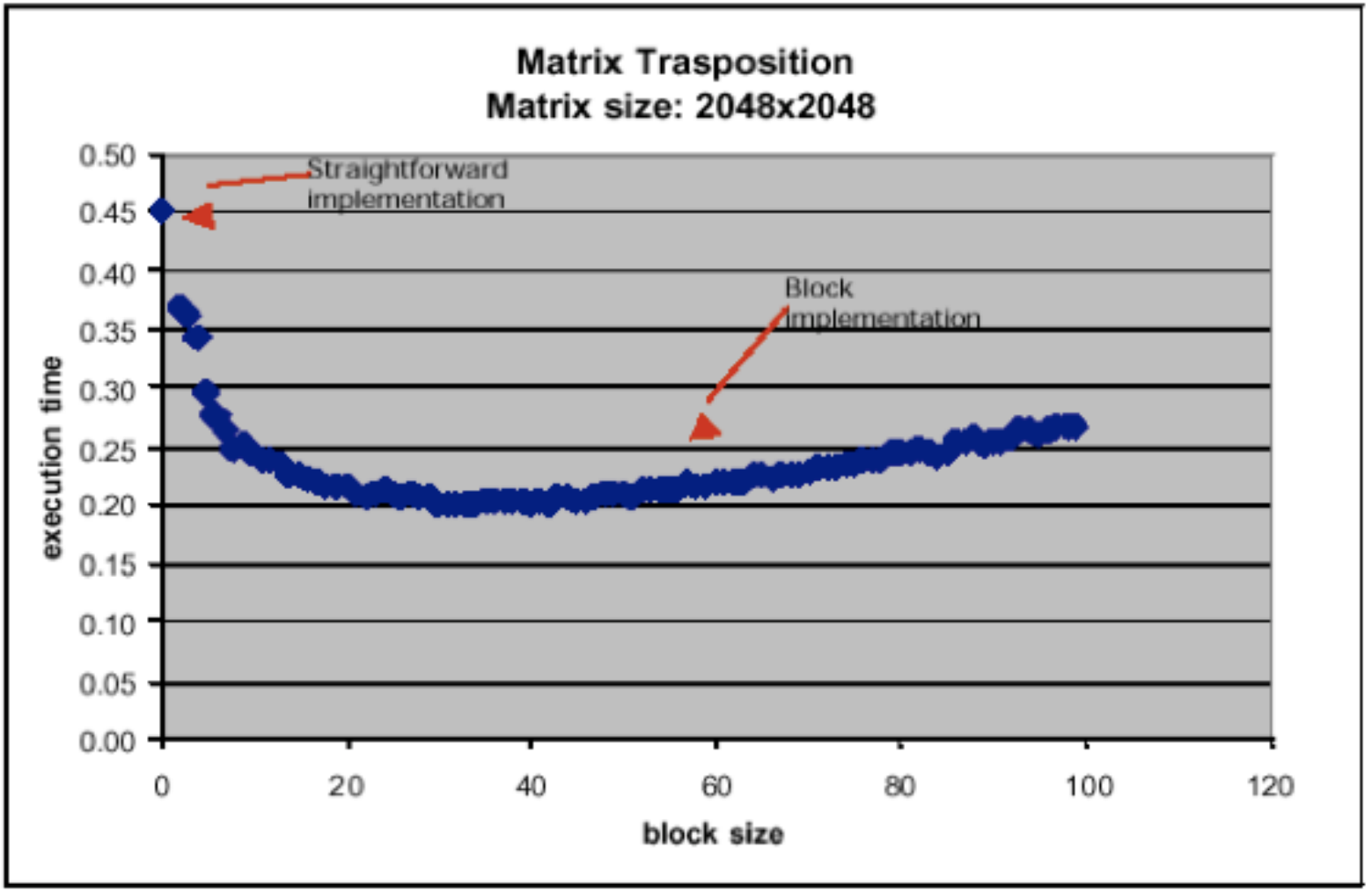
```
do i=1,n
  do j=1,n
     a(i,j)=b(j,i)
  end do
end do
```

Block algorithm for transposing a matrix:

- block data size= bsize
  - mb=n/bsize
  - nb=n/bsize
- Code is a little bit more complicated if
  - MOD(n,bize) is not zero
  - MOD(m,bize) is not zero

```
do ib = 1, nb
 ioff = (ib-1) * bsiz
 do jb = 1, mb
   joff = (jb-1) * bsiz
    do j = 1, bsiz
      do i = 1, bsiz
        buf(i,j) = x(i+ioff, j+joff)
     enddo
    enddo
    do j = 1, bsiz
      do i = 1, j-1
       bswp = buf(i,j)
       buf(i,j) = buf(j,i)
       buf(j,i) = bswp
      enddo
     enddo
    do i=1,bsiz
      do j=1,bsiz
       y(j+joff, i+ioff) = buf(j,i)
      enddo
    enddo
  enddo
enddo
```

Results... (Carlo Cavazzoni data)



Consejo: Use bibliotecas como BLAS, ATLAS, y optimizadas por fabricantes MKL, ACML, ESSL.

Ejemplo: multiplicación de matrices

```
do i=1,ndim
 do j=1,ndim
  temp=0.d0
  do k=1,ndim
   temp=temp+A(i,k)*B(k,j)
  end do
  C(i,j)=temp
 end do
end do
```

A(i,k) no se accede en orden contiguo, fuerza continuo recambio de la cache.

Multiplicación de matrices optimizada

Loop interchange

```
! Adaptado de la rutina BLAS DGEMM
  zero=0 d0
  DO j = 1,ndim !ciclo en j
   DO i = 1,ndim ! ciclo en i
     c(i,j) = zero
   END DO          !fin de ciclo en i
   DO l = 1,ndim  ! ciclo en l
!     IF (B(l,j).NE.zero) THEN
       temp = b(l,j)
       DO i = 1,ndim
         c(i,j) = c(i,j) + temp*a(i,l)
       END DO
 !     END IF
   END DO  ! Fin ciclo en l
  END DO ! fin de ciclo en j
```

Fuera del ciclo interno, evita recambio frecuente de cache

Sumario

- Los compiladores optimizan bien a nivel de instrucción y a veces hacen transformaciones a los bucles.
- Programar en el orden natural para las optimizaciones.
- Las técnicas son generales, pero los detalles dependen de la arquitectura.
- Blocking es usado en muchos programas. Depende fuertemente de la arquitectura.

Operaciones de punto flotante

Ver Charla de Stefano Cozzini