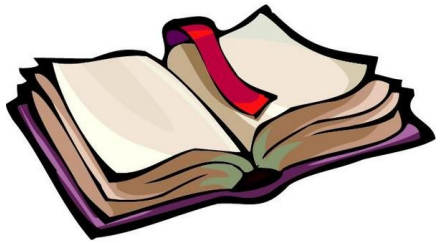
Sistema de Memoria

Universidad Nacional de La Plata

Facultad de Informática

Prof. Dr. Adrian Pousa

Agenda

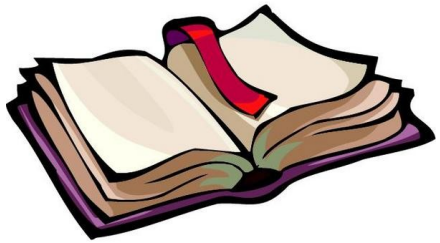


I. Sistema de memoria
   i. Limitaciones del sistema de memoria
   ii. La importancia de la memoria caché
      i. Principio de localidad
      ii. Conceptos asociados a la memoria cache (Estados, Niveles, Contención, Coherencia)

II. Principio de localidad aplicado a la multiplicación de matrices

III. Minimizando el espacio de memoria. Caso de estudio matrices triangulares

Agenda



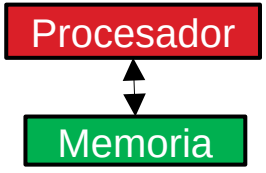
I. Sistema de memoria
   i. Limitaciones del sistema de memoria
   ii. La importancia de la memoria caché
      i. Principio de localidad
      ii. Conceptos asociados a la memoria cache (Estados, Niveles, Contención, Coherencia)

II. Principio de localidad aplicado a la multiplicación de matrices

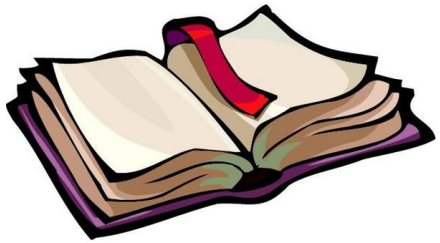
III. Minimizando el espacio de memoria. Caso de estudio matrices triangulares

Limitaciones del Sistema de memoria

- En muchas aplicaciones la limitación no está en la potencia de cómputo del procesador sino en el sistema de memoria.
- Dos parámetros esenciales del sistema de memoria son:
  - **La latencia:** tiempo desde el "*memory request*" hasta que los datos están disponibles.
  - **El ancho de banda:** velocidad con la cual el sistema de memoria puede depositar los datos en el procesador.
- Los procesadores han evolucionado mas rápido que la memoria (*Memory Wall*). Por esta razón, un sistema donde el procesador y la memoria están conectados directamente se ve afectado por lo siguiente:
  - El procesador puede realizar varias instrucciones por ciclo pero el acceso a memoria para traer operandos puede demorar mas de un ciclo.
  - El rendimiento del procesador se ve afectado dado que queda ocioso.



Agenda

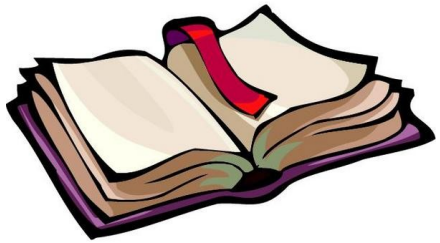


I. Sistema de memoria
   i. Limitaciones del sistema de memoria
   ii. La importancia de la memoria caché
      i. Principio de localidad
      ii. Conceptos asociados a la memoria cache (Estados, Niveles, Contención, Coherencia)

II. Principio de localidad aplicado a la multiplicación de matrices

III. Minimizando el espacio de memoria. Caso de estudio matrices triangulares

Agenda



I. Sistema de memoria
   i. Limitaciones del sistema de memoria
   ii. La importancia de la memoria caché
      i. Principio de localidad
      ii. Conceptos asociados a la memoria cache (Estados, Niveles, Contención, Coherencia)

II. Principio de localidad aplicado a la multiplicación de matrices

III. Minimizando el espacio de memoria. Caso de estudio matrices triangulares

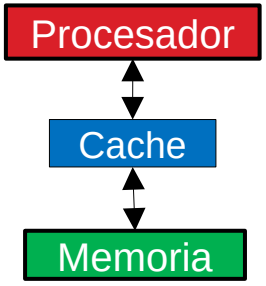
Memoria cache
Principio de localidad

- **Principio de localidad:** los accesos a memoria de un determinado programa no están distribuidos por todo el espacio de direcciones, sino que temporalmente se concentran en posiciones contiguas.

- El principio de localidad se manifiesta en dos aspectos:
  - **Localidad temporal:** referido a la naturaleza repetitiva de los programas. Un dato (o instrucción) accedido recientemente es muy probable que se acceda en un futuro muy próximo.
  - **Localidad espacial:** un dato que se encuentra en una posición cercana a un dato recientemente usado es muy probable que se acceda muy pronto.

- Este principio, demostrado en la práctica, se aprovecha en la jerarquía de memoria para mejorar el rendimiento. Específicamente, permite alcanzar mejoras en el tiempo de ejecución cuando se utilizan mecanismos de **memoria caché** (Por ejemplo: traer a **cache de datos** más de un dato).

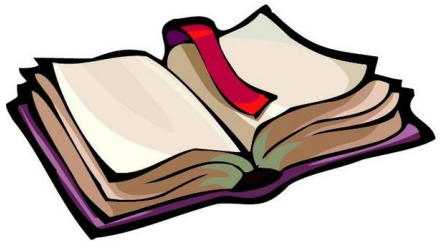
Memoria Cache



- Agregar una memoria cache reduce los tiempos de respuesta.
  - **Ventaja:** posee menor latencia que la memoria RAM.
  - **Desventaja:** es mas costosa y posee menor capacidad.
- Ante un requerimiento de memoria, en primer lugar se buscará el dato en la memoria cache, pudiendo ocurrir dos situaciones:
  - **Cache miss (fallo):** el dato no se encuentra en cache y debe traerse desde memoria RAM.
  - **Cache hit (acierto):** el dato se encuentra en cache entonces el acceso posee menor latencia y como consecuencia se mejora el tiempo de respuesta.
- El objetivo es establecer estrategias para minimizar la cantidad de accesos a memoria RAM maximizando la cantidad de accesos a memoria cache.

La memoria cache surge como consecuencia del principio de localidad!!!

Agenda



I. Sistema de memoria
   i. Limitaciones del sistema de memoria
   ii. La importancia de la memoria caché
      i. Principio de localidad
      ii. Conceptos asociados a la memoria cache (Estados, Niveles, Contención, Coherencia)

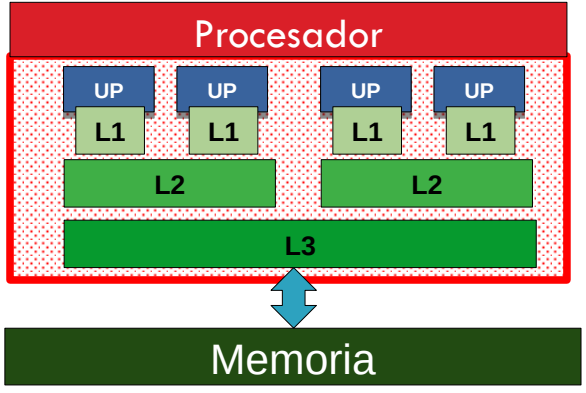
II. Principio de localidad aplicado a la multiplicación de matrices

III. Minimizando el espacio de memoria. Caso de estudio matrices triangulares

Memoria Cache
Estados y Niveles

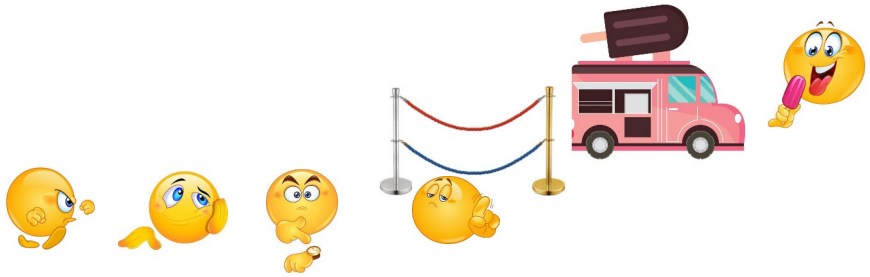
- La memoria cache puede encontrarse en dos estados:
  - **Cold cache:** cuando la cache esta vacía o con datos irrelevantes.
  - **Hot cache:** cuando la cache tiene datos relevantes y todas las lecturas del programa son satisfechas por la cache.

- En procesadores modernos, con varias unidades de procesamiento (cores), es habitual encontrar más de un nivel de cache (L1, L2, L3) conformando lo que se conoce como **jerarquía de memoria** (RAM + Caches).
  - A mayor nivel, mayor latencia y mayor tamaño.
  - En este caso se agrega un nuevo estado:
    - **Warm cache:**
      - L1 hot
      - L3 cold
      - L2 en un estado intermedio



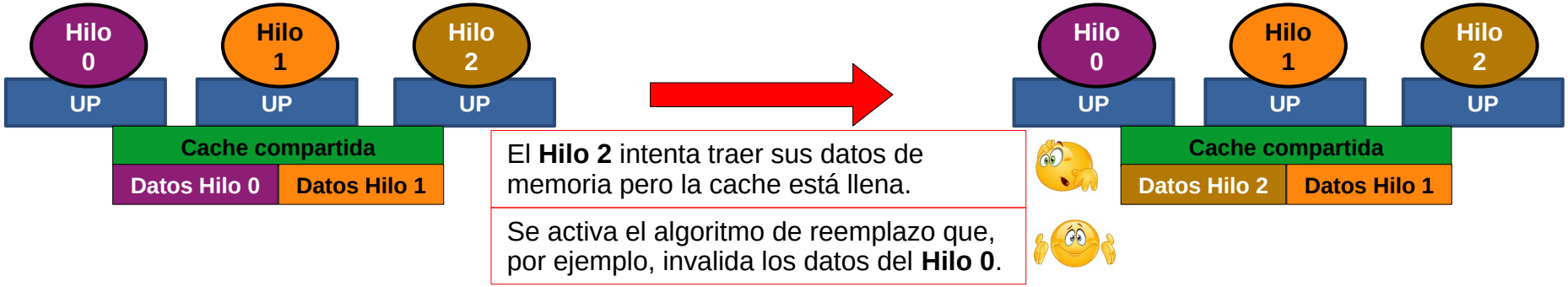
Memoria Cache
Contención

- Se llama contención de recursos (del inglés resource contention) a un conflicto en el acceso a un recurso compartido como RAM, unidad de disco, memoria caché, buses internos o dispositivos de red.
- La contención de recursos se produce cuando varios procesos/hilos intentan usar el mismo recurso compartido. Por ejemplo: dos procesos/hilos de un programa intentan leer celdas del mismo bloque de memoria a la vez.
- Una de las funciones básicas de los sistemas operativos es resolver problemas de contención de recursos. Para esto, utilizan mecanismos de bajo nivel.
- En los últimos años, la investigación sobre contención se ha centrado más en la jerarquía de memoria: cachés de último nivel (LLC), FSB (front side bus), socket de memoria.



Memoria Cache
Contención

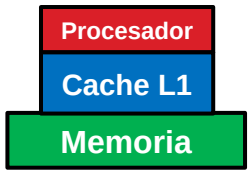
- Un problema de contención puede presentarse al usar **cachés compartidas**:
  - Una aplicación paralela **intensiva en memoria** donde varios hilos comparten una memoria caché. Cada hilo podría invalidar los datos de otro hilo generando fallos de caché continuos.



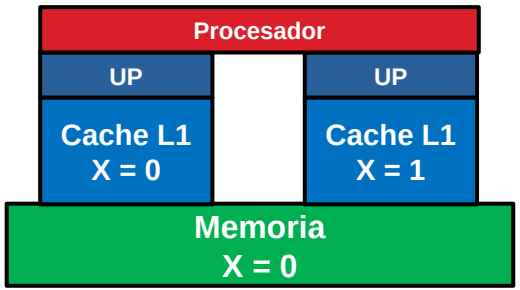
- Algunas veces, el programador debe resolver los problemas de contención:
  - Crear menos hilos.
  - Distribuir los hilos de manera de aprovechar mejor la localidad.
  - Utilizar software para particionado de memoria cache.
  - Rediseñar el algoritmo para obtener patrones de acceso libres de contención.

Memoria Cache
Coherencia – Problema de coherencia

- Una lectura de memoria debe retornar el último valor escrito en esa dirección, sin importar el proceso/hilo que escribió dicho valor:
  - En un sistema con un procesador y una unidad de procesamiento, no hay interferencias debido a que múltiples procesos/hilos se ejecutan de a uno a la vez en el único procesador y ven la memoria a través de la misma jerarquía.
  - En un sistema con un procesador y varias unidades de procesamiento, los múltiples procesos/hilos deberían ver el mismo estado de la memoria independientemente de si estos se ejecutan o no en diferentes unidades de procesamiento.



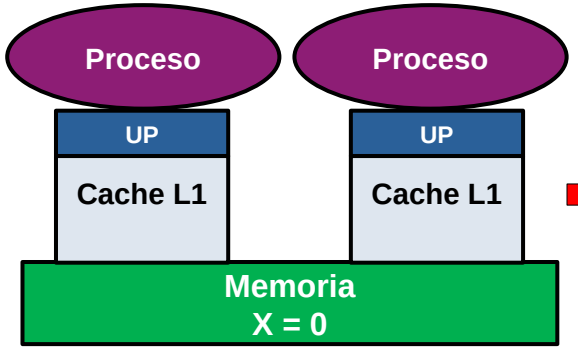
Problema de coherencia: varios procesos/hilos ven la memoria compartida a través de diferentes cachés, uno de ellos ve un valor actualizado en su caché mientras que otros todavía ven el antiguo.



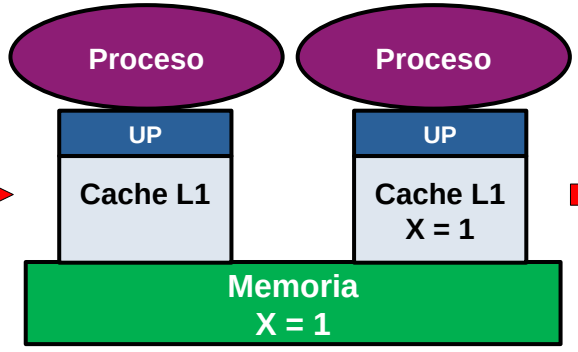
Memoria Cache
Coherencia - Ejemplo

- El problema de coherencia es un problema de **hardware** independiente de la sincronización del programa.
- **Ejemplo:** varios procesos actualizan un variable atómicamente.

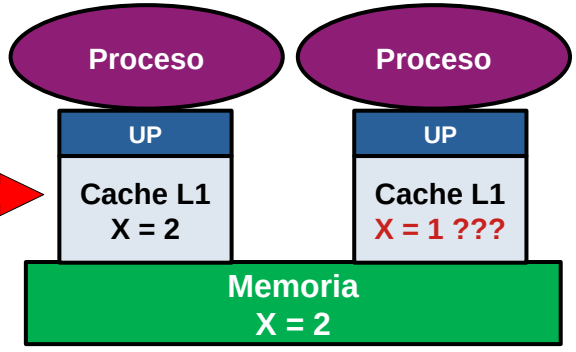
```
Var x=0;
Process [id:0 to P]{
        …
        atomic(x=x+1)
        …
}
```



Inicialmente una variable reside en memoria inicializada en 0.



Un proceso lee e incrementa la variable. Se actualiza el valor en memoria y se almacena en la cache privada del proceso.



Otro proceso lee e incrementa la variable. Se actualiza el valor en memoria y se almacena en la cache privada del proceso, pero queda desactualizado en las caches de otros procesos.

Memoria Cache
Coherencia - Soluciones

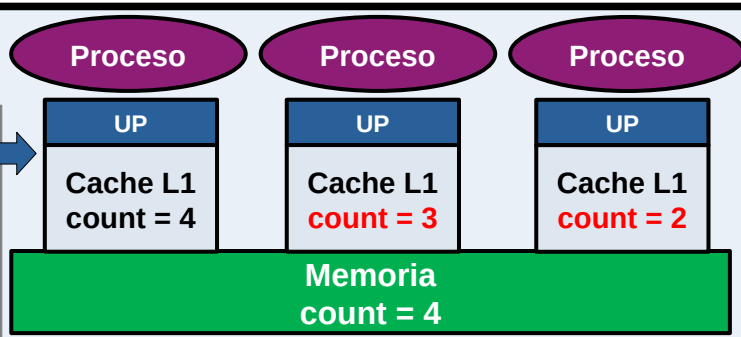
- Los problemas de coherencia se resuelven mediante protocolos hardware.
- Existen dos tipos de protocolos de coherencia:
  - **Sistemas basados en bus (snoopy):** continuamente se observa si hay modificaciones en las líneas de cache.
  - **Sistemas basados en diccionario:** un diccionario almacena la validez de las líneas de cache.
- Cuando se manifiesta un problema de coherencia, estos sistemas siguen alguna de las siguientes políticas:
  - **Basadas en invalidación (MSI, MESI, MOSI, MOESI):** un cambio en el valor de una variable almacenada en cache de una unidad de procesamiento, genera la **invalidación** de las líneas de cache de toda unidad de procesamiento que tengan una copia de esa variable.
  - **Basadas en actualización (Dragon):** un cambio en el valor de una variable almacenada en cache de una unidad de procesamiento, genera la **actualización** de las líneas de cache de toda unidad de procesamiento que tengan una copia de esa variable.

Memoria Cache
Contención + Coherencia

- Suponer el siguiente problema (Práctica 1 de Concurrencia y Paralelismo) resuelto con P procesos sobre P unidades de procesamiento: Dado un número N verifique cuantas veces aparece ese número en un arreglo v de longitud M (M proporcional a P)

```
Var count=0;
Var v[M];

Process [id:0 to P-1]{
 Parte=M/P
 Ini = id*Parte
 Fin = Ini + Parte - 1
   for i = Ini to Fin
     if v[i]==N
      atomic(count=count+1)
   End for
}
```



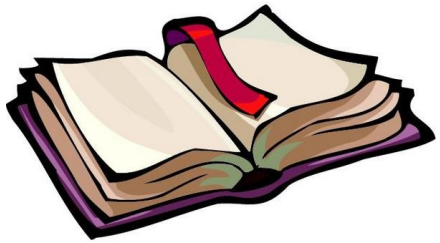
En el peor caso, cada posición del arreglo tiene el valor N.
Se generan M accesos de escritura a la misma posición de memoria.
Los accesos se secuencializan (contención alta)
El protocolo de coherencia se activa una y otra vez.

```
Var count=0;
Var v[M];

Process [id:0 to P-1]{
 Local_count = 0;
 Parte=M/P
 Ini = id*Parte
 Fin = Ini + Parte - 1
   for i = Ini to Fin
     if v[i]==N
      Local_count=Local_count+1
   End for
  atomic(count=count+Local_count)
}
```

Se soluciona llevando la cuenta en una variable local a cada proceso y minimizando accesos a la variable compartida (contención baja).

Agenda



I. Sistema de memoria
   i. Limitaciones del sistema de memoria
   ii. La importancia de la memoria caché
      i. Principio de localidad
      ii. Conceptos asociados a la memoria cache (Estados, Niveles, Contención, Coherencia)

II. Principio de localidad aplicado a la multiplicación de matrices

III. Minimizando el espacio de memoria. Caso de estudio matrices triangulares

Principio de localidad
Aplicación a la multiplicación de matrices

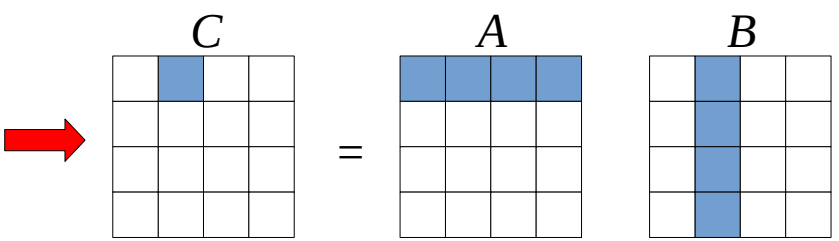
- Supongamos una memoria cache con las siguientes características:
  - Cada línea dos palabras.
  - El valor de cualquier variable ocupa una palabra.
  - Traer una palabra trae la palabra siguiente en memoria.
  - Campos de control:
    - **Variable:** Variable en cache.
    - **Fallo:** Si la línea de cache provocó un fallo.

Memoria Cache			
Variable	Palabra 1	Palabra 2	Fallo
Fallos de cache totales			

- Analizaremos el comportamiento de esta memoria cache cuando resolvemos el problema de la multiplicación de dos matrices cuadradas de NxN:

$$C = AB$$

$$c_{i,j} = \sum_{k=0}^{N-1} a_{i,k} \cdot b_{k,j}$$



Para nuestro programa ejemplo, almacenamos en memoria A, B y C como arreglos.

Principio de localidad
Aplicación a la multiplicación de matrices

- Supongamos $C=AB$ con A, B y C matrices de NxN con $N=2$.

$$c_{i,j}=\sum_{k=0}^{N-1} a_{i,k}\cdot b_{k,j} \Rightarrow \begin{pmatrix} c_{0,0} & c_{0,1} \\ c_{1,0} & c_{1,1} \end{pmatrix} = \begin{pmatrix} a_{0,0} & a_{0,1} \\ a_{1,0} & a_{1,1} \end{pmatrix} \cdot \begin{pmatrix} b_{0,0} & b_{0,1} \\ b_{1,0} & b_{1,1} \end{pmatrix}$$

Total de instrucciones

$$\begin{aligned} c_{0,0} &= a_{0,0}\cdot b_{0,0} + a_{0,1}\cdot b_{1,0} \\ c_{0,1} &= a_{0,0}\cdot b_{0,1} + a_{0,1}\cdot b_{1,1} \\ c_{1,0} &= a_{1,0}\cdot b_{0,0} + a_{1,1}\cdot b_{1,0} \\ c_{1,1} &= a_{1,0}\cdot b_{0,1} + a_{1,1}\cdot b_{1,1} \end{aligned}$$

- Supongamos A y C ordenadas en memoria por filas consecutivas:

$c_{0,0}$	$c_{0,1}$	$c_{1,0}$	$c_{1,1}$	$a_{0,0}$	$a_{0,1}$	$a_{1,0}$	$a_{1,1}$
0	1	2	3	0	1	2	3

Memoria

En cuanto al orden en memoria de la matriz B, suponemos dos escenarios:

B por filas

```
For i = 0 to N-1
 For j = 0 to N-1
  For k = 0 to N-1
   // C(i,j)+=A(i,k)+B(k,j)
   C[i*N+j] += A[i*N+k] * B[k*N+j]
```

$b_{0,0}$	$b_{0,1}$	$b_{1,0}$	$b_{1,1}$
0	1	2	3

Memoria

B por columnas

```
For i = 0 to N-1
 For j = 0 to N-1
  For k = 0 to N-1
   // C(i,j)+=A(i,k)+B(k,j)
   C[i*N+j] += A[i*N+k] * B[k+j*N]
```

$b_{0,0}$	$b_{1,0}$	$b_{0,1}$	$b_{1,1}$
0	1	2	3

Memoria

Acceder al elemento (x,y) de una matriz Mf ordenada en memoria por filas: $Mf_{(x,y)} = Mf[x.N+y]$

Acceder al elemento (x,y) de una matriz Mc ordenada en memoria por columnas: $Mc_{(x,y)} = Mc[x + y.N]$

Principio de localidad
Aplicación a la multiplicación de matrices

$c_{0,0}$	$c_{0,1}$	$c_{1,0}$	$c_{1,1}$	$a_{0,0}$	$a_{0,1}$	$a_{1,0}$	$a_{1,1}$
0	1	2	3	0	1	2	3

Memoria

B por filas

```
For i = 0 to N-1
 For j = 0 to N-1
  For k = 0 to N-1
   // C(i,j)+=A(i,k)+B(k,j)
    C[i*N+j] += A[i*N+k] * B[k*N+j]
```

$b_{0,0}$	$b_{0,1}$	$b_{1,0}$	$b_{1,1}$
0	1	2	3

Memoria

B por columnas

```
For i = 0 to N-1
 For j = 0 to N-1
  For k = 0 to N-1
   // C(i,j)+=A(i,k)+B(k,j)
    C[i*N+j] += A[i*N+k] * B[k+j*N]
```

$b_{0,0}$	$b_{1,0}$	$b_{0,1}$	$b_{1,1}$
0	1	2	3

Memoria

Veamos que pasa con el cálculo de $C_{0,0}$

Total de instrucciones

$$c_{0,0}=a_{0,0}\cdot b_{0,0}+a_{0,1}\cdot b_{1,0}$$
$$c_{0,1}=a_{0,0}\cdot b_{0,1}+a_{0,1}\cdot b_{1,1}$$
$$c_{1,0}=a_{1,0}\cdot b_{0,0}+a_{1,1}\cdot b_{1,0}$$
$$c_{1,1}=a_{1,0}\cdot b_{0,1}+a_{1,1}\cdot b_{1,1}$$

Memoria cache inicialmente vacía

Memoria Cache			
Variable	Palabra 1	Palabra 2	Fallo
Fallos de cache totales			0

En ambos casos, los primeros fallos se producen al cargar los indices i, j, k

Memoria Cache			
Variable	Palabra 1	Palabra 2	Fallo
i	*0*	…	*X*
j	*0*	…	*X*
k	*0*	…	*X*
Fallos de cache totales			3

Principio de localidad
Aplicación a la multiplicación de matrices

$c_{0,0}$	$c_{0,1}$	$c_{1,0}$	$c_{1,1}$	$a_{0,0}$	$a_{0,1}$	$a_{1,0}$	$a_{1,1}$
0	1	2	3	0	1	2	3

Memoria

B por filas

```
For i = 0 to N-1
 For j = 0 to N-1
  For k = 0 to N-1
   // C(i,j)+=A(i,k)+B(k,j)
    C[i*N+j] += A[i*N+k] * B[k*N+j]
```

$b_{0,0}$	$b_{0,1}$	$b_{1,0}$	$b_{1,1}$
0	1	2	3

Memoria

`C[0] += A[0] * B[0]`

Variable	Palabra 1	Palabra 2	Fallo
Lineas i, j k … ya no fallan			
C	$c_{0,0}$	$c_{0,1}$	X
A	$a_{0,0}$	$a_{0,1}$	X
B	$b_{0,0}$	$b_{0,1}$	X
Fallos de cache totales			6

`C[0] += A[1] * B[2]`

Variable	Palabra 1	Palabra 2	Fallo
Lineas i, j k … ya no fallan			
Lineas C y A … ya no fallan			
B	$b_{1,0}$	$b_{1,1}$	X
Fallos de cache totales			7

$b_{1,0}$ falla!!!

Primera iteración (K=0)

$c_{0,0} = a_{0,0} \cdot b_{0,0} + a_{0,1} \cdot b_{1,0}$

Segunda iteración (K=1)

$c_{0,0} = a_{0,0} \cdot b_{0,0} + a_{0,1} \cdot b_{1,0}$

B por columnas

```
For i = 0 to N-1
 For j = 0 to N-1
  For k = 0 to N-1
   // C(i,j)+=A(i,k)+B(k,j)
    C[i*N+j] += A[i*N+k] * B[k+j*N]
```

$b_{0,0}$	$b_{1,0}$	$b_{0,1}$	$b_{1,1}$
0	1	2	3

Memoria

`C[0] += A[0] * B[0]`

Variable	Palabra 1	Palabra 2	Fallo
Lineas i, j k … ya no fallan			
C	$c_{0,0}$	$c_{0,1}$	X
A	$a_{0,0}$	$a_{0,1}$	X
B	$b_{0,0}$	$b_{1,0}$	X
Fallos de cache totales			6

`C[0] += A[1] * B[1]`

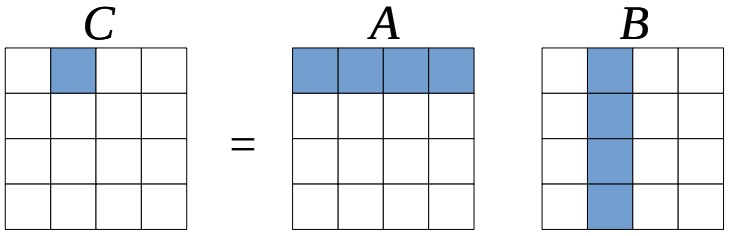
Variable	Palabra 1	Palabra 2	Fallo
Lineas i, j k … ya no fallan			
Lineas C y A … ya no fallan			
B	$b_{0,0}$	$b_{1,0}$	
Fallos de cache totales			6

$b_{1,0}$ NO falla!!!

Principio de localidad
Aplicación a la multiplicación de matrices

- En nuestro ejemplo, almacenar en memoria por filas las matrices A y C y por columnas la matriz B, nos permite obtener un fallo menos de caché en la primera iteración.
- Aunque un fallo parece despreciable, resulta significativo al seguir iterando y los fallos serán mayores a medida que aumentamos el tamaño de las matrices.

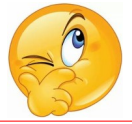
- En general, por la naturaleza del **patrón de acceso** en una multiplicación de matrices, almacenar en memoria por filas las matrices A y C, y por columnas la matriz B, nos permite obtener **menos fallos de caché** alcanzando mayor rendimiento.



- En la práctica, se pueden utilizar herramientas de **contadores hardware** (PMC siglas en inglés de Performance monitoring counters) para contar los fallos de cache de los algoritmos.

Principio de localidad
Matrices en los lenguajes de programación

- Todos los lenguajes de programación permiten declarar matrices.



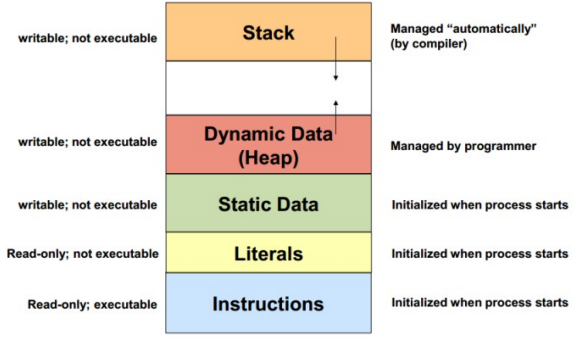
¿Cómo almacenan los lenguajes una matriz de NxN en memoria?

¿Cómo puede el programador indicar que una matriz debe almacenarse por filas o por columnas?

- **El lenguaje C** permite crear matrices de la siguiente forma:

```
int A [N][N];
int B [N][N];
int C [N][N];
```

- Declarar las variables de esta forma tiene **desventajas**:
  - El lenguaje determina como se almacenan (El lenguaje C siempre por filas).
  - Se almacenan en memoria estática (Pila o Stack).
  - El tamaño de las matrices estará limitado por la Pila.

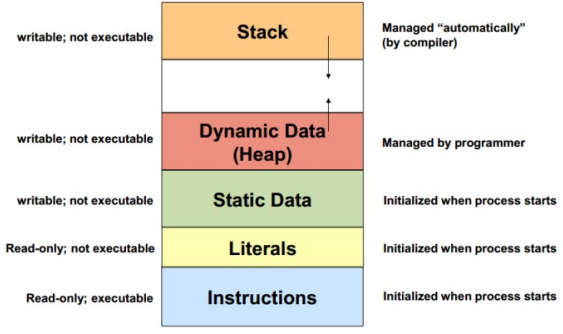


Principio de localidad
Matrices en los lenguajes de programación

- Para controlar el orden en memoria de las matrices y que estas puedan tener mayor tamaño, podemos definirlas en lenguaje C como sigue:

```
tipo* A;
tipo* B;
tipo* C;
…
A= (tipo*)malloc(sizeof(tipo)*N*N);
B= (tipo*)malloc(sizeof(tipo)*N*N);
C= (tipo*)malloc(sizeof(tipo)*N*N);
…
free(A);
free(B);
free(C);
```

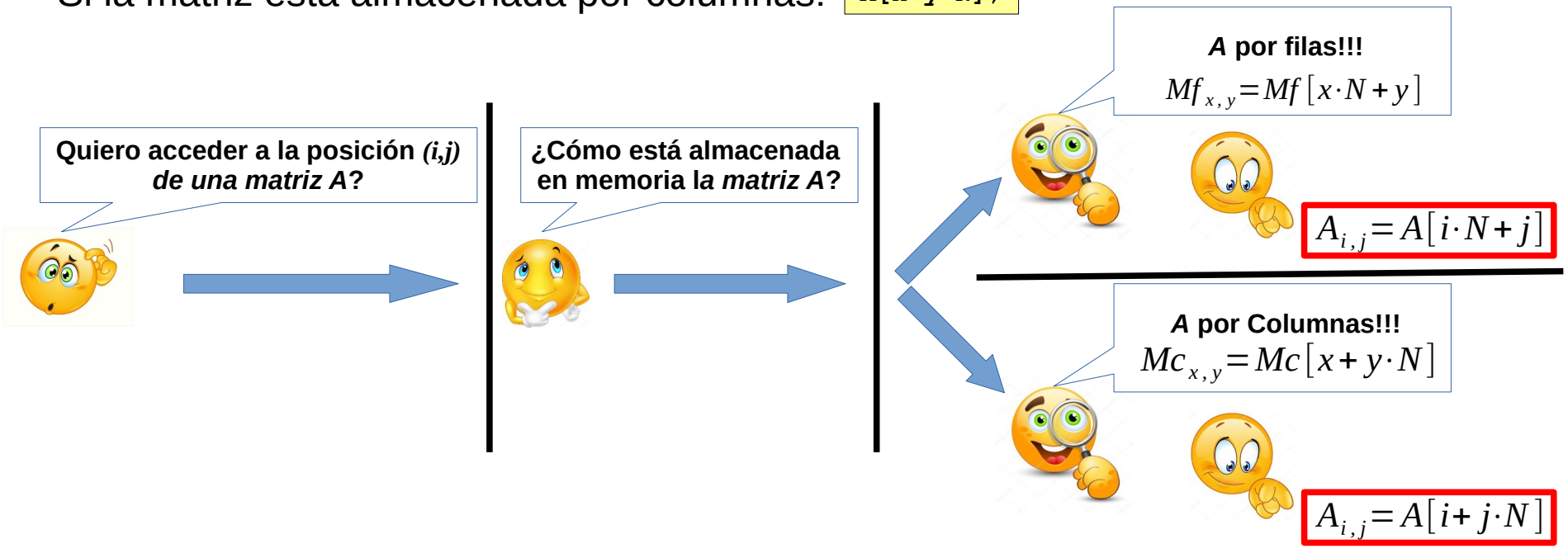
- Declarar las variables de esta forma tiene **ventajas**:
  - El programador determina como se almacenan en memoria.
  - Se almacenan en memoria dinámica (Heap).
  - El tamaño de las matrices será eventualmente “ilimitado”.



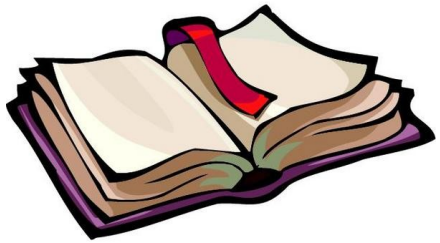
Principio de localidad
Matrices en los lenguajes de programación

¿Cómo acceder al elemento *(x,y)* de una matriz M?

- Si la matriz está almacenada por filas: `M[x*N+y];`
- Si la matriz está almacenada por columnas: `M[x+y*N];`



Agenda



I. Sistema de memoria
   i. Limitaciones del sistema de memoria
   ii. La importancia de la memoria caché
      i. Principio de localidad
      ii. Conceptos asociados a la memoria cache (Estados, Niveles, Contención, Coherencia)
II. Principio de localidad aplicado a la multiplicación de matrices
III. Minimizando el espacio de memoria. Caso de estudio matrices triangulares

Minimizando el espacio de memoria
Matrices Triangulares

- Una matriz triangular es una matriz cuadrada donde los elementos por encima o por debajo de la diagonal son cero:

$$U = \begin{pmatrix} u_{0,0} & u_{0,1} & u_{0,2} \\ 0 & u_{1,1} & u_{1,2} \\ 0 & 0 & u_{2,2} \end{pmatrix}$$

Matriz Triangular Superior

$$L = \begin{pmatrix} l_{0,0} & 0 & 0 \\ l_{1,0} & l_{1,1} & 0 \\ l_{2,0} & l_{2,1} & l_{2,2} \end{pmatrix}$$

Matriz Triangular Inferior

- Cuando los requerimientos de memoria son críticos y estas matrices son muy grandes, no es necesario almacenar una gran cantidad de ceros.
- La idea es minimizar el espacio de almacenamiento evitando guardar los elementos de la parte nula de la matriz.

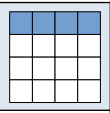
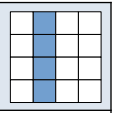
Vamos a suponer que NUNCA accederemos a las posiciones nulas.
Cualquier operación que las involucre da como resultado cero (Ejemplo: Multiplicación de matrices).

$$U = \begin{pmatrix} u_{0,0} & u_{0,1} & u_{0,2} \\ 0 & u_{1,1} & u_{1,2} \\ 0 & 0 & u_{2,2} \end{pmatrix} = \begin{pmatrix} u_{0,0} & u_{0,1} & u_{0,2} \\ \cancel{u_{1,0}} = 0 & u_{1,1} & u_{1,2} \\ \cancel{u_{2,0}} = 0 & \cancel{u_{2,1}} = 0 & u_{2,2} \end{pmatrix}$$

Minimizando el espacio de memoria
Matrices Triangulares

¿Cómo accedemos a la posición *T[i,j]* de una matriz triangular almacenada en forma reducida?

- El acceso a una matriz triangular almacenada de forma reducida en memoria va a depender de:
  - Si es triangular superior (U) o inferior (L)
  - Si está almacenada por filas o columnas.
- La combinación de estas características nos da 4 posibilidades:

	Almacenada por filas 	**Almacenada por columnas** 
Triangular Superior $U=\begin{pmatrix} u_{0,0} & u_{0,1} & u_{0,2} \\ 0 & u_{1,1} & u_{1,2} \\ 0 & 0 & u_{2,2} \end{pmatrix}$	$u_{0,0}$ $u_{0,1}$ $u_{0,2}$ $u_{1,1}$ $u_{1,2}$ $u_{2,2}$ 0 1 2 3 4 5 **Memoria**	$u_{0,0}$ $u_{0,1}$ $u_{1,1}$ $u_{0,2}$ $u_{1,2}$ $u_{2,2}$ 0 1 2 3 4 5 **Memoria**
Triangular inferior $L=\begin{pmatrix} l_{0,0} & 0 & 0 \\ l_{1,0} & l_{1,1} & 0 \\ l_{2,0} & l_{2,1} & l_{2,2} \end{pmatrix}$	$l_{0,0}$ $l_{1,0}$ $l_{1,1}$ $l_{2,0}$ $l_{2,1}$ $l_{2,2}$ 0 1 2 3 4 5 **Memoria**	$l_{0,0}$ $l_{1,0}$ $l_{2,0}$ $l_{1,1}$ $l_{2,1}$ $l_{2,2}$ 0 1 2 3 4 5 **Memoria**

Minimizando el espacio de memoria
Matrices Triangulares

- **Matriz triangular superior almacenada por filas** 
- Matriz triangular inferior almacenada por columnas
- Matriz triangular inferior almacenada por filas
- Matriz triangular superior almacenada por columnas

Minimizando el espacio de memoria
Matriz triangular superior almacenada por filas

- **Matriz triangular superior** almacenada **por filas** como un arreglo en memoria.
- Cada vez que se accede a una nueva fila, debe restarse del cálculo $U[i*N + j]$ la cantidad de elementos en cero hasta esa posición.
- Para una matriz de NxN:

$$U=\begin{pmatrix} u_{0,0} & u_{0,1} & u_{0,2} & \dots & u_{0,n-1} \\ 0 & u_{1,1} & u_{1,2} & \dots & u_{1,n-1} \\ 0 & 0 & u_{2,2} & \dots & u_{2,n-1} \\ \dots & \dots & \dots & \dots & \dots \\ 0 & 0 & 0 & \dots & u_{n-1,n-1} \end{pmatrix}$$

Ceros hasta la **fila 0: 0**

Ceros hasta la **fila 1: 1**

Ceros hasta la **fila 2: 3**

Ceros hasta la **fila 3: 6**

Ceros hasta la **fila i-ésima:**

$$\sum_{fila=0}^{i} fila = \frac{i\cdot(i+1)}{2}$$

Acceder al elemento *U[i,j]*:

$$U[i,j]=U_{arreglo}\left[i\cdot N+j-\frac{i\cdot(i+1)}{2}\right]$$

Ejemplo N=3

$$U=\begin{pmatrix} u_{0,0} & u_{0,1} & u_{0,2} \\ 0 & u_{1,1} & u_{1,2} \\ 0 & 0 & u_{2,2} \end{pmatrix}$$

$u_{0,0}$	$u_{0,1}$	$u_{0,2}$	$u_{1,1}$	$u_{1,2}$	$u_{2,2}$
0	1	2	3	4	5

Memoria

	i	j	$U[i,j]=U_{arreglo}\left[i\cdot N+j-\frac{i\cdot(i+1)}{2}\right]$
$u_{0,0}$	0	0	0
$u_{0,1}$	0	1	1
$u_{0,2}$	0	2	2
$u_{1,1}$	1	1	3
$u_{1,2}$	1	2	4
$u_{2,2}$	2	2	5

Minimizando el espacio de memoria
Matrices Triangulares

- Matriz triangular superior almacenada por filas
- **Matriz triangular inferior almacenada por columnas** 
- Matriz triangular inferior almacenada por filas
- Matriz triangular superior almacenada por columnas

Minimizando el espacio de memoria
Matriz triangular inferior almacenada por columnas

- **Matriz triangular inferior** almacenada **por columnas** como un arreglo en memoria.
- Cada vez que se accede a una nueva columna, debe restarse del cálculo $L[i + j*N]$ la cantidad de elementos en cero hasta esa posición.
- Para una matriz de NxN:

$$L=\begin{pmatrix} l_{0,0} & 0 & 0 & \dots & 0 \\ l_{1,0} & l_{1,1} & 0 & \dots & 0 \\ l_{2,0} & l_{2,1} & l_{2,2} & \dots & 0 \\ \dots & \dots & \dots & \dots & \dots \\ l_{n-1,0} & l_{n-1,1} & l_{n-1,2} & \dots & l_{n-1,n-1} \end{pmatrix}$$

Ceros hasta la **columna 0: 0**

Ceros hasta la **columna 1: 1**

Ceros hasta la **columna 2: 3**

Ceros hasta la **columna 3: 6**

Ceros hasta la **columna j-ésima:**

$$\sum_{columna=0}^{j} columna = \frac{j\cdot(j+1)}{2}$$

Acceder al elemento *L[i,j]*:

$$L[i,j]=L_{arreglo}\left[i+j\cdot N-\frac{j\cdot(j+1)}{2}\right]$$

Ejemplo N=3

$$L=\begin{pmatrix} l_{0,0} & 0 & 0 \\ l_{1,0} & l_{1,1} & 0 \\ l_{2,0} & l_{2,1} & l_{2,2} \end{pmatrix}$$

$l_{0,0}$	$l_{1,0}$	$l_{2,0}$	$l_{1,1}$	$l_{2,1}$	$l_{2,2}$
0	1	2	3	4	5

Memoria

	i	j	$L[i,j]=L_{arreglo}\left[i+j\cdot N-\frac{j\cdot(j+1)}{2}\right]$
$l_{0,0}$	0	0	0
$l_{1,0}$	1	0	1
$l_{2,0}$	2	0	2
$l_{1,1}$	1	1	3
$l_{2,1}$	2	1	4
$l_{2,2}$	2	2	5

Minimizando el espacio de memoria
Matrices Triangulares

- Matriz triangular superior almacenada por filas
- Matriz triangular inferior almacenada por columnas
- **Matriz triangular inferior almacenada por filas**

- Matriz triangular superior almacenada por columnas

Minimizando el espacio de memoria
Matriz triangular inferior almacenada por filas

- **Matriz triangular inferior** almacenada **por filas** como un arreglo en memoria.
- Cada vez que se accede a una nueva fila, debe restarse del cálculo $L[i*N + j]$ la cantidad de elementos en cero hasta esa posición.
- Para una matriz de NxN:

$$L=\begin{pmatrix} l_{0,0} & 0 & 0 & \dots & 0 \\ l_{1,0} & l_{1,1} & 0 & \dots & 0 \\ l_{2,0} & l_{2,1} & l_{2,2} & \dots & 0 \\ \dots & \dots & \dots & \dots & \dots \\ l_{n-1,0} & l_{n-1,1} & l_{n-1,2} & \dots & l_{n-1,n-1} \end{pmatrix}$$

Ceros hasta la **fila 0: 0**

Ceros hasta la **fila 1: N-1**

Ceros hasta la **fila 2: 2N-3**

Ceros hasta la **fila 3: 3N-6**

Ceros hasta la **fila i-ésima:**

$$i\cdot N-\sum_{fila=0}^{i} fila = i\cdot N-\frac{i\cdot(i+1)}{2}$$

Acceder al elemento *L[i,j]*:

$$L[i,j]=L_{arreglo}\left[i\cdot N+j-\left(i\cdot N-\frac{i\cdot(i+1)}{2}\right)\right]$$

Ejemplo N=3

$$L=\begin{pmatrix} l_{0,0} & 0 & 0 \\ l_{1,0} & l_{1,1} & 0 \\ l_{2,0} & l_{2,1} & l_{2,2} \end{pmatrix}$$

$l_{0,0}$	$l_{1,0}$	$l_{1,1}$	$l_{2,0}$	$l_{2,1}$	$l_{2,2}$
0	1	2	3	4	5

Memoria

	i	j	$L[i,j]=L_{arreglo}\left[i\cdot N+j-\left(i\cdot N-\frac{i\cdot(i+1)}{2}\right)\right]$
$l_{0,0}$	0	0	0
$l_{1,0}$	1	0	1
$l_{1,1}$	1	1	2
$l_{2,0}$	2	0	3
$l_{2,1}$	2	1	4
$l_{2,2}$	2	2	5

Minimizando el espacio de memoria
Matrices Triangulares

- Matriz triangular superior almacenada por filas
- Matriz triangular inferior almacenada por columnas
- Matriz triangular inferior almacenada por filas
- **Matriz triangular superior almacenada por columnas**

Minimizando el espacio de memoria
Matriz triangular superior almacenada por columnas

- **Matriz triangular superior** almacenada **por columnas** como un arreglo en memoria.
- Cada vez que se accede a una nueva columna, debe restarse del cálculo $U[i + j*N]$ la cantidad de elementos en cero hasta esa posición.
- Para una matriz de NxN:

$$U=\begin{pmatrix} u_{0,0} & u_{0,1} & u_{0,2} & \dots & u_{0,n-1} \\ 0 & u_{1,1} & u_{1,2} & \dots & u_{1,n-1} \\ 0 & 0 & u_{2,2} & \dots & u_{2,n-1} \\ \dots & \dots & \dots & \dots & \dots \\ 0 & 0 & 0 & \dots & u_{n-1,n-1} \end{pmatrix}$$

Ceros hasta la **columna 0: 0**

Ceros hasta la **columna 1: N-1**

Ceros hasta la **columna 2: 2N-3**

Ceros hasta la **columna 3: 3N-6**

Ceros hasta la **columna j-ésima:**

$$j\cdot N-\sum_{columna=0}^{j} columna = j\cdot N-\frac{j\cdot(j+1)}{2}$$

Acceder al elemento *U[i,j]*:

$$U[i,j]=U_{arreglo}\left[i+j\cdot N-\left(j\cdot N-\frac{j\cdot(j+1)}{2}\right)\right]$$

Ejemplo N=3

$$U=\begin{pmatrix} u_{0,0} & u_{0,1} & u_{0,2} \\ 0 & u_{1,1} & u_{1,2} \\ 0 & 0 & u_{2,2} \end{pmatrix}$$

$u_{0,0}$	$u_{0,1}$	$u_{1,1}$	$u_{0,2}$	$u_{1,2}$	$u_{2,2}$
0	1	2	3	4	5

Memoria

	i	j	$U[i,j]=U_{arreglo}\left[i+j\cdot N-\left(j\cdot N-\frac{j\cdot(j+1)}{2}\right)\right]$
$u_{0,0}$	0	0	0
$u_{0,1}$	0	1	1
$u_{1,1}$	1	1	2
$u_{0,2}$	0	2	3
$u_{1,2}$	1	2	4
$u_{2,2}$	2	2	5

Minimizando el espacio de memoria
Resumen

$$U[i,j] = U_{filas}\left[i \cdot N + j - \frac{i \cdot (i+1)}{2}\right]$$

$$L[i,j] = L_{columnas}\left[i + j \cdot N - \frac{j \cdot (j+1)}{2}\right]$$

$$L[i,j] = L_{filas}\left[i \cdot N + j - (i \cdot N - \frac{i \cdot (i+1)}{2})\right] = L_{filas}\left[\cancel{i \cdot N} + j - \cancel{i \cdot N} + \frac{i \cdot (i+1)}{2}\right] = L_{filas}\left[j + \frac{i \cdot (i+1)}{2}\right]$$

$$U[i,j] = U_{columnas}\left[i + j \cdot N - (j \cdot N - \frac{j \cdot (j+1)}{2})\right] = U_{columnas}\left[i + \cancel{j \cdot N} - \cancel{j \cdot N} + \frac{j \cdot (j+1)}{2}\right] = U_{columnas}\left[i + \frac{j \cdot (j+1)}{2}\right]$$

Minimizando el espacio de memoria
Matrices Triangulares

¿Cómo multiplicar con matrices triangulares?

- Dependiendo del tipo de multiplicación que hagamos (LM, UM, ML, MU), debemos definir que posiciones **NO recorrer** por ser **CERO**.
- Suponemos que las variables i y j representan los índices que nos ubican en la fila y columna, respectivamente.
- Las iteraciones **FOR** de i y j, no se ven afectadas porque definen los indices a calcular en la matriz *Resultado* (todas las posiciones!!!).
- Los límites lo define la iteración **FOR** de k.

NO!!!

```
For i = 0 to N-1
 For j = 0 to N-1
  For k = 0 to N-1
   if("k está en los límites")
     "Multiplicar"
```

Sólo recorremos las posiciones que no son cero.
Determinarlas en la práctica!!!

```
For i = 0 to N-1
 For j = 0 to N-1
  For k = ?? to ??
   "Multiplicar"
```

Minimizando el espacio de memoria
Trade-off (Cómputo - Memoria)

- **Trade-off:** situación en la cual aceptamos perder en cierta cualidad a cambio de ganar en otra.

- En general, la mejoras pueden introducir overhead. Si el overhead introducido por “la mejora” no se puede evitar se debe intentar minimizar, y de esta forma mitigar el impacto negativo en el rendimiento.

- En el caso de las matrices triangulares:
  - Ganamos espacio de almacenamiento
  - La cantidad de cálculos es mayor por lo tanto podría perderse en rendimiento

¿Qué ocurre empiricamente?
Implementarlo en la práctica y analizar lo que ocurre.