Técnicas SuperEscalares en la Paralelización de Bibliotecas de Computación Matricial sobre Procesadores Multinúcleo y GPUs

Enrique S. Quintana-Ortí
quintana@icc.uji.es

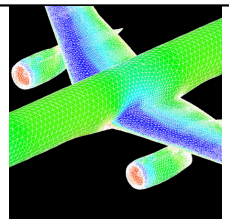
High Performance Computing & Architectures Group
Universidad Jaime I de Castellón

Santiago de Compostela – Enero, 2009

En colaboración con:

UJI	UT-Austin	BSC/UPC
S. Barrachina	Ernie Chan	R. M. Badia
M. Castillo	Robert van de Geijn	J. R. Herrero
F. D. Igual	Field G. Van Zee	J. Labarta
M. Marqués		J. M. Pérez
R. Mayo		
G. Quintana-Ortí		

Computación Matricial

Problemas principales	Fuentes de problemas grandes
Sistemas lineales $Ax = b$	Aeronáutica: BEM 
Valores propios $Ax = \lambda x$	Química computacional 
Valores singulares $A = U\Sigma V^T$	Minería de datos 

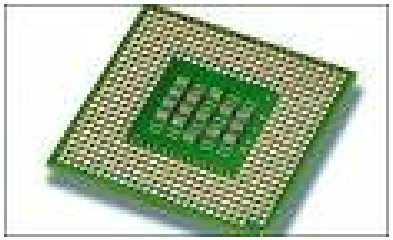
Computación Matricial

Problemas principales	Algoritmos
Sistemas lineales $Ax = b$	Factorizaciones LU, QR y de Cholesky
Valores propios $Ax = \lambda x$	Algoritmos QR y de Jacobi
Valores singulares $A = U\Sigma V^T$	Factorizaciones SVD, QRP

Bibliotecas de Computación Matricial

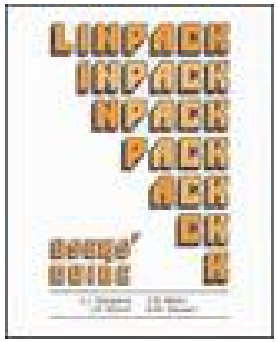
Persiguiendo a la arquitectura HPC del momento...

ISA vectoriales



Memoria caché

Memoria distribuida



Bibliotecas de Computación Matricial

Persiguiendo a la ¿arquitectura HPC del momento?

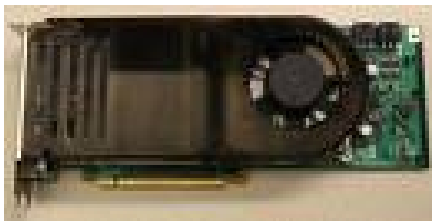
La Ley de Moore sigue vigente, pero...

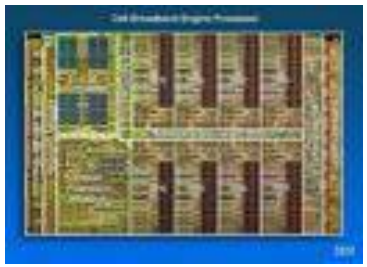
1. No es posible seguir aumentando la frecuencia de los procesadores por problemas de disipación del calor
2. No hay más paralelismo disponible a nivel de instrucción
3. La latencia de acceso a memoria no ha cambiado significativamente

Bibliotecas de Computación Matricial

Persiguiendo a la ¿arquitectura HPC del momento?

Aceleradores hardware:

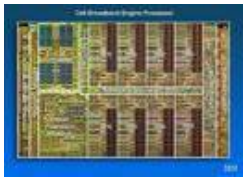




Procesadores multinúcleo:

Bibliotecas de Computación Matricial






Problemas y retos

Escalabilidad	Cientos de núcleos
Localidad	Accesos fuera del chip muy costosos
Heterogeneidad	Chips heterogéneos
Coste	Reescribir de nuevo las bibliotecas

Bibliotecas de Computación Matricial

Técnicas SuperEscalares en la Paralelización de Bibliotecas de Computación Matricial sobre Procesadores Multinúcleo y GPUs:

- Ejecución fuera de orden dirigida por las dependencias de datos (*data-flow parallelism*)
- Uso de cachés (*software*) para ocultar la existencia de espacios de direcciones distribuidos y la latencia del sistema de memoria (incluido el disco):
  - *Distributed Shared Memory*
  - *Prefetch* inteligente

Factorización de Cholesky

Para una matriz $A \rightarrow n \times n$ simétrica definida positiva, calcular

$$A = L \cdot L^T$$

con el factor de Cholesky $L \rightarrow n \times n$ triangular inferior

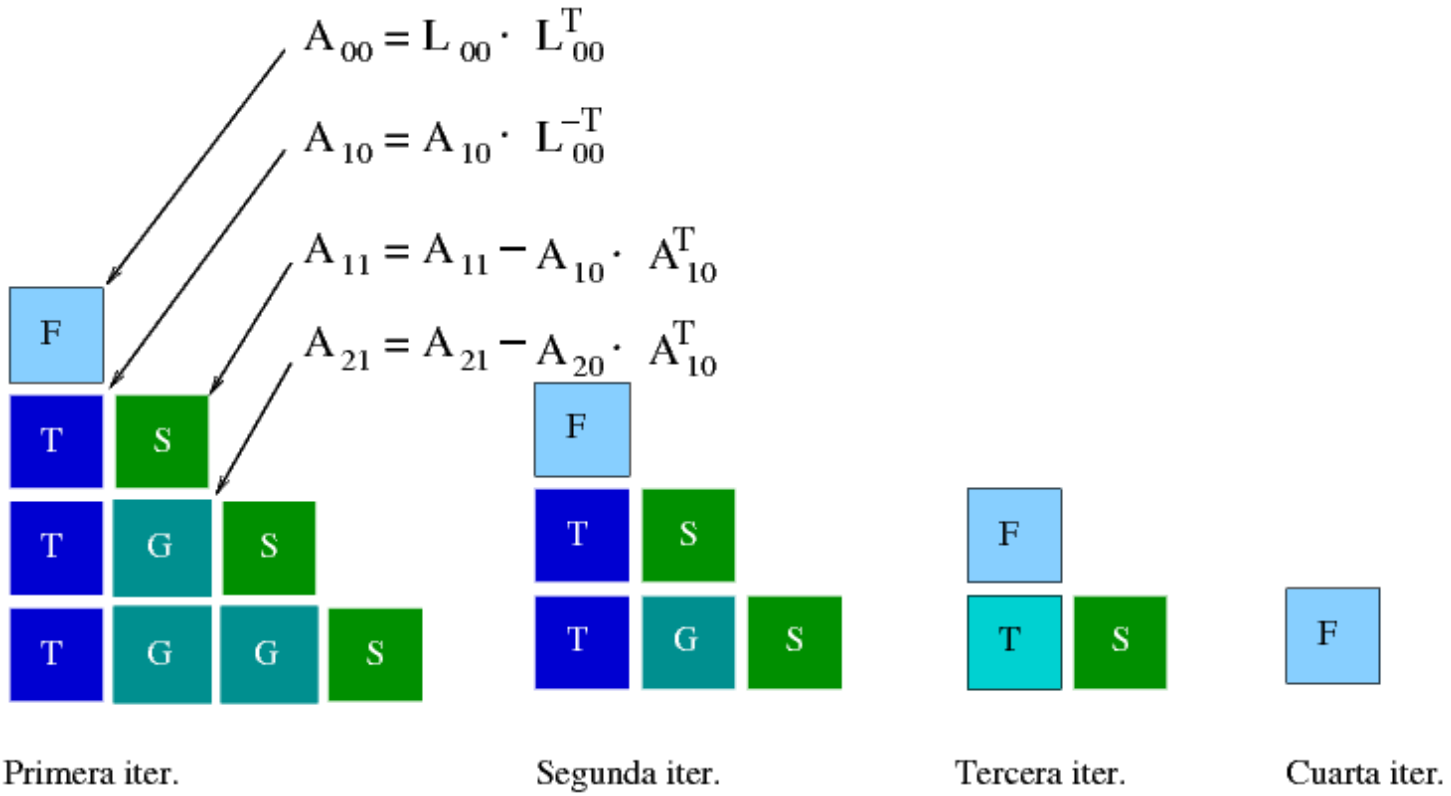
Primer paso en la solución de sistemas de ecuaciones lineales

$$\begin{aligned} Ax = b \quad \equiv \quad (LL^T)x &= b \\ Ly &= b \quad (\rightsquigarrow y) \\ L^T x &= y \quad (\rightsquigarrow x) \end{aligned}$$

Factorización de Cholesky

Algoritmo (por bloques):

$$A = \left(\begin{array}{c|c|c|c} A_{00} & A_{01} & A_{02} & A_{03} \\ \hline A_{10} & A_{11} & A_{12} & A_{03} \\ \hline A_{20} & A_{21} & A_{22} & A_{03} \\ \hline A_{30} & A_{31} & A_{32} & A_{33} \end{array}\right)$$



Paralelismo de Flujo de Datos

Ejecución fuera de orden dirigida
por las dependencias de datos (*data-flow parallelism*)

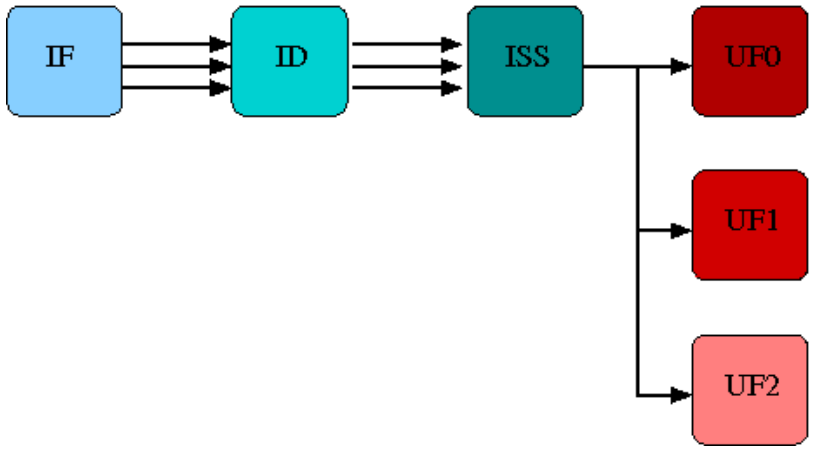
Objetivo: Aumentar el grado de paralelismo durante la ejecución de los algoritmos de computación matricial

Paralelismo de Flujo de Datos

Código escalar:

```
loop: ld   f0, 0(r1)
      addd f4, f0, f2
      sd   f4, 0(r1)
      addi r1, r1, #8
      subi r2, r2, #1
      bnez r2, loop
```

Procesador superescalar:



- Las etapas IF e ID se ejecutan en orden, pero la etapa ISS puede proceder posiblemente fuera de orden:
  - Se respeta el orden de las instrucciones que tienen dependencias de datos (RAW) entre sí...
  - pero las que no presentan dependencias pueden reordenarse

Paralelismo de Flujo de Datos

¿Algo similar para códigos de computación matricial?

Factorización de Cholesky: $A = L \cdot L^T$

```
for (k = 0; k < nt; k++){
  chol_dpotrf(A[k,k]);                          // Akk = Lkk*Lkk^T
  for (i=k+1; i<nt;i++)
    chol_dtrsm(A[k,k], A[i,k]);                 // Aik = Aik*Lkk^-T
  for (i=k+1; i<nt;i++){
    for (j=k+1; j<i; j++)
      chol_dgemm(A[i,k], A[j,k], A[i,j]); // Aij = Aij-Aik*Ajk^T
    chol_dsyrk(A[i,k], A[i,i]);                // Aii = Aii-Aik*Aik^T
  }
}
```

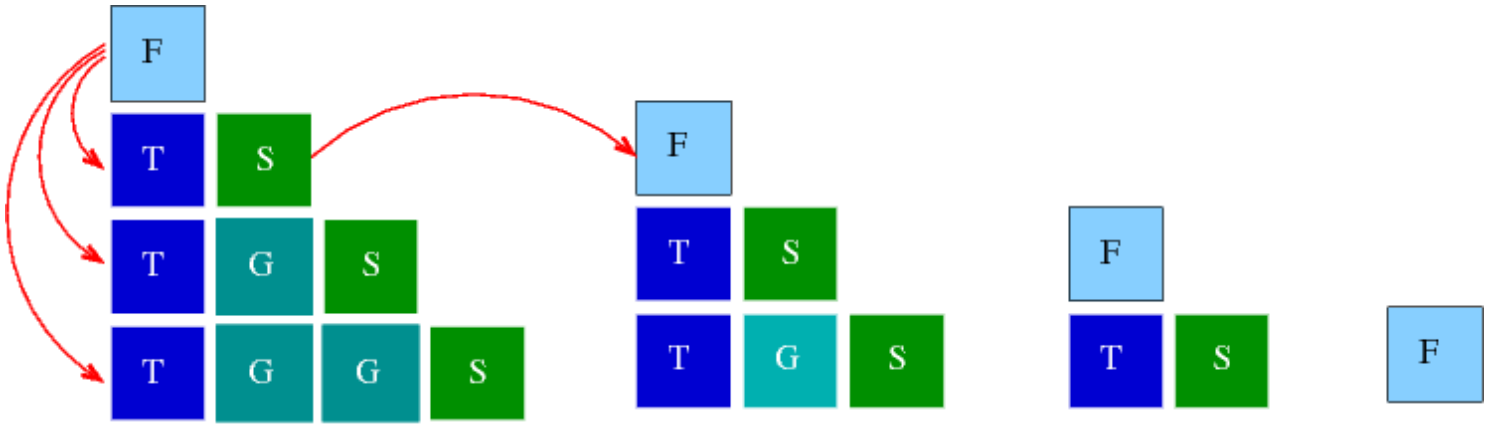
- Paralelización software frente a hardware
- Objetivo son los múltiples núcleos del procesador, no las unidades funcionales
- Paralelismo a nivel de tarea/hebra frente a paralelismo a nivel de instrucción

Paralelismo de Flujo de Datos

Los bloques leídos/escritos en la operación determinan las dependencias, del mismo modo que ocurre con los operandos de las instrucciones de un código escalar:

```
loop: ld   f0, 0(r1)        for (k = 0; k < nt; k++){
      addd f4, f0, f2         chol_dpotrf(A[k,k]);            // Akk = Lkk*Lkk^T
      sd   f4, 0(r1)          for (i=k+1; i<nt;i++)
      addi r1, r1, #8           chol_dtrsm(A[k,k], A[i,k]);   // Aik = Aik*Lkk^-T
      ...                       ...
```

Las dependencias entre bloques de una operación matricial definen un árbol de tareas:

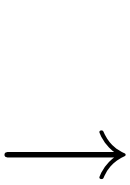


Paralelismo de Flujo de Datos

Código por bloques:

```
for (k = 0; k < nt; k++){
  chol_dpotrf(A[k,k]);

  for (i=k+1; i<nt;i++)
    chol_dtrsm(A[k,k], A[k,i]);
  ...
}
```



Procesador multinúcleo:

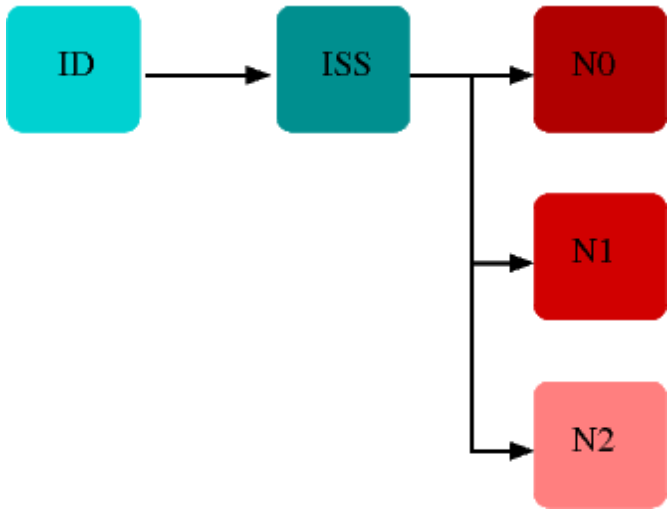
- ¿Cómo generamos el árbol de tareas?
- ¿Qué debemos tener en cuenta para ejecutar las tareas del árbol?

Paralelismo de Flujo de Datos

Run-time

Compuesto por dos etapas (parcialmente solapables):

- Decodificación (ID): Generación del árbol de tareas mediante el análisis simbólico del código
- Emisión (ISS): Ejecución del árbol de tareas consciente de las dependencias y de la arquitectura



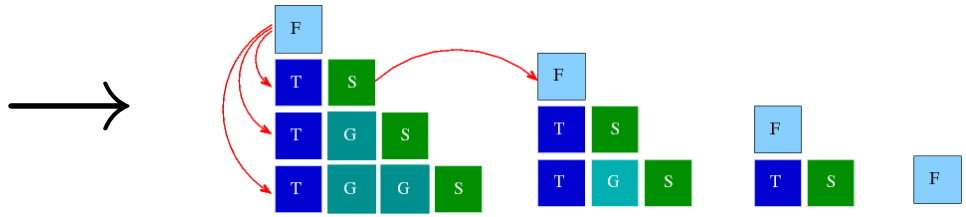
Etapa de decodificación

Análisis simbólico del código

Código por bloques:

```
for (k = 0; k < nt; k++){
  chol_dpotrf(A[k,k]);

  for (i=k+1; i<nt;i++)
    chol_dtrsm(A[k,k], A[k,i]);
  ...
}
```



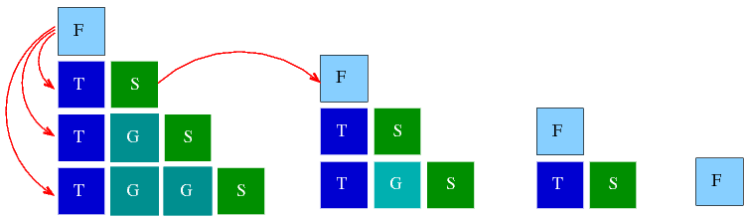
Paralelismo de Flujo de Datos

Etapa de emisión

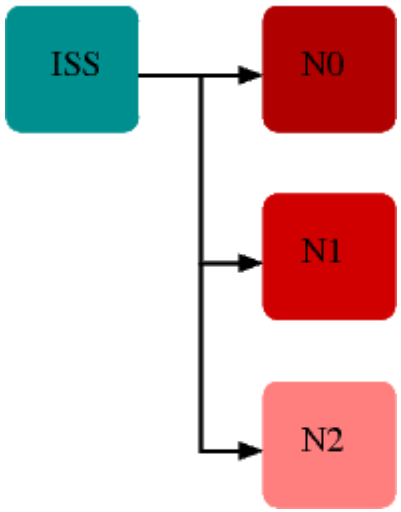
Ejecución real de las operaciones codificadas en las tareas

- Planificación temporal de las tareas en función de las dependencias
- Asignación espacial de tareas a núcleos consciente de la historia (localidad espacial de datos)

Árbol de tareas:



Procesador multinúcleo:



Paralelismo de Flujo de Datos

Implementaciones del run-time:

- SuperMatrix
  - Bloques de lectura/escritura definidos intrínsecamente por las operaciones
  - Sólo válido para computación matricial densa

- SMPSs
  - Bloques de lectura/escritura definidos por pragmas

```
pragma css task inout(A[ts*ts])
void chol_dpotrf(double *A);
```

  - Válido para cualquier operación por bloques

Resultados Experimentales

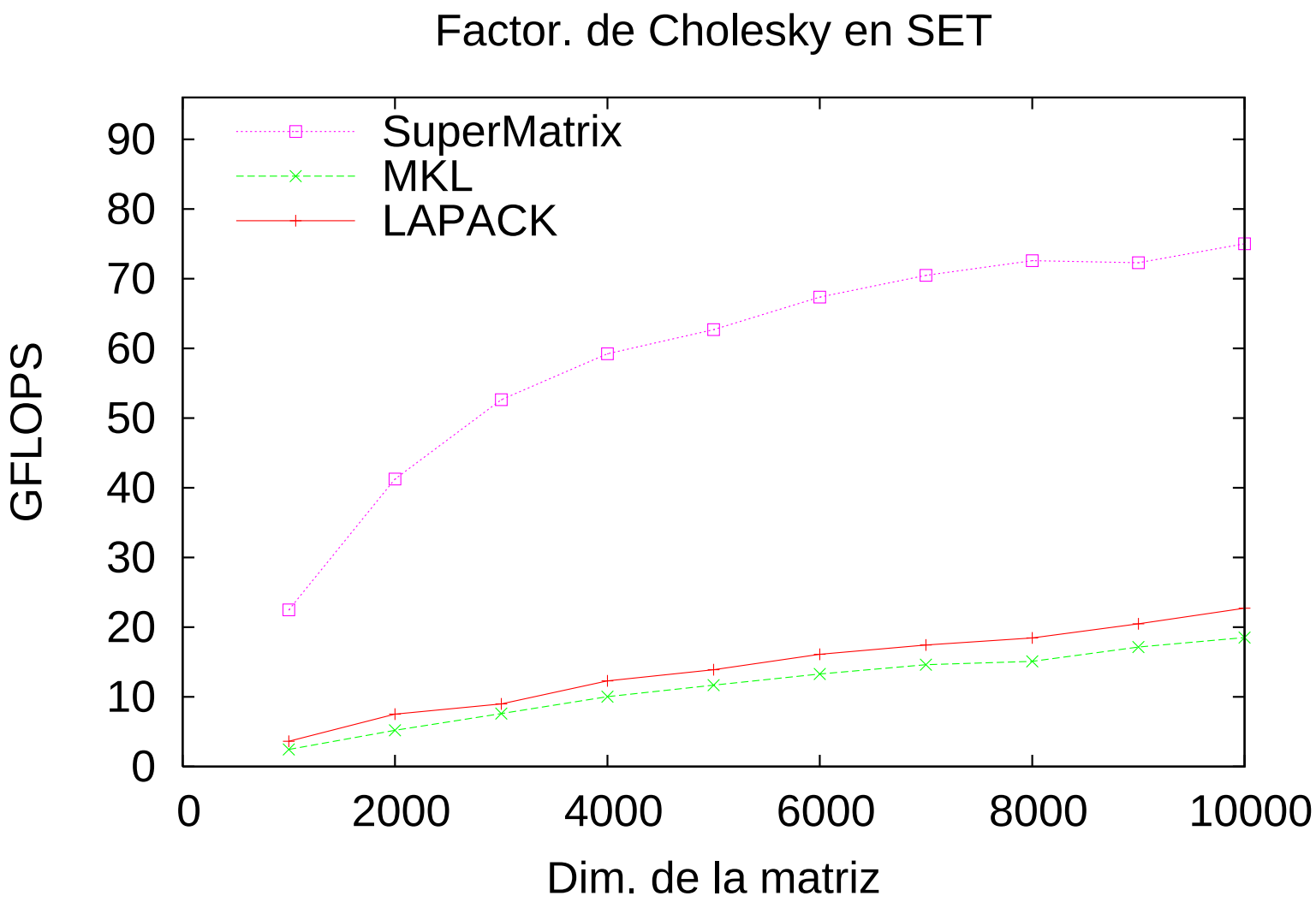
Arquitecturas

Plataforma	Especificaciones
SET	CC-NUMA con 16 procesadores Intel Itanium 2
NEUMANN	SMP con 8 procesadores dual core AMD Opteron

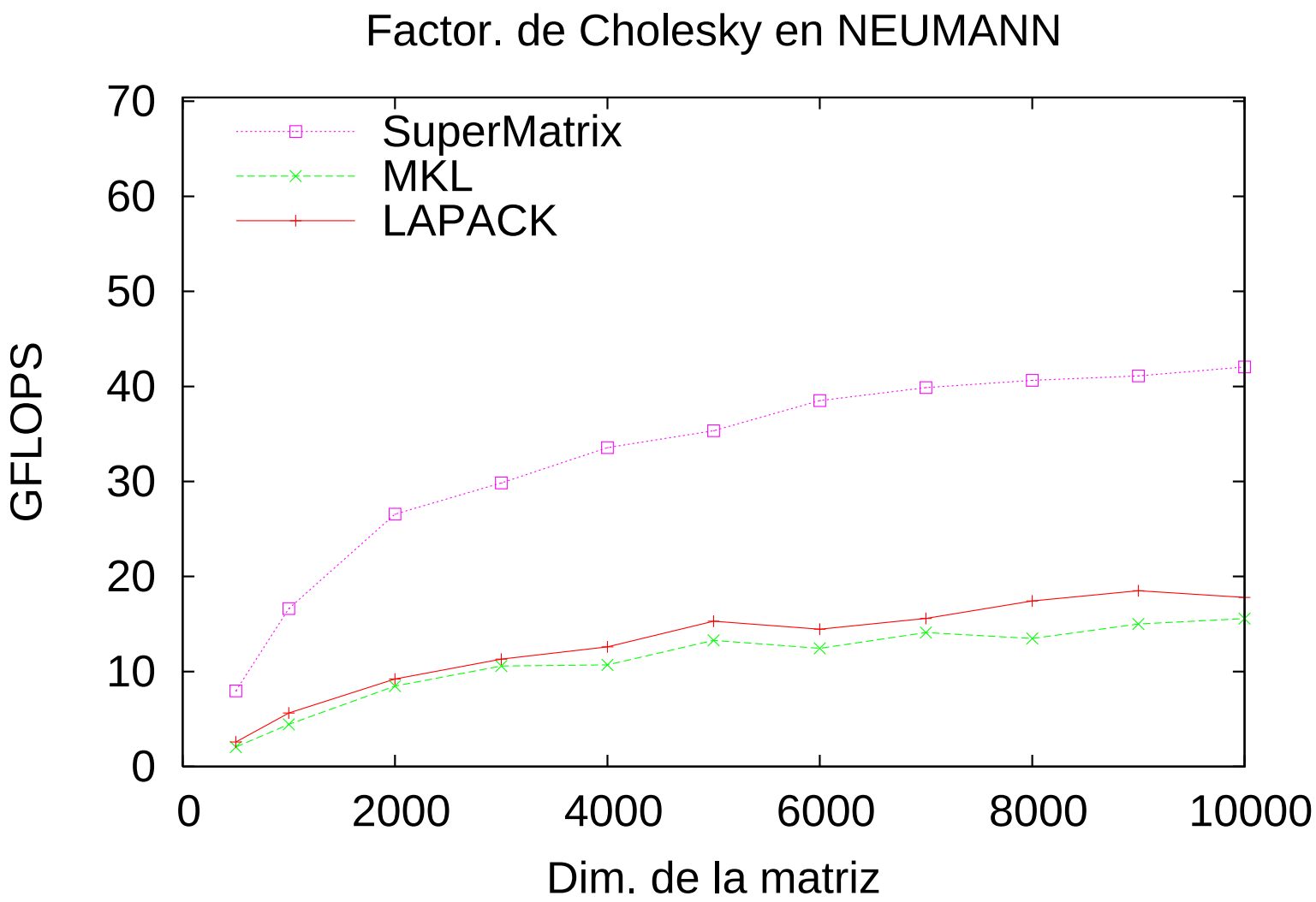
Implementaciones

- Rutina LAPACK 3.0 + MKL multihebra
- Rutina multihebra de MKL
- Run-time SuperMatrix + MKL monohebra (+ almacenamiento por bloques)

Resultados Experimentales



Resultados Experimentales



Memoria Distribuida Virtualmente Compartida (DSM)

Uso de cachés (*software*)

Objetivos:

- Ocultar la existencia de espacios de direcciones locales
- Ocultar la latencia del sistema de memoria

Memoria Distribuida Virtualmente Compartida (DSM)

Código por bloques:

```
for (k = 0; k < nt; k++){
  chol_dpotrf(A[k,k]);

  for (i=k+1; i<nt;i++)
    chol_dtrsm(A[k,k], A[k,i]);
  ...
}
```

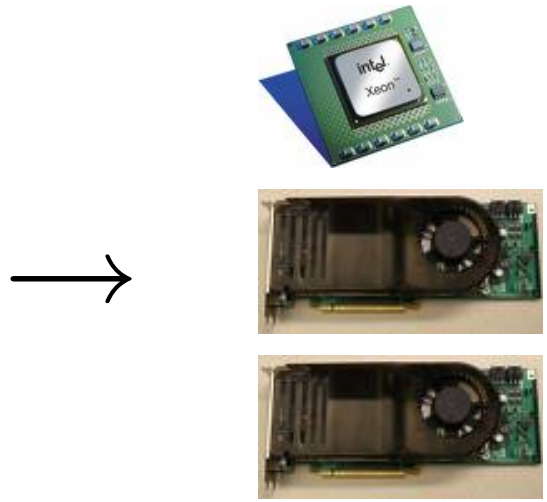
Servidor con varios aceleradores hardware:

- El *Host* (servidor) y cada *device* (acelerador) tienen espacios de direcciones (memoria) propios

Memoria Distribuida Virtualmente Compartida (DSM)

Solución básica

- Antes de iniciar una tarea, se transfieren los datos necesarios a una GPU libre
- Cuando termina la ejecución, se recuperan los resultados
- No se explota la localidad de referencia :-(



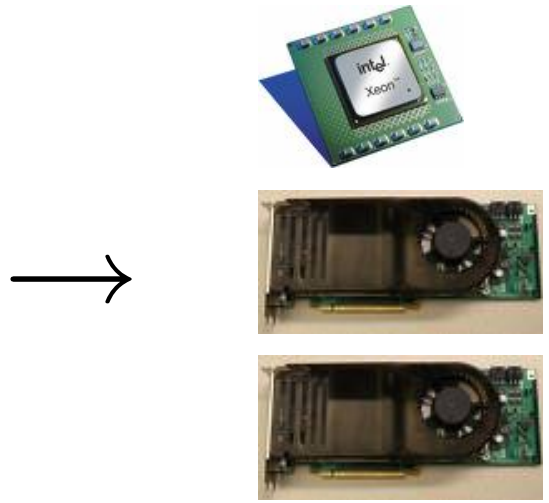
Memoria Distribuida Virtualmente Compartida (DSM)

Solución elaborada

- Asignación prefijada de tareas a GPUs: 2-D

$$\begin{pmatrix} \bar{A}_{00} & & & \\ \bar{A}_{10} & \bar{A}_{11} & & \\ \bar{A}_{20} & \bar{A}_{21} & \bar{A}_{22} & \\ \bar{A}_{30} & \bar{A}_{31} & \bar{A}_{32} & \bar{A}_{33} \end{pmatrix}$$

- Caché software en *devices*
- *write-back* para bloques en memoria del *host*
- *write-invalidate* para bloques en memoria de otros *devices*



Resultados Experimentales

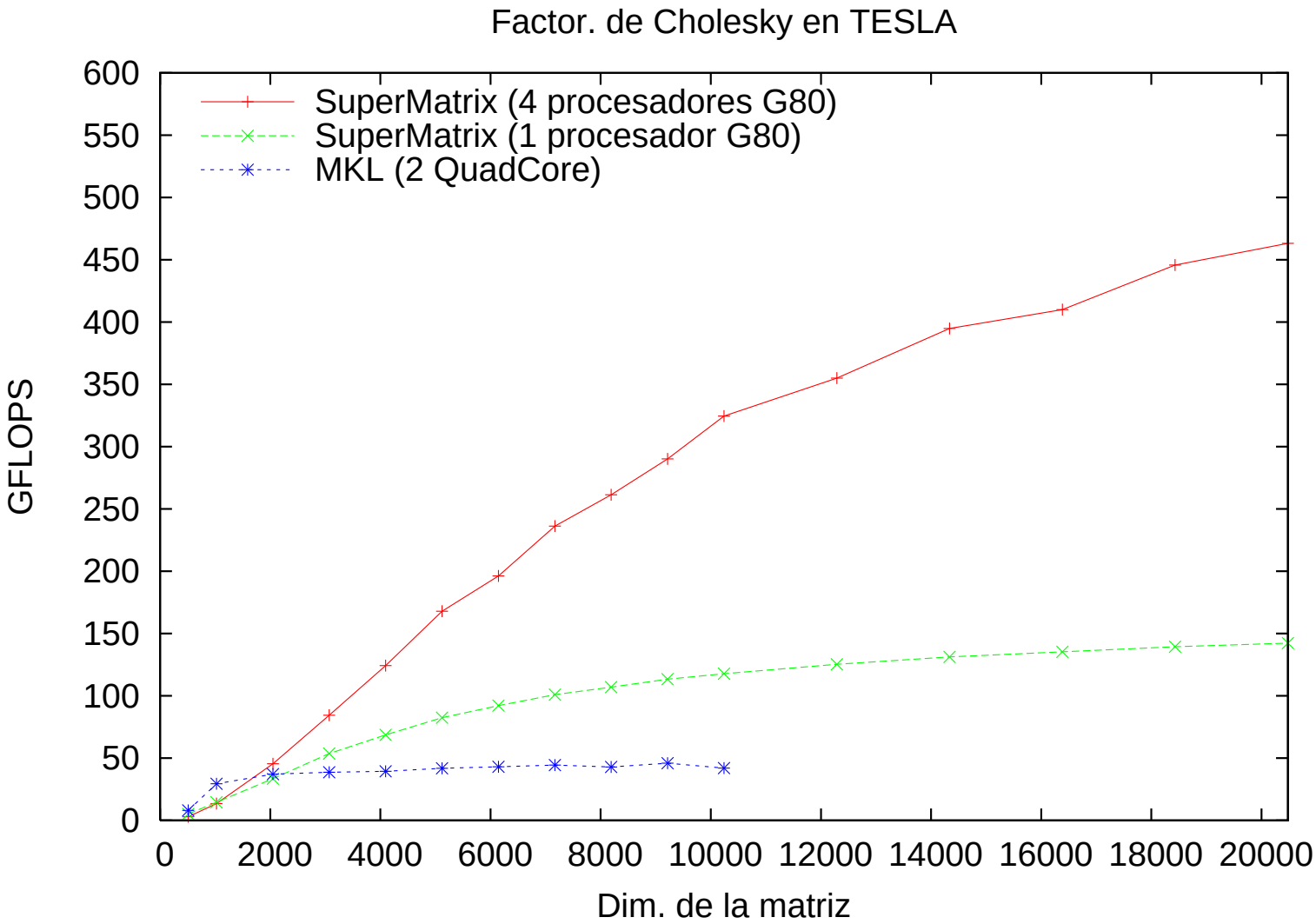
Arquitectura

Plataforma	Especificaciones
TESLA	SMP con 2 procesadores Intel Xeon QuadCore Tesla S870 con 4 NVIDIA G80 Conexión: 2 PCI Express Gen2

Implementaciones

- Rutina in-core multihebra de MKL
- Run-time SuperMatrix en 1 procesador G80
- Run-time SuperMatrix en 4 procesador G80

Resultados Experimentales



Computación con Datos en Disco (OOC)

Uso de cachés (*software*)

Objetivos:

- Ocultar la latencia del sistema de memoria: disco

Computación con Datos en Disco (OOC)

Interés:

- Algunos problemas de computación matricial densa son "realmente" grandes: 795k$\times$ 795k
- El coste de la memoria no es despreciable
- Con un poco de paciencia, es posible resolver problemas grandes usando un simple PCs, con un coste mucho menor del de un cluster

Computación con Datos en Disco (OOC)

Código por bloques:

```
for (k = 0; k < nt; k++){
  chol_dpotrf(A[k,k]);

  for (i=k+1; i<nt;i++)
    chol_dtrsm(A[k,k], A[k,i]);
  ...
}
```

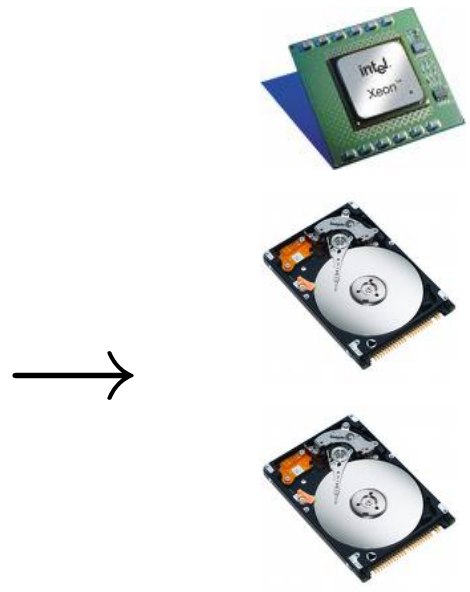
Servidor conectado a disco:

- El disco es bastante más lento que los niveles del sistema de memoria más cercanos al procesador
- El uso de E/S asíncrona complica bastante la programación

Computación con Datos en Disco (OOC)

Solución básica

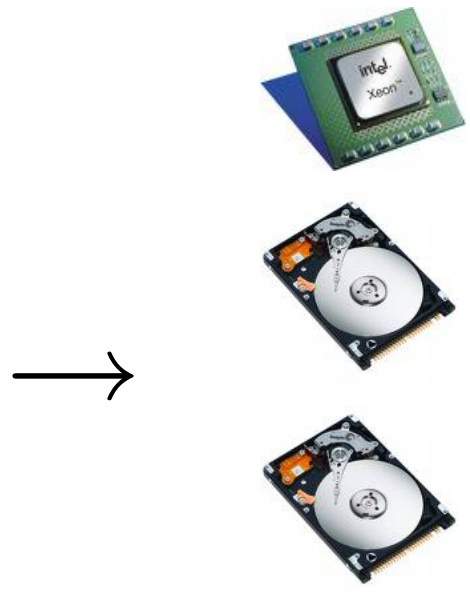
- Antes de iniciar una tarea, se recuperan los datos necesarios de disco
- Cuando termina la ejecución, se guardan los resultados en disco
- No se explota la localidad de referencia :-(
- No se solapan cálculos y operaciones de E/S :-(



Computación con Datos en Disco (OOC)

Solución elaborada

- Análisis simbólico del código para determinar la lista de tareas
- *Prefetch* inteligente para solapar cálculo y E/S mediante dos hebras
- Caché software en memoria principal



Resultados Experimentales

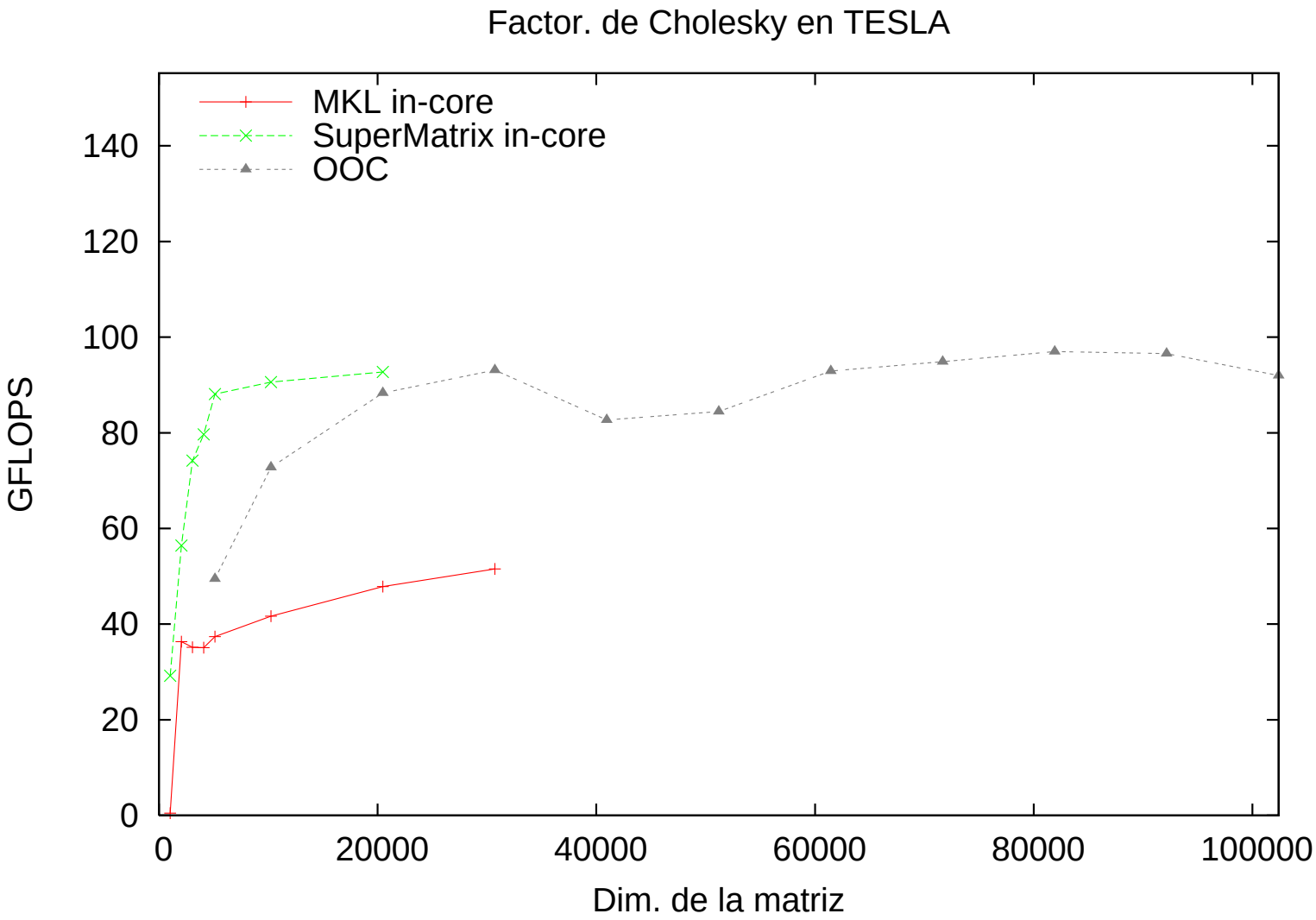
Arquitecturas

Plataforma	Especificaciones
TESLA	SMP con 2 procesadores Intel Xeon QuadCore SATA-I 160 GB, 1.5 Gb/s.

Implementaciones

- Rutina multihebra de MKL in-core
- SuperMatrix in-core
- OOC

Resultados Experimentales



Resumen

Técnicas SuperEscalares en la Paralelización de Bibliotecas de Computación Matricial sobre Procesadores Multinúcleo y GPUs:

- Ejecución fuera de orden dirigida por las dependencias de datos (*data-flow parallelism*)
- Uso de cachés (*software*) para ocultar la existencia de espacios de direcciones distribuidos y la latencia del sistema de memoria (incluido el disco):
  - *Distributed Shared Memory*
  - *Prefetch* inteligente