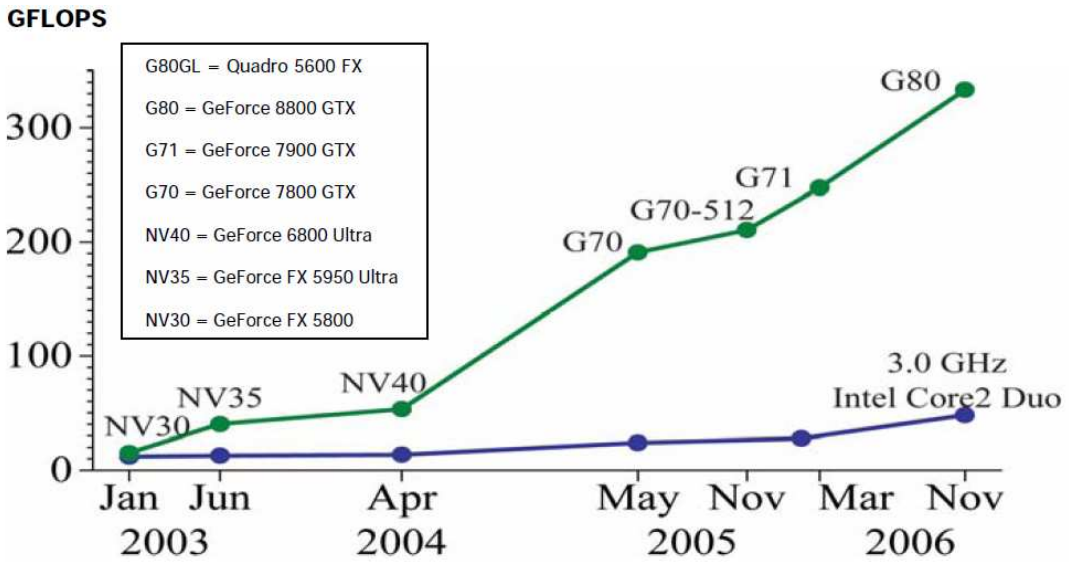
Solving Dense Linear Systems on Platforms with Multiple Hardware Accelerators

Maribel Castillo, Francisco D. Igual, Rafael Mayo, Gregorio Quintana-Ortí, Enrique S. Quintana-Ortí, Robert van de Geijn

Grupo de Arquitecturas y Computación de Altas Prestaciones
Universidad Jaume I de Castellón (Spain)

Motivación

Prestaciones



Facilidad de programación

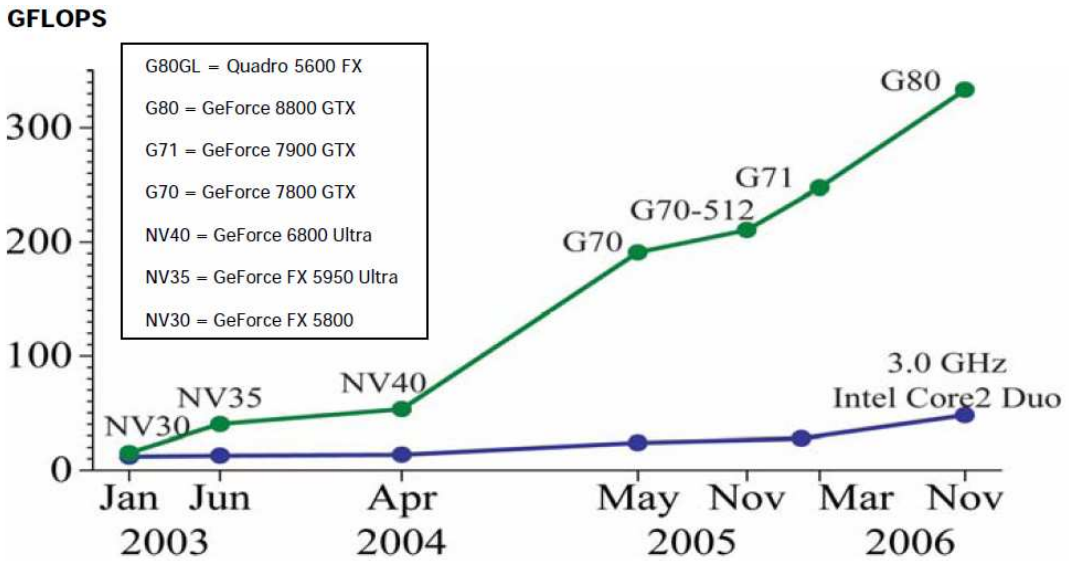
GPU → OpenGL + Cg

GPU → CUDA

multiGPU

Motivación

Prestaciones



Facilidad de programación

GPU

OpenGL + Cg

$\longrightarrow$

GPU

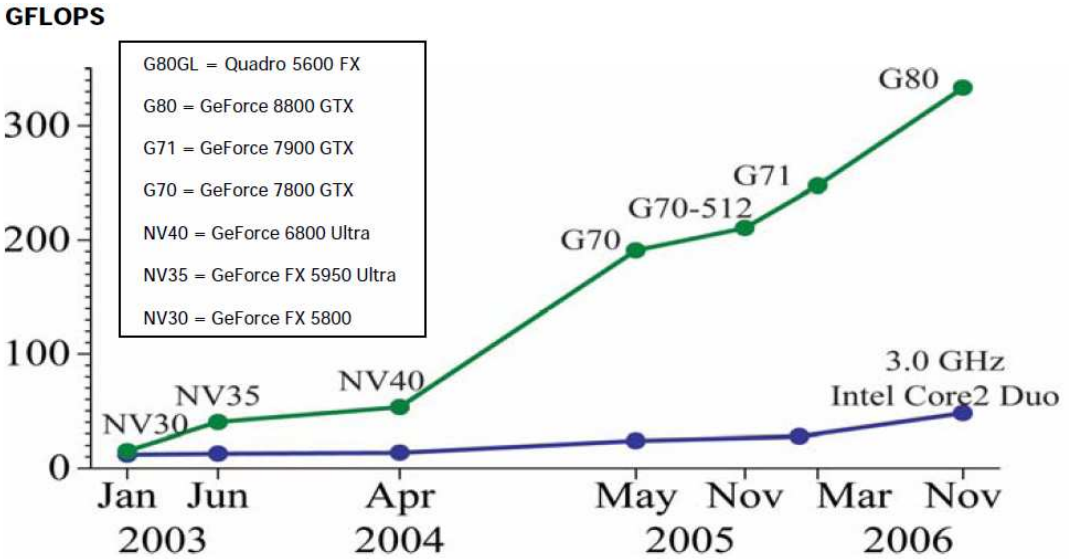
CUDA

$\longrightarrow$

multiGPU

Motivación

Prestaciones



Facilidad de programación

GPU

OpenGL + Cg

$\longrightarrow$

GPU

CUDA

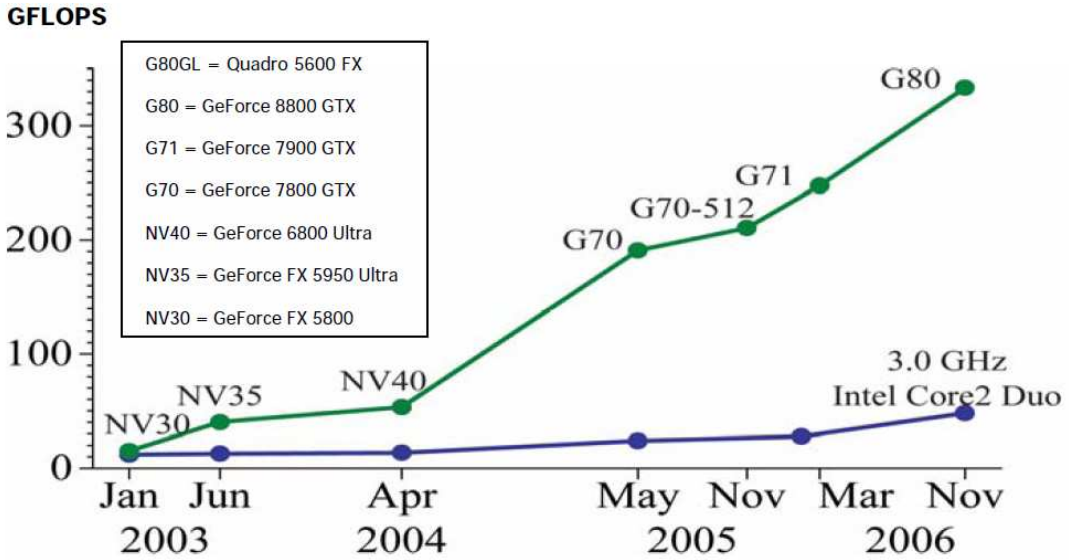
$\longrightarrow$

multiGPU

? ?

Motivación

Prestaciones



Facilidad de programación

GPU

OpenGL + Cg

$\longrightarrow$

GPU

CUDA

$\longrightarrow$

multiGPU

FLAME + SuperMatrix ? ?

Índice

Introducción

- Procesadores multicore/aceleradores hardware
- Elevado potencial $\Rightarrow$ programación paralela
- GPUs: ¿Cómo programarlas de forma fácil y eficiente?
- El problema surge tanto para una como para múltiples GPUs

Trabajo previo y aportaciones

- Implementaciones existentes para sistemas multi-GPU:
  - Distribución de datos entre GPUs $+$ código estilo ScaLAPACK
  - Planteamientos similares a las arquitecturas de memoria distribuida
- Independencia entre código y arquitectura
- Ejecución paralela $\longrightarrow$ *runtime*
  - Nueva arquitectura $\longrightarrow$ adaptación del runtime
  - Ideas ya contrastadas para *multicore* pueden ser utilizadas en el *runtime*

FLAME

- FLAME: **F**ormal **L**inear **A**lgebra **M**ethods **E**nvironment
- Notación para expresar algoritmos de álgebra lineal densa y banda
- Metodología para derivar familias de algoritmos (variantes)
- APIs: FLAME/C, FLAME@lab, FLAG@lab...
- Sencilla transformación notación $\rightarrow$ código
- FLASH API: extiende FLAME para algoritmos por bloques

FLAME: de algoritmo a código

Partition $A \rightarrow \left(\begin{array}{c|c} A_{TL} & A_{TR} \\ \hline A_{BL} & A_{BR} \end{array}\right)$

where A_{TL} is 0×0

while $m(A_{TL}) < m(A)$ **do**

Determine block size n_b

Repartition

$$\left(\begin{array}{c|c} A_{TL} & A_{TR} \\ \hline A_{BL} & A_{BR} \end{array}\right) \rightarrow \left(\begin{array}{c|c|c} A_{00} & A_{01} & A_{02} \\ \hline A_{10} & A_{11} & A_{12} \\ \hline A_{20} & A_{21} & A_{22} \end{array}\right)$$

where A_{11} is $n_b \times n_b$

$$A_{11} := \text{CHOL\_UNB}(A_{11})$$
$$A_{21} := A_{21}\,\text{TRIL}\,(A_{11})^{-\text{T}}$$
$$A_{22} := A_{22} - A_{21}A_{21}^{\text{T}}$$

Continue with

$$\left(\begin{array}{c|c} A_{TL} & A_{TR} \\ \hline A_{BL} & A_{BR} \end{array}\right) \leftarrow \left(\begin{array}{c|c|c} A_{00} & A_{01} & A_{02} \\ \hline A_{10} & A_{11} & A_{12} \\ \hline A_{20} & A_{21} & A_{22} \end{array}\right)$$

endwhile

```
FLA_Error FLASH_Chol_by_blocks_var1( FLA_Obj A ) {
  FLA_Obj ATL, ATR,       A00, A01, A02,
          ABL, ABR,       A10, A11, A12,
                          A20, A21, A22;

  FLA_Part_2x2( A, &ATL, &ATR,
                   &ABL, &ABR, 0, 0, FLA_TL );

  while ( FLA_Obj_length( ATL ) < FLA_Obj_length( A ) ) {
    FLA_Repart_2x2_to_3x3(
      ATL, ATR,       &A00, &A01, &A02,
                      &A10, &A11, &A12,
      ABL, ABR,       &A20, &A21, &A22,
      1, 1, FLA_BR );
    /*-------------------------------------------------------*/
    FLA_Chol_unb_var1( FLASH_MATRIX_AT( A11 ) );
    FLASH_Trsm( FLA_RIGHT, FLA_LOWER_TRIANGULAR,
                FLA_TRANSPOSE, FLA_NONUNIT_DIAG,
                FLA_ONE, A11, A21 );
    FLASH_Syrk( FLA_LOWER_TRIANGULAR, FLA_NO_TRANSPOSE,
                FLA_MINUS_ONE, A21,
                FLA_ONE,       A22 );
    /*-------------------------------------------------------*/
    FLA_Cont_with_3x3_to_2x2(
      &ATL, &ATR,       A00, A01, A02,
                        A10, A11, A12,
      &ABL, &ABR,       A20, A21, A22, FLA_TL );
  }
}
```

El *runtime SuperMatrix*

- Extracción de paralelismo a alto nivel
- Descomposición de operaciones en tareas
- Identificación de dependencias entre tareas
- Planificación de ejecución cuando las tareas están preparadas (dependencias satisfechas)
- Mapeado de tareas sobre **núcleos/aceleradores**
- Oculta detalles de paralelización al programador

FLAME en plataformas con múltiples aceleradores

Trabajo previo

- Cada Nvidia G80/IBM Cell B.E. es visto como una arquitectura multicore

- Cada acelerador es visto como un único core
- Existe una implementación optimizada de BLAS disponible
- Analogía multicore $\Longleftrightarrow$ multiacelerador

El paralelismo se extrae a dos niveles:

1. Alto nivel: *SuperMatrix*
2. Bajo nivel: BLAS

Modelo de ejecución

- Estación de trabajo (multicore) + múltiples aceleradores *hardware*
- Aceleradores: elementos pasivos a la espera de trabajo
- Independencia entre RAM y memoria del acelerador (sin mecanismos de coherencia)
- Comunicación CPU-acelerador: copias entre memorias, a través de RAM
- Ejemplo: Nvidia Tesla s870

SuperMatrix: funcionamiento

Ejecución del algoritmo por bloques en dos fases (tiempo de ejecución):

1. **Fase de análisis**
   - Un *thread* "ejecuta" el código
   - No se realizan operaciones inmediatamente → lista de tareas pendientes
   - Identificación de dependencias entre tareas → D.A.G.
2. **Fase de lanzamiento**
   - Los *threads* ociosos inspeccionan la cola de tareas pendientes
   - Al encontrar una lista para ejecución, se computa en un acelerador
   - Actualización de la información de dependencias

- **Fase de análisis**: no se modifica
- **Fase de lanzamiento**: posibilidad de adaptación al entorno específico (i.e. Tesla)

Implementación básica

- Creación de tantos *threads* en el host como aceleradores
- Detección de tarea lista:
  1. Copia de datos asociados a memoria del acelerador
  2. Orden de ejecución de la operación BLAS correspondiente
  3. Transferencia de datos a RAM
- Ejecución híbrida: la CPU **planifica**, el acelerador **ejecuta**
- Ciertas tareas pueden ser ejecutadas en CPU si es conveniente
- **Problema**: excesivas transferencias entre memorias

Mejorando el rendimiento de *SuperMatrix*

Cachés y coherencia de memoria

- Políticas de coherencia de memoria:
  1. *Write-through*: escrituras inmediatamente reflejadas en memoria principal
  2. *Write-back*: datos actualizados únicamente cuando la línea de cache es reemplazada
- En sistemas con memoria compartida:
  1. *Write-update*: escrituras por uno de los procesadores propagadas inmediatamente a las copias en caches del resto de procesadores
  2. *Write-invalidate*: escrituras por uno de los procesadores invalidan las copias en las caches del resto de procesadores
- Objetivo: reducción de transferencias

Mejorando el rendimiento de *SuperMatrix*

Analogías:

Multiprocesador con MC $\Longleftrightarrow$ Sistema multiacelerador

Acelerador $\Longleftrightarrow$ Procesador

Memoria del acelerador $\Longleftrightarrow$ Caches

RAM $\Longleftrightarrow$ Memoria compartida por los procesadores (aceleradores)

Aplicación de las políticas de coherencia conocidas a nuestro sistema multiacelerador

2-D + write-through

- Distribución de datos: mapeado 2-D cíclico
- Operaciones que escriben sobre un bloque dado se asignan a la misma GPU
- Bloques propietarios y no propietarios
- Funcionamiento:
  - Inicialmente todos los bloques residen en RAM
  - Tarea asignada a procesador $\rightarrow$ los bloques se copian a GPU (si no residen ya allí)
  - Bloques propietarios: se mantienen en memoria de GPU hasta el final de la ejecución
  - Bloques no propietarios: se descartan al completar la operación
  - Política *write-through* para bloques propietarios: propagación inmediata de datos escritos a RAM

Cache + write-invalidate

- **Problema**: gran cantidad de transferencias de bloques no propietarios
- Cache *software* para bloques de sólo lectura (no propietarios)
- Se mantienen en memoria de GPU los bloques más recientemente usados
- Operaciones que escriben sobre un bloque dado se asignan a la misma GPU
- El *thread* correspondiente en CPU invalida las copias en el resto de GPUs
- Políticas de reemplazo y número de bloques de caché por acelerador pueden ser modificados

Write-back

- **Objetivo**: reducción de transferencias a RAM para bloques propietarios
- Política *write-back*: inconsistencias entre GPU y RAM
- Actualización de bloques propietarios en RAM: sólo si otra GPU los necesita
- Finalización del algoritmo: actualización de datos a RAM

Optimización de rutinas BLAS

1. *Sgemm* optimizado: CUBLAS 2.0
2. *Ssyrk* optimizado: basado en *sgemm* de CUBLAS 2.0
3. *Strsm* optimizado: inversión de la matriz en CPU + producto matriz-matriz en GPU

Resultados experimentales

Nvidia Tesla s870

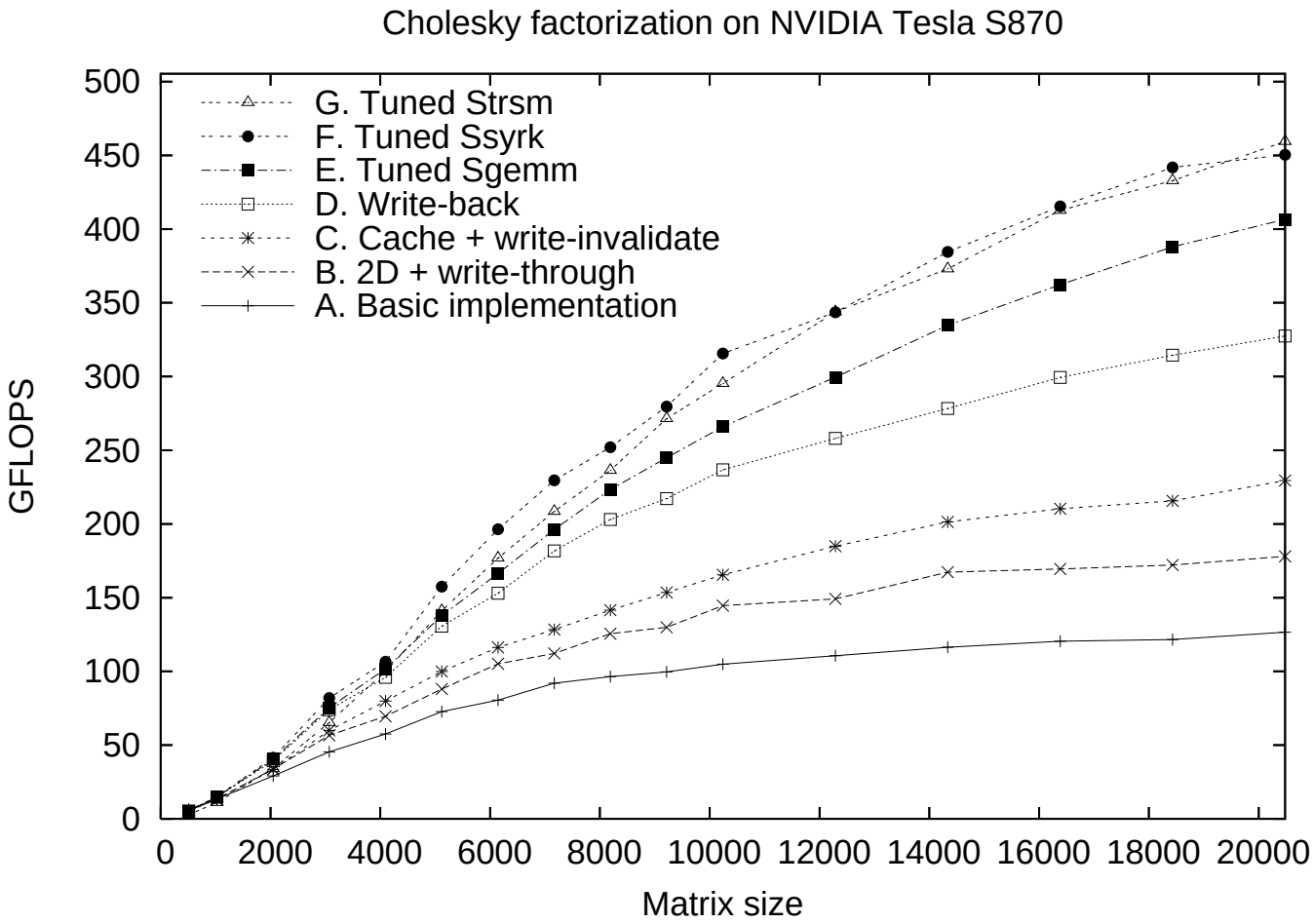
- 4 GPUs Nvidia Quadro 5600
- Interfaz doble PCIExpress Gen2
- 1.5 Gbytes RAM por GPU
- Host: Dual Intel Xeon QuadCore 2.00 Ghz, 8 Gb RAM

Resultados experimentales

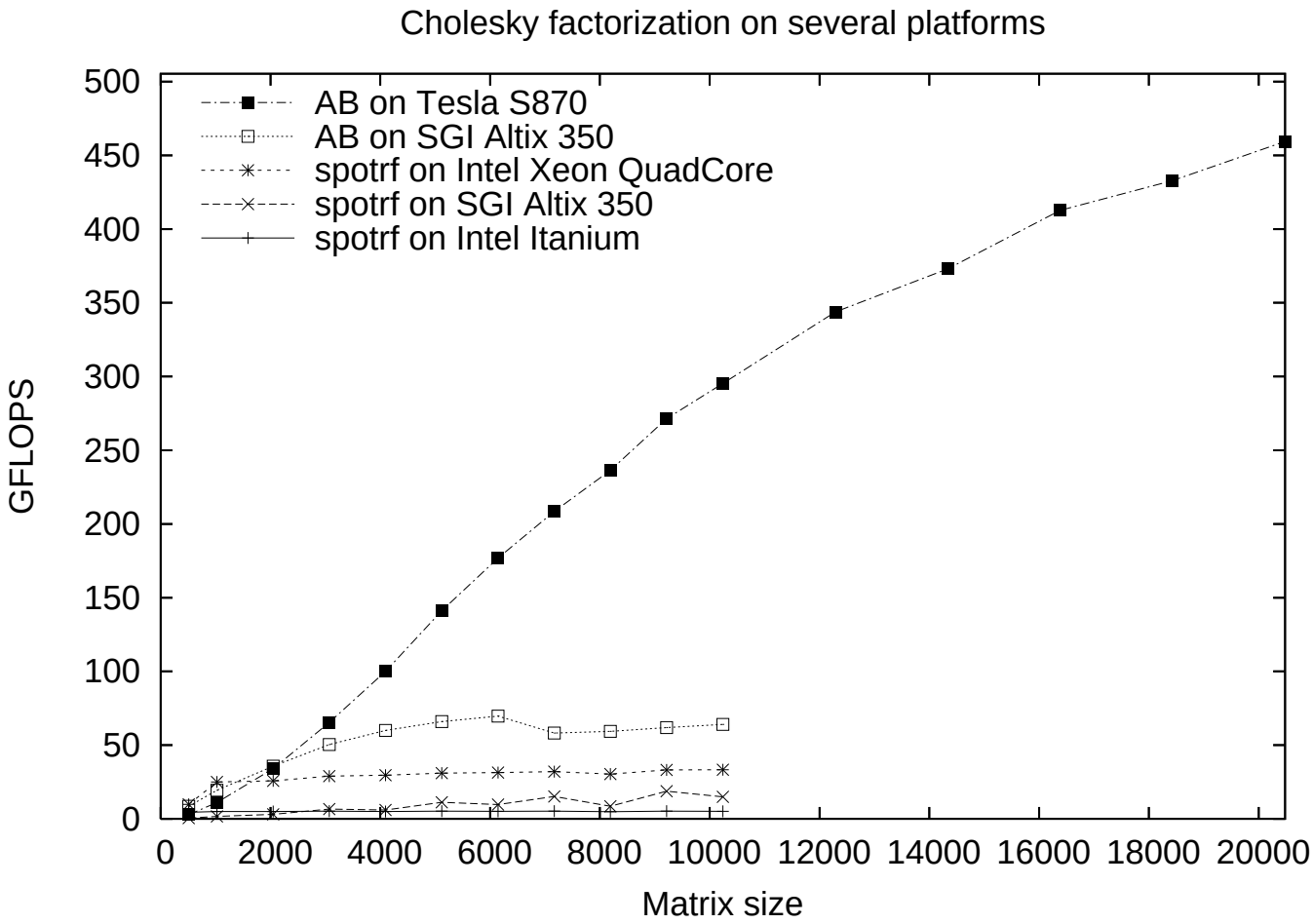
Evaluación de diversos sistemas

- `spotrf` en Intel Itanium: MKL 8.1 en Intel Itanium2 a 1.5 Ghz
- `spotrf` en SGI Altix 350: MKL 8.1 multihilo en 16 Intel Itanium2 a 1.5 Ghz (ccNUMA)
- `spotrf` en Intel Xeon QuadCore: MKL 10 multihilo en Intel Xeon QuadCore a 2 Ghz
- AB en SGI Altix 350: nuestro algoritmo por bloques en SGI Altix 350
- AB en Tesla s870: nuestro algoritmo por bloques (mejor variante) en una máquina Nvidia Tesla s870 (4 GPUs)

Resultados experimentales



Resultados experimentales



Conclusiones

- *SuperMatrix*: solución flexible para multicore y **multiaceleradores**
- Posibilidad de aplicación de técnicas ya contrastadas
- Transparencia para el programador
- *Runtime* configurable y fácilmente ampliable
- No sólo aplicable a sistemas multiGPU $\Rightarrow$ IBM Cell B.E.