# High Performance Matrix Inversion on a Multi-core Platform with Several GPUs

Pablo Ezzatti[1], Enrique S. Quintana-Ortí[2] and Alfredo Remón[2]

[1]Centro de Cálculo-Instituto de Computación, Univ. de la República
pezzatti@fing.edu.uy
[2]Depto. de Ing. y Ciencia de los Computadores, Univ. Jaume I de Castellón
{quintana,remon}@icc.uji.es

Ayia Napa – February 2011

## Matrix inversion of large-scale matrices

- Appears in a number of scientific applications like model reduction or optimal control
- Requires a high computational effort, $2n^3$ floating-point operations (flops)

## Graphics processors (GPUs)

- Massively parallel architectures
- Good results on the acceleration of linear algebra operations

# Outline

# Matrix inversion

## Two different approaches

1. Based on the Gaussian elimination (i.e., the LU factorization)
2. Based on the Gauss-Jordan elimination (GJE)

Both approaches present similar computational cost
but different properties

# Matrix inversion via LU factorization

1. $PA = LU$
2. $U \rightarrow U^{-1}$
3. Solve the system $XL = U^{-1}$ for $X$
4. Undo the permutations $A^{-1} := XP$

## Implementation

- The algorithm sweeps through the matrix four times.
- Presents a mild load imbalance, due to the work with triangular factors.

Algorithm implemented by LAPACK

# Matrix inversion via GJE

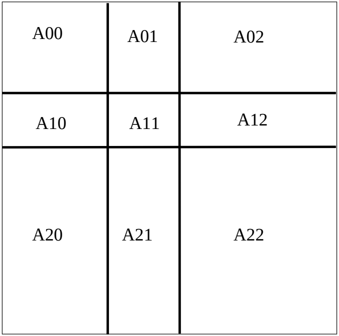
## Based on Gauss-Jordan elimination

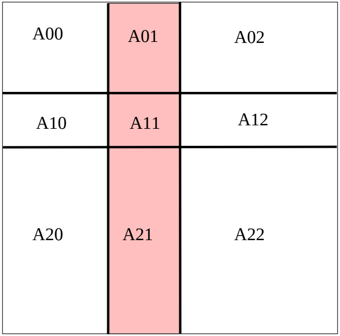
- In essence, it is a reordering of the operations
- Same arithmetical cost

## Implementation

- The algorithm sweeps through the matrix once
  $\longrightarrow$ Less memory accesses
- Most of the computations are highly parallel
  $\longrightarrow$ More parallelism

At a given iteration:

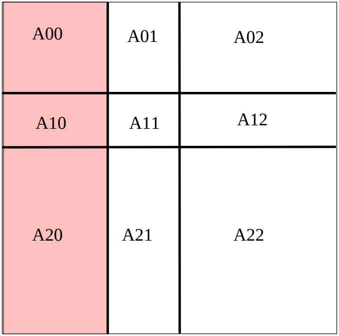


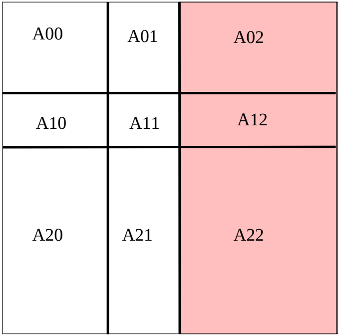




$$\begin{bmatrix} A_{01} \\ A_{11} \\ A_{21} \end{bmatrix} := GJE_{\mathrm{UNB}}\left(\begin{bmatrix} A_{01} \\ A_{11} \\ A_{21} \end{bmatrix}\right)$$

$$A_{11} \text{ is } b \times b$$

# Matrix inversion via GJE




$$A_{00} := A_{00} + A_{01} A_{10}$$

$$A_{20} := A_{20} + A_{21} A_{10}$$

$$A_{10} := A_{11} A_{10}$$

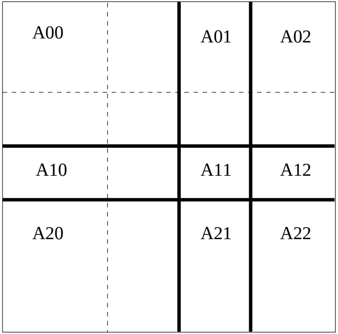
$$A_{02} := A_{02} + A_{01} A_{12}$$

$$A_{22} := A_{22} + A_{21} A_{12}$$

$$A_{12} := A_{11} A_{12}$$

# Matrix inversion via GJE

Move boundaries for the next iteration




$$A_{11} \text{ is } b \times b$$

# Outline

1. Motivation
2. Matrix inversion
3. Implementations on a multi-core CPU and multiple CPUs
4. Experimental analysis
5. Concluding remarks

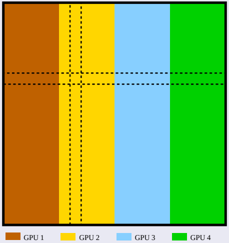
# Implementations on a multi-core CPU and multiple GPUs

## Implementation $GJE_{mGPU}$

- Executes every operation on the most convenient device:
  - $GJE_{UNB}$ on the CPU
  - Matrix-matrix products on the GPUs

## Data distribution

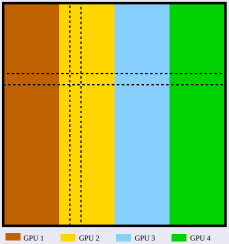
- Data is uniformly distributed among GPUs
- Each GPU performs the operations that involve their respective panel



# Implementations on a multi-core CPU and multiple GPUs

## Implementation $GJE_{mGPU}$

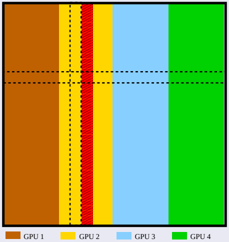
- All GPUs compute concurrently
- The update of current panel on the CPU is overlapped with some updates on the GPUs
- The active column panel is transferred to the CPU initially, and broadcasted to the GPUs after being updated



# Implementations on a multi-core CPU and multiple GPUs

## Implementation $\text{GJE}_{LA}$

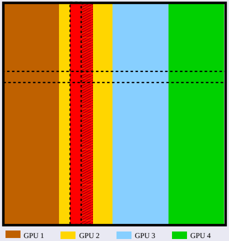
- Based on $\text{GJE}_{mGPU}$ but reordering the operations applying a look-ahead approach
- The first $b$ columns of block $[A_{02}; A_{12}; A_{22}]$ are updated and transferred to the CPU



# Implementations on a multi-core CPU and multiple GPUs

## Implementation $GJE_{LA}$

- The CPU updates $[A_{01}; A_{11}; A_{21}]$ and the new received block (that is, blocks $[A_{01}; A_{11}; A_{21}]$ of the next iteration)
- Concurrently, the GPUs update the rest of the matrix.



# Implementations on a multi-core CPU and multiple GPUs

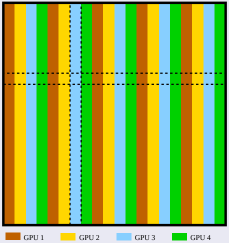
## Implementation $GJE_{ML}$

- The optimal algorithmic block-size for CPU and GPU are significantly different
- The use of the same block-size in both architectures limits the performance of $GJE_{LA}$
- In this version, the CPU employs a blocked version of GJE instead of the unblocked one
- The CPU and GPUs employ block-sizes $b$ and $b_c$ respectively, allowing each architecture to optimize its performance

# Implementations on a multi-core CPU and multiple GPUs

## Implementation $GJE_{CD}$

- At any iteration, one of the GPUs require more time than the rest, leading to load imbalance
- We can partially overcome this problem by employing a cyclic distribution
- The matrix is partitioned in blocks of $b$ columns, and the $i$th block of columns is stored and updated by the $(i \; mod \; k)$-th GPU

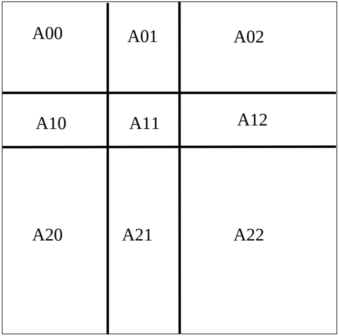


# Implementations on a multi-core CPU and multiple GPUs

## Implementation $GJE_{Merge}$

- All GPUs perform 3 operations per iteration
- Performing a minor change on our algorithm, we can reduce it to one operation per iteration with the following advantages:
  - Less overhead due to routines invokations
  - Avoid the matrix-matrix products that involve small blocks

At a given iteration:



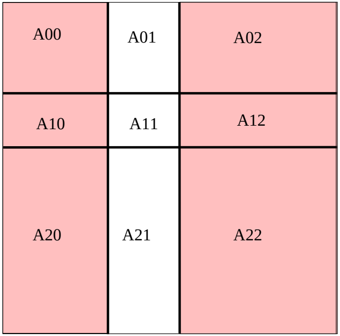


$A_{11}$ *is* $b \times b$

$$\begin{bmatrix} A_{01} \\ A_{11} \\ A_{21} \end{bmatrix} := GJE_{\mathrm{BLK}}\left(\begin{bmatrix} A_{01} \\ A_{11} \\ A_{21} \end{bmatrix}\right)$$

# Implementations on a multi-core CPU and multiple GPUs

## Implementation $GJE_{Merge}$



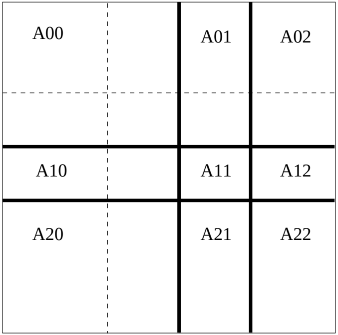

$$W_1 := A_{10}$$
$$A_{10} := 0$$
$$W_2 := A_{12}$$
$$A_{12} := 0$$

$$\begin{bmatrix} A_{00} A_{02} \\ A_{10} A_{12} \\ A_{20} A_{22} \end{bmatrix} := \begin{bmatrix} A_{00} A_{02} \\ A_{10} A_{12} \\ A_{20} A_{22} \end{bmatrix} + \begin{bmatrix} A_{01} \\ A_{11} \\ A_{21} \end{bmatrix} [W_1 W_2]$$

# Implementations on a multi-core CPU and multiple GPUs

## Implementation $GJE_{Merge}$

Move boundaries for the next iteration




$$A_{11} \text{ is } b \times b$$

# Implementations on a multi-core CPU and multiple GPUs

## Implementation $\text{GJE}_{Merge}$

- An important improvement in performance can be obtained by merging the extra copies with the swap stage required by pivoting.
  - Thus, $W_1$ and $W_2$ will contain blocks $A_{10}$ and $A_{12}$ after the pivoting has been applied.
  - This considerably reduces the number of memory accesses and partially hides the overhead introduced by the copies of $A_{10}$ and $A_{12}$

# Outline

# Experimental analysis

| Processors | # Cores | Frequency | L2 cache (MB) | Memory (GB) |
|---|---|---|---|---|
| Intel Xeon QuadCore | 8 (2x4) | 2.27 | 8 | 48 |
| NVIDIA TESLA c1060 | 960 (4x240) | 1.3 | - | 16(4x4) |

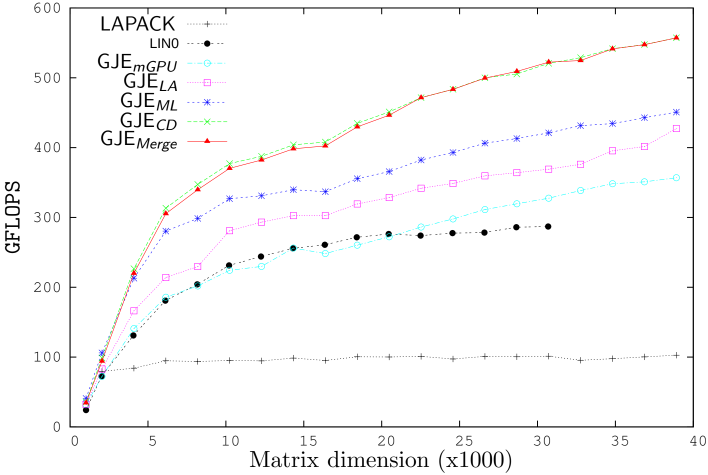
## BLAS Implementations

- MKL 11.1
- NVIDIA CUBLAS 3.0

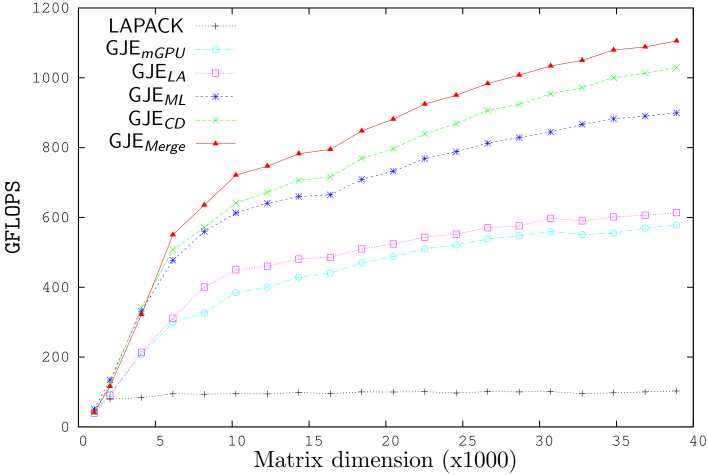
Results for matrices with $1000 \leq n \leq 64000$

# Experimental analysis

## (Martix inversion on 2 GPUs)

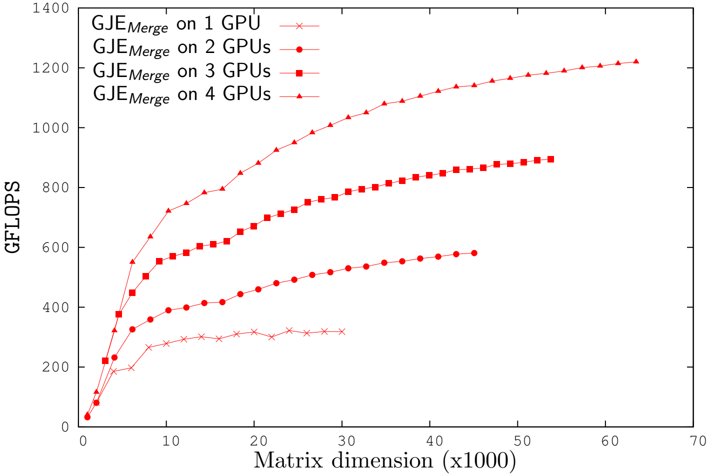


# Experimental analysis
## (Martix inversion on 4 GPUs)



# Experimental analysis

## (Variant $GJE_{Merge}$ on 1, 2, 3 and 4 GPUs)



# Outline

# Concluding Remarks

- We have presented five implementations of matrix inversion based on the GJE method on multiple GPUs
- Those implementations are between 6 and 12 times faster than LAPACK using 4 GPUs and exhibit excellent scalability properties (with a nearly linear speed-up)
- The use of multiple GPUs
  - Reduce the computational time
  - Increments the amount of memory available, allowing the inversion of larger matrices

# Future Work

- Evaluation of double precision arithmetic on the new NVIDIA Fermi architecture
- $GJE_{Merge}$ implementation is clearly limited by the performance of CUBLAS routine for matrix-matrix products. Other GPU Kernels should be evaluated

Questions ?