J Supercomput (2011) 58:215–225
DOI 10.1007/s11227-009-0360-z

Neville elimination on multi- and many-core systems: OpenMP, MPI and CUDA

P. Alonso · R. Cortina · F.J. Martínez-Zaldívar · J. Ranilla

Published online: 19 November 2009

Abstract This paper describes several parallel algorithmic variations of the Neville elimination. This elimination solves a system of linear equations making zeros in a matrix column by adding to each row an adequate multiple of the preceding one. The parallel algorithms are run and compared on different multi- and many-core platforms using parallel programming techniques as MPI, OpenMP and CUDA.

Keywords Neville · Multi-core · Many-core · OpenMP · MPI · GPU · CUDA · CUBLAS

1 Introduction

A multi-core processor combines two or more independent cores or CPUs in a single die or integrated circuit, sharing the external bus interface and some cache memory [1]. This combination usually represents the most extreme form of tightly-coupled multiprocessing. Many current multi-core processors have two, four, eight or even more cores. When the number of cores is higher (several hundreds), the architecture is named many-core processor (e.g. GPUs) [2]. These systems are specially

P. Alonso
Departamento de Matemáticas, Universidad de Oviedo, Oviedo, Spain
e-mail: palonso@uniovi.es

R. Cortina · J. Ranilla (✉)
Departamento de Informática, Universidad de Oviedo, Oviedo, Spain
e-mail: ranilla@uniovi.es

R. Cortina
e-mail: raquel@uniovi.es

F.J. Martínez-Zaldívar
Departamento de Comunicaciones, Universidad Politécnica de Valencia, Valencia, Spain
e-mail: fjmartin@dcom.upv.es

suited in applications where there are at least as many running processes or independent process threads as cores. The shared memory programming paradigm can be considered the natural way of programming these systems. Other paradigms, like message passing, can be easily implemented using the shared memory communication mechanisms found in some of these kinds of multiprocessors.

Neville elimination is an alternative procedure to that of Gauss to transform a square matrix A into an upper-triangular one. Strictly speaking, Neville elimination makes zeros in an A column adding to each row a multiple of the previous one. It is a better alternative to Gaussian elimination when working with totally positive matrices, sign-regular matrices or other related types of matrices (see [3] and [4]). According to [5] and [6], Neville elimination is considered to be an interesting alternative to Gauss elimination in certain types of research. Furthermore, there are other works (see [7–9]) that show the advantages of the aforesaid procedure in the field of High Performance Computing. The mentioned works have been performed considering distributed memory environments.

This paper compares the performance of several parallel implementations of the Neville algorithm devised to solve a system of linear equations on multi- and many-core architectures. We have used implementations of the MPI and OpenMP standards in order to program parallel Neville algorithms with distributed and shared memory schemes and the CUBLAS library written in CUDA for GPUs.

This paper is organized as follows: first we will show the sequential Neville algorithm; next we will describe two variations of the shared memory parallel algorithm; later some tests will show the performance of these algorithmic variations in several multi- and many-core platforms, and finally the conclusions of this work will be stated.

2 Neville algorithm

A system of equations $Ax = b$ is usually solved in two stages ($A = LU$). First, through a series of algebraic manipulations the original system of equations is reduced to an upper-triangular system $Ux = y$. In the second stage, the upper-triangular system is solved by a procedure known as back-substitution.

If A is a square matrix of order n, the Neville elimination procedure consists of $n-1$ successive major steps (see [4] for a detailed and formal introduction), resulting in a sequence of matrices as follows: $A = A^{(1)} \longrightarrow A^{(2)} \longrightarrow \cdots \longrightarrow A^{(n)} = U$, where U is an upper-triangular matrix. If A is non-singular, the matrix $A^{(k)} = (a_{i,j}^{(k)})_{1\le i,j\le n}$ has zeros below its main diagonal in the $k-1$ first columns.

Let us consider the case in which Neville elimination can be performed without changing rows. The work presented in [3] shows that row changes are not necessary when the Neville elimination process is applied to a totally positive matrix (a matrix whose minors are non-negative). In order to get $A^{(k+1)}$ from $A^{(k)}$, zeros are obtained in the kth column below the main diagonal, subtracting a multiple of the ith row from the $(i+1)$th for $i = n-1, n-2, ..., k$, according to the formula:

$$a_{i,j}^{(k+1)} = \begin{cases} a_{i,j}^{(k)} & \text{if } 1 \le i \le k, \\ a_{i,j}^{(k)} - \frac{a_{i,k}^{(k)}}{a_{i-1,k}^{(k)}} a_{i-1,j}^{(k)} & \text{if } k+1 \le i \le n \text{ and } a_{i-1,k}^{(k)} \neq 0, \\ a_{i,j}^{(k)} & \text{if } k+1 \le i \le n \text{ and } a_{i-1,k}^{(k)} = 0. \end{cases} \quad (1)$$

Algorithm 1 shows the Neville iterations to solve a non-singular system of linear equations. $A(i, j)$ denotes the (i, j)-element of the matrix A. When all the iterations finish, A becomes upper-triangular and x can be obtained via back-substitution from A and b. The information of the method steps can be stored in the matrix.

The inner loop represents a typical `axpy` operation ($y = ax + y$), in which a portion of the ith row is updated with a linear combination of it and the same portion of the $(i-1)$th row; and a is the ratio $-A(i, j)/A(i-1, j)$. The cost of this algorithm is $2/3n^3$ flops.

Figure 1 shows the range of the indices and the rows and elements involved in one inner iteration.

Algorithm 1 Sequential Neville

```
Require: $\mathbf{A} \in \mathbb{R}^{n\times n}$ and $\mathbf{b} \in \mathbb{R}^n$ such that $\mathbf{Ax} = \mathbf{b}$
Ensure: processed $\mathbf{A}$ and $\mathbf{b}$ with $\mathbf{A}$ upper-triangular and $\mathbf{x} = \mathbf{A}^{-1}\mathbf{b}$
  for $j = 1 : n-1$ do
    for $i = n : -1 : j+1$ do
      $\alpha = \mathbf{A}(i, j)/\mathbf{A}(i-1, j)$
      for $r = j+1 : n$ do
        $\mathbf{A}(i, r) = \mathbf{A}(i, r) - \alpha\mathbf{A}(i-1, r)$
      end for
      $\mathbf{b}(i) = \mathbf{b}(i) - \alpha\mathbf{b}(i-1)$
      $\mathbf{A}(i, j) = 0$
    end for
  end for
```

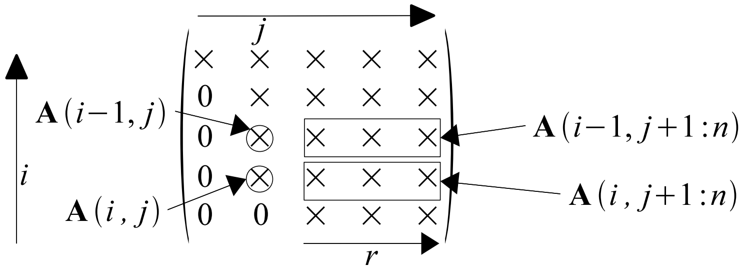

Fig. 1 Sequential Neville algorithm

3 Multi-core parallel algorithms

In previous works (see [7–9]) we have proposed an organization of the Neville elimination algorithm for computers with the message passing model and we have carried out a general analysis based on upper bounds for the execution time, efficiency/speedup and scalability metrics.

The realized work has allowed us to confirm the good performance of Neville's method using parallel computation on single-core systems. For example, in [10] we have verified that using a column-wise cyclic-striped distribution of the data (size of block 1), the reached efficiency is near to one.

Next we describe two parallel algorithmic variations for multi-core platforms. The parallel algorithms are run and compared using two parallel programming techniques: MPI and OpenMP.

3.1 Blocks of contiguous rows

The dependency graph of the parallel algorithm would show that the result in the ith row depends only on it and on the $(i-1)$th row for the jth iteration. Hence, we can group the rows in blocks of consecutive rows and apply the `axpy` operation sequentially in the block. The (not yet updated) row of highest index in a block will be needed (without updating) by the lowest index row of the next block, hence it should be saved in a buffer before its updating. It will be necessary to synchronize the threads in order to avoid race conditions in the last row of the blocks. The block workload may be assigned to a thread.

The configuration of every block of contiguous rows may be static or dynamic. If we use the static configuration, there will be load unbalancing because the workload associated to the first rows will finish earlier (as the j index increases, the number of rows involved in the computation decreases, as can be observed in the Neville Algorithm 1), hence the associated threads will be idle. One way to avoid this load unbalancing is to reconfigure the blocks of contiguous rows every j-iteration, dividing the number of the active rows among the threads.

Hence, this first version can be derived directly from the sequential version (with the proper additional controls over the buffer of the not yet updated row, and the row indices associated to the thread).

Figure 2 shows the division of the workload in form of blocks of consecutive rows, the dependency of the first row of a block with the last row of the upper block, and the way the matrix lower triangle is symbolically zeroed.

3.2 Blocks of contiguous columns

The updating of the rows may be done on a column basis beginning at the bottom and finishing at the top of every column in an independent way. It can be easily accomplished using the Algorithm 2.

The columns ranged by every private copy of the j index (columns processed by a thread) depend on the optional scheduling parameter of the `#pragma` directive, [11]. Figure 3 shows the independence among the columns ranged by the j index.

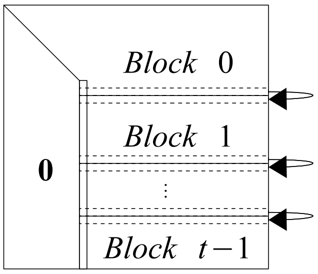

Fig. 2 Parallel Neville algorithm with block of contiguous rows

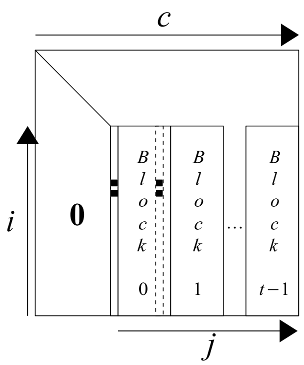

Fig. 3 Parallel Neville algorithm with block of contiguous columns

Algorithm 2 Parallel Neville algorithm with block of contiguous columns

```
Require: $\mathbf{A} \in \mathbb{R}^{n\times n}$ and $\mathbf{b} \in \mathbb{R}^n$ such that $\mathbf{Ax} = \mathbf{b}$
Ensure: processed $\mathbf{A}$ and $\mathbf{b}$ with $\mathbf{A}$ upper-triangular and $\mathbf{x} = \mathbf{A}^{-1}\mathbf{b}$
  for $c = 0 : N - 2$ do
    #pragma omp parallel for private(i,j,alfa)
    for $j = c + 1 : N - 1$ do
      for $i = N - 1 : -1 : c + 1$ do
        $\alpha = \mathbf{A}(i, c)/\mathbf{A}(i - 1, c)$
        $\mathbf{A}(i, j) = \mathbf{A}(i, j) - \alpha\mathbf{A}(i - 1, j)$
        if $(j == N - 1)$ then
          $\mathbf{b}(i) = \mathbf{b}(i) - \alpha\mathbf{b}(i - 1)$
        end if
      end for
    end for
  end for
```

4 Many-core parallel algorithm

The many-core parallel version is derived directly from the *block of contiguous rows* parallel version, using a block size of one row and without workload balancing. We have used calls to the CUBLAS library (CUBLAS-1 `cublas[D|S]axpy` subroutine) using single and double precision.

5 Experimental results

We have tested the parallel algorithms on the next platforms:

- HP ProLiant BL465 G5 dual processors AMD Opteron(TM) Quad 2356, 2.3 GHz, 512 kB cache, MPI: HP-MPI V2.2 (it complies fully with the MPI-1.2 standard and provides full MPI-2 functionality).
- DELL PowerEdge 2950 III dual processors Intel(R) Xeon(R) Quad E5420, 2.50 GHz, 6 MB cache, MPI: MPICH-1.2.7.p1.
- NVIDIA Tesla C1060 GPU running CUDA (2.1) inside a CPU with one Intel Core 2 Quad Q9550, 2.0 GHz, 6 MB cache.

5.1 OpenMP and MPI versions

To compare the behavior of OpenMP and MPI we use the efficiency metric obtained from the total execution wall-time of the sequential and parallel algorithms presented in this work. Afterwards, the implementation made in [10] for column-wise cyclic-striped distribution using MPI was tested. The obtained efficiencies are shown in Figs. 4, 5, 6, 7, 8, 9 (for clarity reasons, the executions with odd number of cores are not shown).

We can observe a superlinearity behavior in all the variants due to cache effect. In general, the OpenMP implementations do not get a high efficiency for high size problems because of the non-exploitation of locality. The row blocks based parallel algorithm needs to synchronize the threads to avoid the race condition over the shared

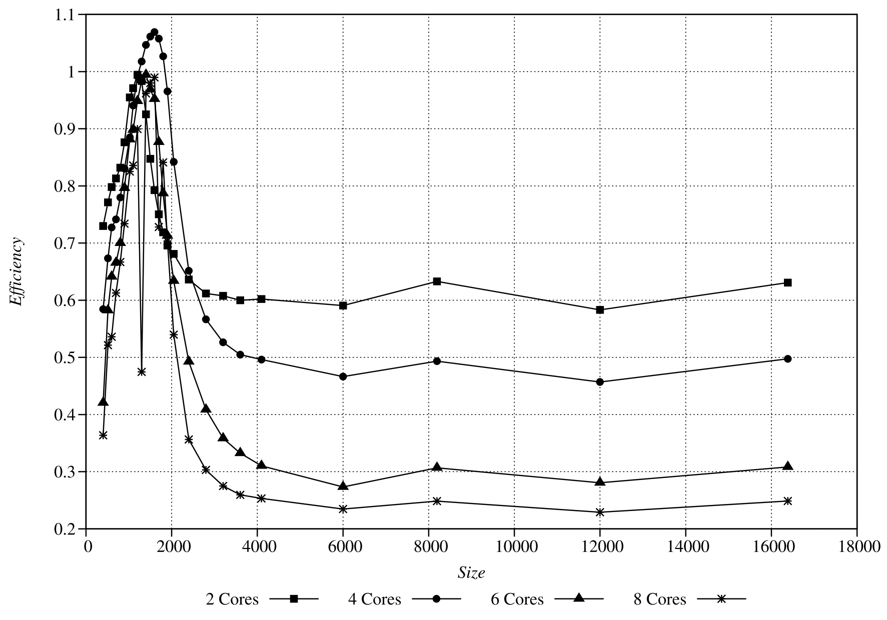

Fig. 4 Efficiency of the block of contiguous rows multi-core version in Dell

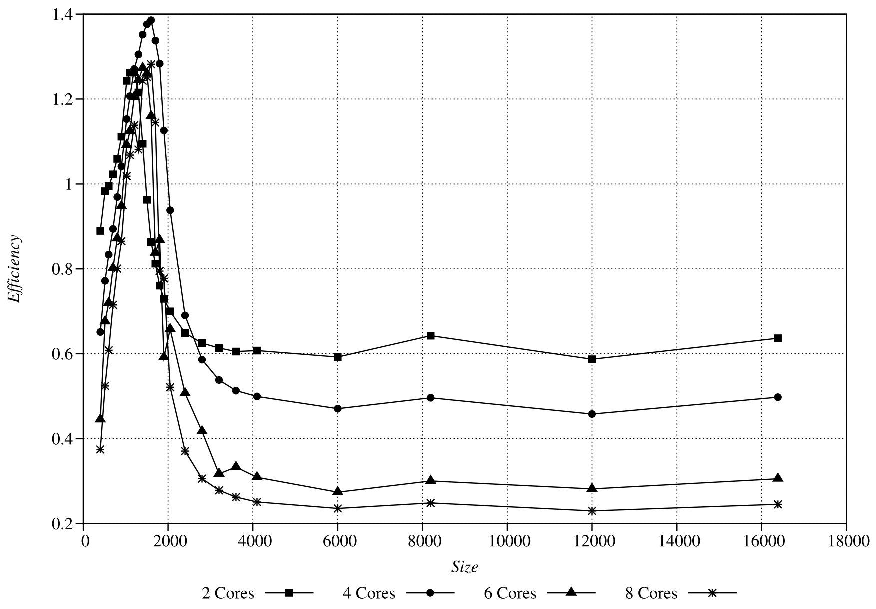

Fig. 5 Efficiency of the block of contiguous columns multi-core version in Dell

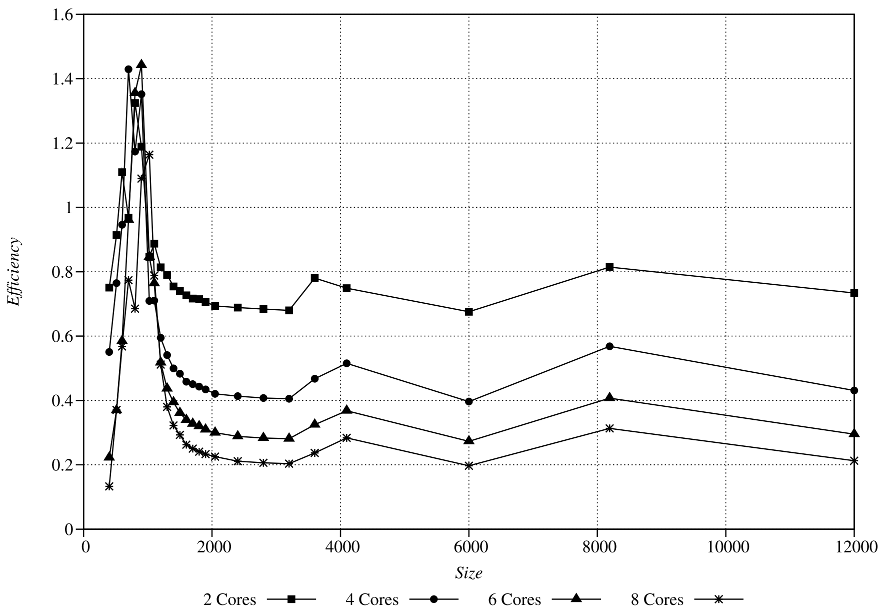

Fig. 6 Efficiency of the block of contiguous rows multi-core version in HP

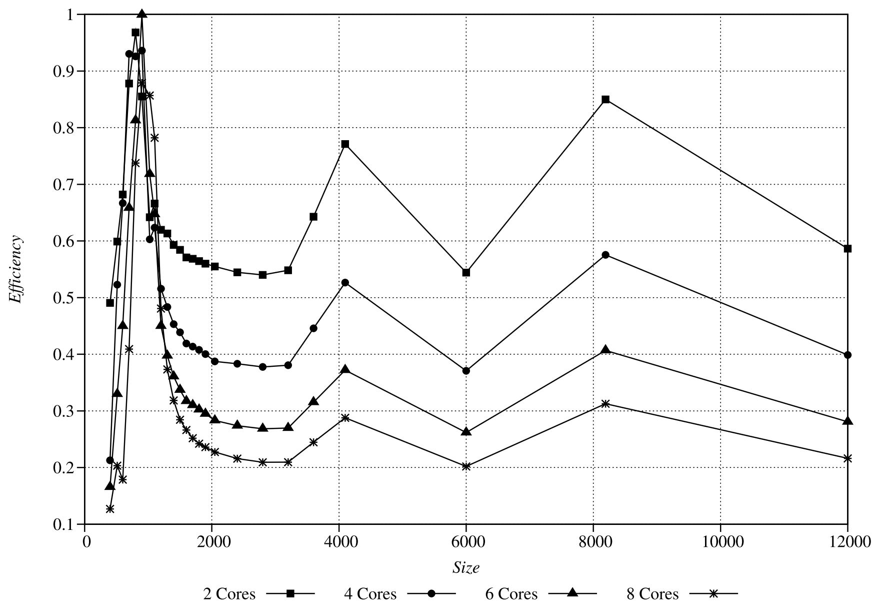

Fig. 7 Efficiency of the block of contiguous columns multi-core version in HP

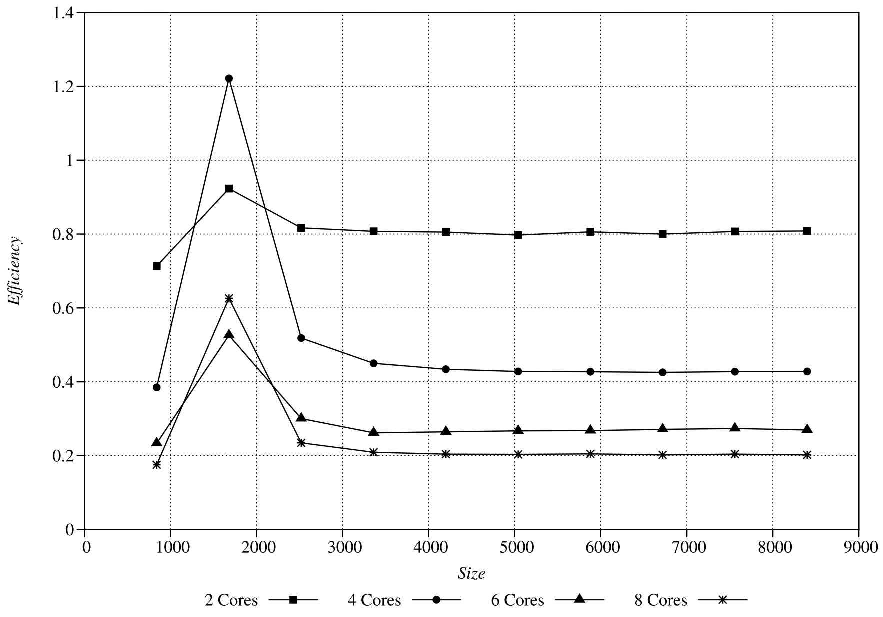

Fig. 8 Efficiency of the block of contiguous columns MPI version in Dell

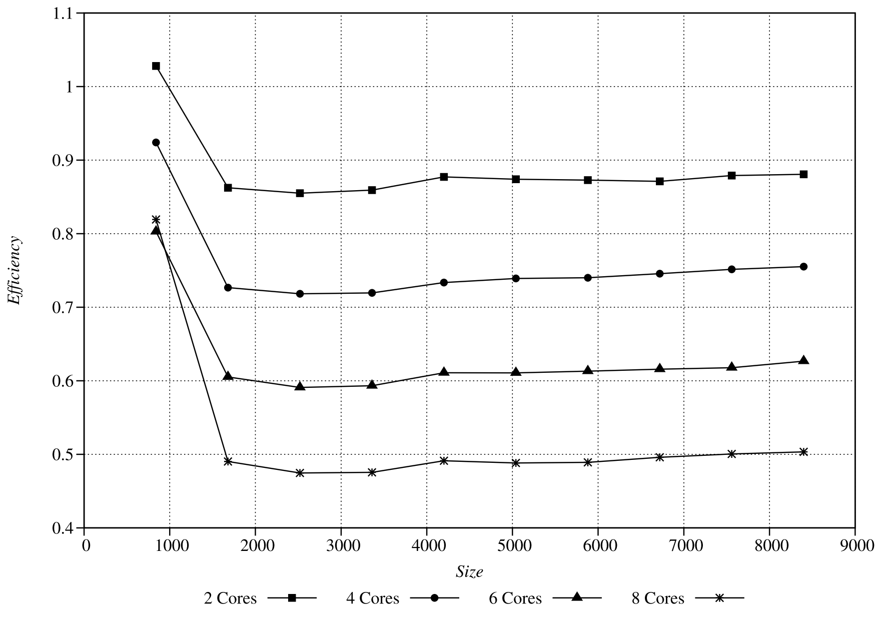

Fig. 9 Efficiency of the block of contiguous columns HP-MPI version in HP

buffer. Finally, the power of two problem sizes improves the efficiency (in a special way in the HP Proliant—this machine is a pure CC-NUMA).

The efficiency behaviors of the MPI versions are better and more stable (in fact, the HP-MPI library is highly optimized for HP servers). Here, the increment of the number of cores does not penalize so much the obtained efficiency.

5.2 CUBLAS/CUDA version

We have tested single and double precision variations due to the special architecture characteristics of the GPUs [2].

Figure 10 shows the execution time of the Neville algorithm, observing that the slowest result corresponds to the double precision single-core CPU execution and the fastest one to the single precision GPU execution, as expected. The speedup of the GPU with respect to the single-core CPU grows with the problem size n and is about 4 for high values of n.

Figure 11 shows the megaflops per second computed as the ratio between the expected cost or number of floating point operations ($2/3n^3$) and the required execution time. The single-core CPU megaflop/s results have a peak for relatively low values of n (the problem size threshold value is closely related to the cache size of the core) and then drops and remains constant with the problem size.

The GPU behavior is more regular and it grows with the problem size. The reason for this behavior is that the algorithm is poor in (CU)BLAS-3 or -2 level operations that get the best of these systems (only CUBLAS-1 level operations are effectively applicable: `axpy`).

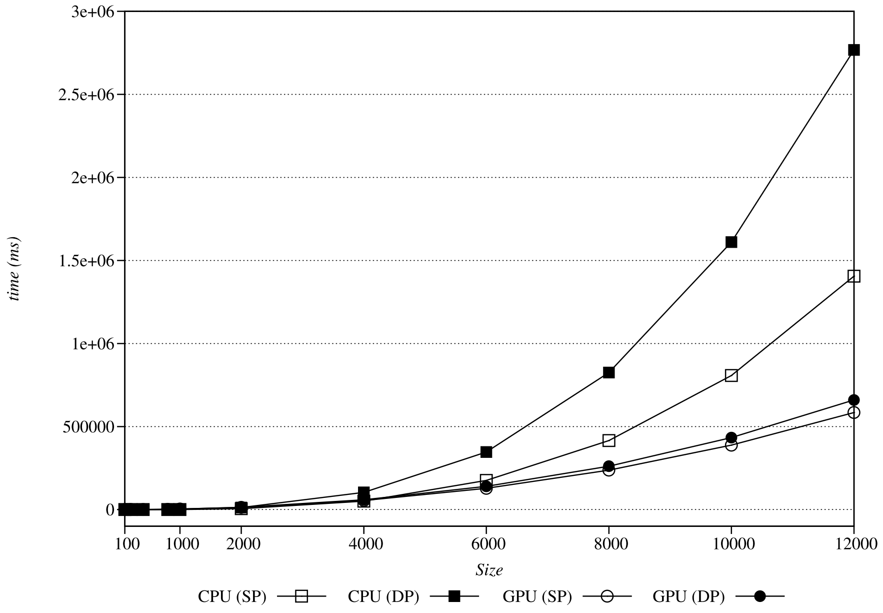

Fig. 10 Execution time of Neville in CPU and GPU with single (SP) and double precision (DP)

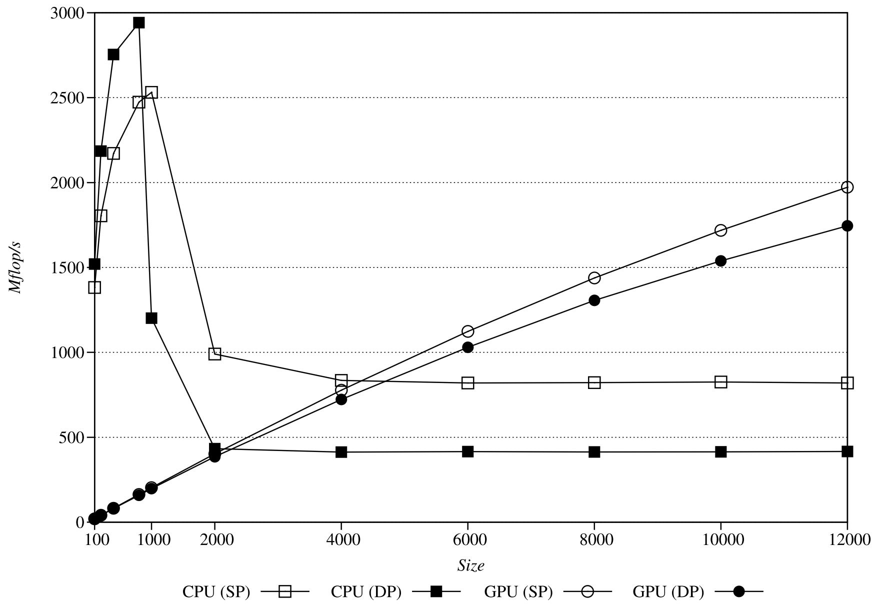

Fig. 11 Mflop/s of Neville in CPU and GPU with single (SP) and double precision (DP)

6 Conclusions

In general, the parallel efficiency is poorer in the OpenMP versions than in the MPI versions. The main reason is the difficulty to exploit the cache data locality in this algorithm. Also, the competition for the common resources increases as the number of cores increases, hence appearing bottlenecks that avoid a good efficiency. In OpenMP, only inside some problem size interval it is possible to obtain a good (superlinear) efficiency due to the cache effect over the data; eventually, it can be observed a better behavior for power of two sizes due to similar reasons. Obviously, there is an important performance difference between the HP-MPI implementation and the MPICH implementation of the MPI standard in favor of the former.

The CUBLAS/CUDA behavior is very good in respect to a single-core CPU execution with high problem sizes. The Neville algorithm is poor in high level (CU)BLAS operations, hence an even better performance may be obtained if an implementation of a specific CUDA *kernel* is used instead of the CUBLAS library `axpy` function.

Acknowledgements This work has been supported by project TIN2007-61273 and FEDER, and by INCO2 (Excellence Group founded by PROMETEO 2009/013, High Performance Computing on Current Architectures in Multiple Signal Processing Problems, Generalitat Valenciana).

References

1. Intel (2005) Intel multi-core processor architecture developer backgrounder. White paper
2. Owens JD, Houston M, Luebke D, Green S, Stone JE, Phillips JC (2008) GPU computing. Proc IEEE 96(5):879–899
3. Gasca M, Peña JM (1992) Total positivity and Neville elimination. Linear Algebra Appl 165:25–44
4. Gasca M, Peña JM (1994) A matricial description of Neville elimination with applications to total positivity. Linear Algebra Appl 202:33–45
5. Demmel J, Koev P (2005) The accurate and efficient solution of a totally positive generalized Vandermonde linear system. SIAM J Matrix Anal Appl 27:142–152
6. Gemignani L (2008) Neville elimination for rank-structured matrices. Linear Algebra Appl 428(4):978–991
7. Alonso P, Cortina R, Díaz I, Ranilla J (2004) Neville elimination: a study of the efficiency using checkerboard partitioning. Linear Algebra Appl 393:3–14
8. Alonso P, Díaz I, Cortina R, Ranilla J (2008) Scalability of Neville elimination using checkerboard partitioning. Int J Comput Math 85(3–4):309–317
9. Alonso P, Cortina R, Díaz I, Ranilla J (2009) Blocking Neville elimination algorithm for exploiting cache memories. Appl Math Comput 209(1):2–9
10. Cortina R (2008) El método de Neville: un enfoque basado en Computación de Altas Prestaciones. Ph.D. thesis, Univ. of Oviedo, Spain
11. Chandra R et al (2001) Parallel programming in OpenMP. Morgan Kaufmann, San Mateo