J Supercomput (2014) 68:365–413
DOI 10.1007/s11227-013-1043-3

From tile algorithm to stripe algorithm: a CUBLAS-based parallel implementation on GPUs of Gauss method for the resolution of extremely large dense linear systems stored on an array of solid state devices

Manuel Carcenac

Published online: 23 November 2013

Abstract This paper presents an efficient algorithmic approach to the GPU-based parallel resolution of dense linear systems of extremely large size. A formal transformation of the code of Gauss method allows us to develop for matrix calculations the concept of stripe algorithm, as opposed to that of tile algorithm. Our stripe algorithm is based on the partitioning of the linear system's matrix into stripes of rows and is well suited for efficient implementation on a GPU, using cublasDgemm function of CUBLAS library as the main building block. It is also well adapted to storage of the linear system on an array of solid state devices, the PC memory being used as a cache between the SSDs and the GPU memory. We demonstrate experimentally that our code solves efficiently dense linear systems of size up to 400,000 (160 billion matrix elements) using an NIVDIA C2050 and six 240 GB SSDs.

Keywords Dense linear system · Graphical processing unit (GPU) · CUBLAS · Stripe algorithm · Tile algorithm · Solid state device (SSD)

Abbreviations

GPU	Graphical processing unit
CUDA	Compute unified device architecture
CUBLAS	CUDA basic linear algebra subroutines
SSD	Solid state device
L_{SSD}	1st memory level: array of SSDs
L_{PC}	2nd memory level: main memory of PC
L_{GPU}	3rd memory level: global memory of GPU

M. Carcenac (✉)
Computer Engineering Department, European University of Lefke,
via Mersin 10, Gemikonağı, Turkey
e-mail: manuel.carcenac@gmail.com

1 Introduction

1.1 Some global trends

The field of computer hardware has witnessed in the past few years the maturation of graphical processing units (GPUs) into advanced and extremely powerful parallel coprocessors for PCs. The latest generations of GPUs are now fully adapted to scientific calculations with double-precision floating point arithmetic. Although less versatile for general programming than traditional multicore architectures, modern GPU multicore architecture has already proven its scalability, with up to 448 cores (for the Fermi C2050 GPU) able to execute efficiently parallel versions of well-known scientific algorithms (matrix calculations, ray-tracing, N-body problem…).

Another hardware evolution has been the development of affordable solid state device (SSD) memories that are poised to gradually replace traditional hard disks, at least when high speed of access to mass storage is required.

With regard to matrix calculations and the resolution of linear systems in particular, so-called tile algorithms [1,4,9,11,12,15,20] have been developed that rely on the partitioning of a matrix into a number of small square tiles and on the dynamic distribution—or scheduling—of the tasks associated with these tiles for parallel execution over a multicore architecture.

In the meantime, the CUBLAS library [7,25], a parallel implementation of BLAS library [3] over CUDA GPUs, has been used to execute efficiently over a GPU each of the basic linear algebra tasks required by these tile algorithms [1].

1.2 Stripe algorithm and three-level memory system

As a departure from the concept of tile algorithm, the work presented in this paper introduces what we call a "stripe algorithm". Based on the partitioning of the matrix into relatively few stripes of rows (definitely far fewer than the number of tiles), it appears well adapted to exploiting the performance of CUBLAS as well as to storing efficiently the matrix on an array of SSDs, which allows the processing of extremely large linear systems.

Indeed, whereas GPU-based codes traditionally store the matrix either entirely within the GPU memory or at best in PC memory, the higher granularity of stripes allows for an efficient three-level memory system: first, the array of SSDs stores all the stripes; then, the PC main memory is used as a cache for the stripes being processed (or about to); finally, the GPU global memory contains only a portion of stripe being processed (we should assume that this memory is too limited to contain the whole of a stripe). We will label these three memory levels as $\mathrm{L}_{\mathrm{SSD}}$, L_{PC} and $\mathrm{L}_{\mathrm{GPU}}$, respectively.

Note that the expression "stripe algorithm" is already used in other fields than computational matrix algebra [10,26], but with totally different meanings.

1.3 Methodology and direction of our work

The overall methodology and direction of our work is as follows: first, we derive our "stripe algorithm" from the basic Gauss code through a sequence of formal transformations that we keep to a minimum, thereby avoiding to introduce an extra loop on groups of columns that would have otherwise yielded a tile algorithm. Then we develop specifically for our stripe algorithm a three-level memory system and a dynamic scheduling mechanism that take full advantage of the characteristics of the stripe approach. The resulting combination—stripe algorithm, three-level memory system and scheduling mechanism—is quite general and may a priori be implemented over any performant mass storage solution.

Second, we make use of some affordable hardware currently available on the market, i.e. a PC and some SSDs, to implement and test our system. We admit that our overarching objective has been to fine-tune the implementation to a particular platform, so as to process matrices of extreme size with maximal performance (the issue of portability to other platforms is discussed in Sect. 9.7).

1.4 Organization of the paper

The rest of the paper is organized as follows: in Sect. 2, we present the basic sequential version of Gauss method and we discuss the performance of *cublasDgemm*, a function of CUBLAS library. We also present a first trivial, and inefficient, application of *cublasDgemm* to Gauss method. Then, in Sect. 3, we show how to transform formally the initial pseudocode of Gauss method so as to obtain a group of three mutually independent nested loops, necessary to apply *cublasDgemm* efficiently. This transformation requires only that the matrix be partitioned into stripes of rows, and not tiles.

In Sect. 4, we introduce a further optimization of the code which consists in subdividing each stripe of rows into substripes and applying similar transformations as in Sect. 3, but this time at the level of the substripes. That way, all time-consuming parts of the code now correspond to groups of three mutually independent nested loops that are suitable for efficient use of *cublasDgemm*. Section 5 details a local partial pivoting scheme that is well suited to the stripe-based structure of our code (with partial pivoting performed locally within a substripe). Then, the CUBLAS functions used to implement our pseudo-code are listed in Sect. 6.

In Sect. 7, we develop further the concept of stripe algorithm by storing within the array of SSDs (L_{SSD}) the linear system as stripes of rows and using PC memory (L_{PC}) as a cache between the SSDs and the GPU memory (L_{GPU}). This cache is implemented as an associative memory of cells that represent the currently cached stripes. Section 8 presents the dynamic scheduling strategy that allows an efficient use of L_{PC} cache. It is based on an "N-plicated multithreaded finite-state machine" that updates the state of each cell (to keep track of any operation eventually being applied, or waiting to be applied, on the stripe). Also, a formal code transformation similar to those of Sects. 3 and 4 optimizes the use of the L_{PC} cache memory.

Finally, Sect. 9 exposes the experiments carried out over several test configurations, Sect. 10 compares our approach with other related works and Sect. 11 lists some future developments currently envisioned.

2 Gauss method and cublasDgemm

The resolution with Gauss method of a dense linear system $AX = B$ of size n consists of two phases. First, a triangulation phase that transforms the initial system into a triangular system and then a phase of resolution of that triangular system. The algorithmic complexity of the triangulation is in $O(n^3)$, whereas the triangular system resolution itself is just in $O(n^2)$, we focus our study and parallelization efforts on the triangulation phase. It can be seen as a sequence of partial elimination steps. This sequence is expressed by the loop over index i in the pseudocode of Fig. 1.

As depicted graphically in Fig. 2, a partial elimination step for a given value of i amounts to updating with the formula

$$A_{jk} \leftarrow A_{jk} - R_j \times A_{ik} \tag{1}$$

every element of the submatrix $A_{i+1\ldots n-1, i\ldots n-1}$.

```
triangulation
  for i ← 0 , … , n - 1 :
      for j ← i + 1 , … , n - 1 :
          $R_j \leftarrow A_{ji} / A_{ii}$
          $B_j \leftarrow B_j - R_j \times B_i$

      for j ← i + 1 , … , n - 1 :
          for k ← i , … , n - 1 :
              $A_{jk} \leftarrow A_{jk} - R_j \times A_{ik}$
```

Fig. 1 Pseudocode of triangulation phase of Gauss method (sequential mode)

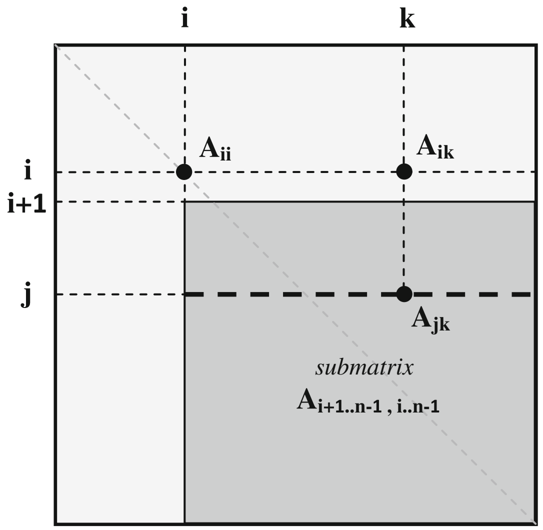


Fig. 2 Partial elimination step, for a given i, of Gauss triangulation

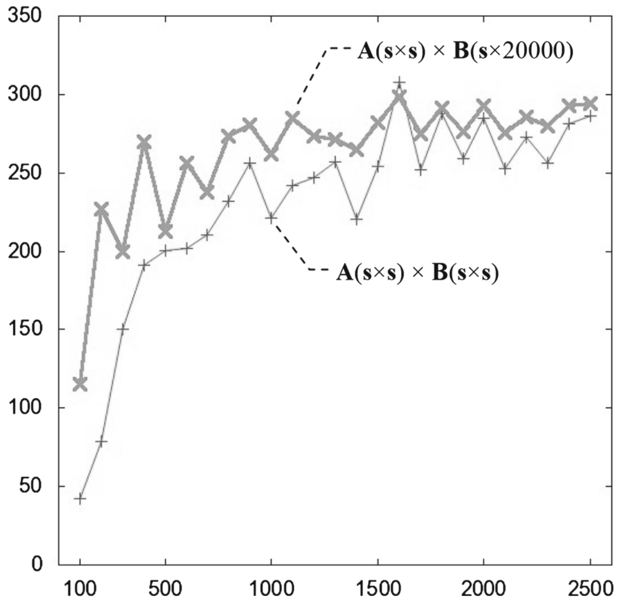


Fig. 3 Performance on a C2050, in Gflops and in function of s, of *cublasDgemm* for $A(s \times s) \times B(s \times s)$ and $A(s \times s) \times B(s \times 20{,}000)$

Although the sequence of elimination steps, i.e. the loop over i, is inherently sequential, there is ample implicit parallelism within each elimination step. Indeed, all the elements of $A_{i+1\ldots n-1, i\ldots n-1}$ may be updated with Eq. 1 independently from each other.

We now discuss the use of the *cublasDgemm* function of CUBLAS library to implement Eq. 1.

CUBLAS is an adaptation of the classical BLAS (Basic Linear Algebra Subprograms) library to the CUDA architecture. With highly optimized data transfers between the diverse memories—global memory, L1 and L2 cache memories, shared memories, core registers—of a CUDA GPU, it is the tool of choice to achieve extreme performance for matrix calculations. The CUBLAS function that is especially famous for its performance is *cublasDgemm*. With double-precision floating point arithmetic, it multiplies two matrices together and adds the result to a third matrix (modulo coefficients $\boldsymbol{\alpha}$ and $\boldsymbol{\beta}$):

$$C \leftarrow \alpha \times A \times B + \beta \times C. \tag{2}$$

Figure 3 exposes the actual performance in Gigaflops of *cublasDgemm* on an NVIDIA C2050 for the matrix products $A(s \times s) \times B(s \times s)$ and $A(s \times s) \times B(s \times 20{,}000)$; this last product corresponding to the basic operation required by the "stripe algorithm" that we develop later in this paper (with s corresponding to a stripe width and 20,000 to a fraction of a stripe length).

It appears that *cublasDgemm* performs particularly well with matrices of large size, especially for the product $A(s \times s) \times B(s \times 20{,}000)$.

The fundamental reason for the high performance of *cublasDgemm* over large matrices is that while the computations are in $O(n^3)$ (due to the 3 nested loops of matrix multiplication), the data are only in $O(n^2)$. This so-called surface-to-volume property [12] means that, in practice, the same data may be loaded once then accessed and processed several times. This leaves open ground for thorough optimizations of data transfers relative to the actual calculations.

As a first attempt to apply CUBLAS over Gauss code (Fig. 1), we may use *cublas-Dgemm* for the elimination step over submatrix $A_{i+1\ldots n-1,i\ldots n-1}$, i.e. to apply in parallel (Eq. 1) over all its elements. Since, in Fig. 1, the inner loops over *j* and *k* are mutually independent and yet are both dependent on the outer loop over *i*, *cublasDgemm* can be used directly only over loops *j* and *k* for a given *i*. This way, *cublasDgemm* actually carries out the multiplication of a column vector by a row vector.

We obtain a very poor performance of at most eight Gflops. This is easily explained by the fact that the computations for the multiplication of a column vector by a row vector are only in $O(n^2)$. Consequently, our objective should be to apply *cublasDgemm* over sets of three mutually independent nested loops in order to have computations in $O(n^3)$.

3 Matrix partition and optimized pseudocode for cublasDgemm

The basic idea is to transform formally the original code so as to obtain a local set of three mutually independent nested loops. To that avail, we first replace equivalently each of the loops over the rows, i.e. over *i* and *j*, with two nested loops: an outer loop over some groups of rows, or horizontal stripes, and an inner loop over the rows within each horizontal stripe of rows.

Assuming that *S* is the maximal size of a stripe of rows, i.e. the maximal possible number of rows that it contains, then the total number of stripes is

$$N \leftarrow \begin{cases} n/s & \text{if } S \text{ divides } n \\ n/s + 1 & \text{else} \end{cases} \tag{3}$$

and the index *I* of the stripe of rows has range [*0, N* [. Also, the range of a row index within stripe *I* is $[i_0, i_1[$, with $i_0 \leftarrow I \times n/N$ and $i_1 \leftarrow (I+1) \times n/N$ (using integer arithmetic); similarly with the range $[j_0, j_1[$ of stripe *J*. Notice that, in case *S* does not divide *n*, this partitioning distributes the truncature effect along the stripes, thus avoiding the embarrassing case of the last stripe containing as little as a single row.

The following formal template transformation is consequently carried out for the loop over *i*:

```
                                          for I ← 0 , ... , N - 1 :
for i ← 0 , ... , n - 1 : [ . . .   ──►      i₀ ← I × n / N  ;  i₁ ← (I + 1) × n / N        (4)
                                             for i ← i₀ , ... , i₁ - 1 : [ . . .
```

The next step of the formal code transformation consists in rearranging the loops so that the loops over the stripe indices become the outer loops and the loops over the rows within the stripes become the inner, local, loops. To that avail, it is necessary to first replace the loop over j from $i + 1$ to $n - 1$ with a sequence of two loops, from $i+1$ to $i_1 - 1$ and from i_1 to $n-1$ (assuming that i is in the range $[i_0, i_1[$), this second loop being then replaced with two nested loops (as in Eq. 4):

```
for j ← i + 1 , … , n - 1 : [ . . .   ──►   for j ← i + 1 , … , i₁ - 1 : [ . . .

                                             for J ← I + 1 , … , N - 1 :
                                                 j₀ ← J × n / N ; j₁ ← (J + 1) × n / N
                                                 for j ← j₀ , … , j₁ - 1 : [ . . .
```
(5)

Indeed, the first values of j after i (often only a fraction of a stripe) are in general positioned before the first stripe of rows of index $J = I + 1$.

Similarly, the loop over k from i to $n - 1$ is replaced with a sequence of two loops, i.e. from i to $i_1 - 1$ and from i_1 to $n - 1$.

However, it should be stressed that we have deemed unnecessary to replace this second loop over k with two nested loops, i.e. it is unnecessary to group the columns as well. That way, we are left in the final code with a non-local loop over k that ranges in general over a sizeable fraction of each matrix row. Put simply, partitioning the matrix into horizontal stripes only—instead of tiles—is enough to create a set of three mutually independent nested loops. Hence the origin of our conceptual shift from the notion of tile algorithm to that of stripe algorithm.

Of course, the code will grow into an assemblage of several combinations of loops. And, for each case in which these inner loops are nested by three and happen to be mutually independent, we will then be able to apply *cublasDgemm* efficiently.

In effect, our reorganization of the code amounts to partitioning the matrix into four areas for elimination, as shown in Fig. 4.

Corresponding to row stripe index *I*, area ⟨*II*⟩ is a square block and area ⟨*IK*⟩ is a rectangular block. Then, covering all values of *J* from *I + 1* to *N − 1*, area ⟨*JI*⟩ is a

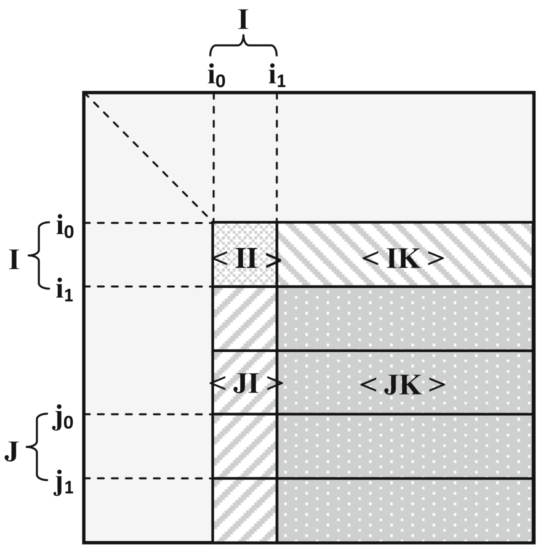


Fig. 4 Partition of the matrix with elimination areas ⟨*II*⟩, ⟨*IK*⟩, ⟨*JI*⟩, ⟨*JK*⟩ and stripes of rows

```
triangulation
  for I ← 0 , ... , N - 1 :
    $i_0$ ← I × n / N  ;  $i_1$ ← (I + 1) × n / N

    elimination block < II >

    elimination block < IK >

    if I < N - 1 :
      for J ← I + 1 , ... , N - 1 :
        $j_0$ ← J × n / N  ;  $j_1$ ← (J + 1) × n / N

        elimination block < JI >

        elimination block < JK >
```

```
elimination block < II >
  for i ← $i_0$ , ... , $i_1$ - 1 :
    for j ← i + 1 , ... , $i_1$ - 1 :
      $R_{j-j0,i-i0}$ ← $A_{ji}$ / $A_{ii}$
      $B_j$ ← $B_j$ - $R_{j-j0,i-i0}$ × $B_i$

    for j ← i + 1 , ... , $i_1$ - 1 :
      for k ← i , ... , $i_1$ - 1 :
        $A_{jk}$ ← $A_{jk}$ - $R_{j-j0,i-i0}$ × $A_{ik}$
```

```
elimination block < IK >
  for i ← $i_0$ , ... , $i_1$ - 1 :
    for j ← i + 1 , ... , $i_1$ - 1 :
      for k ← $i_1$ , ... , n - 1 :
        $A_{jk}$ ← $A_{jk}$ - $R_{j-j0,i-i0}$ × $A_{ik}$
```

```
elimination block < JI >
  for i ← $i_0$ , ... , $i_1$ - 1 :
    for j ← $j_0$ , ... , $j_1$ - 1 :
      $R_{j-j0,i-i0}$ ← $A_{ji}$ / $A_{ii}$
      $B_j$ ← $B_j$ - $R_{j-j0,i-i0}$ × $B_i$

    for j ← $j_0$ , ... , $j_1$ - 1 :
      for k ← i , ... , $i_1$ - 1 :
        $A_{jk}$ ← $A_{jk}$ - $R_{j-j0,i-i0}$ × $A_{ik}$
```

```
elimination block < JK >
  for i ← $i_0$ , ... , $i_1$ - 1 :
    for j ← $j_0$ , ... , $j_1$ - 1 :
      for k ← $i_1$ , ... , n - 1 :
        $A_{jk}$ ← $A_{jk}$ - $R_{j-j0,i-i0}$ × $A_{ik}$
```

Fig. 5 Pseudocode of Gauss triangulation corresponding to the matrix partition of Fig. 4

group of square blocks (to within the truncature effects) and area $\langle JK \rangle$ is a group of rectangular blocks.

Area $\langle II \rangle$ is separated from $\langle IK \rangle$ because it corresponds not only to the application of Eq. 1 over its elements, but also to the prior calculation of ratio A_{ji}/A_{ii} for each row i of stripe I and for each row j of I such that $j > i$. Consequently, area $\langle II \rangle$ must be processed before area $\langle IK \rangle$. Similarly, each block of $\langle JI \rangle$ must be processed before its following block in $\langle JK \rangle$ (with every j in $[j_0, j_1[$ being now always strictly superior to every i in $[i_0, i_1[$).

In practice, the block of $\langle II \rangle$, as well as each block within $\langle JI \rangle$, calculates its ratio values with

$$R_{j-j0,i-i0} \leftarrow A_{ji}/A_{ii} \tag{6}$$

and stores them inside matrix R of size $[i_0, i_1[\times[i_0, i_1[$, respectively, $[j_0, j_1[\times[i_0, i_1[$, for subsequent use inside the following rectangular block within $\langle IK \rangle$ or $\langle JK \rangle$.

Figure 5 shows the resulting pseudocode with the top code calling four functions that process the blocks within areas $\langle II \rangle$, $\langle IK \rangle$, $\langle JI \rangle$ and $\langle JK \rangle$.

So far, our pseudocode exhibits a set of three mutually independent local loops only for the elimination step over the rectangular blocks of area $\langle JK \rangle$. Thus, it is only over this area that *cublasDgemm* can be used efficiently to carry out the fundamental operation underlying our approach, i.e. the application of Eq. 1 to a stripe of rows.

As for the other areas, their treatments involve dependent loops. In particular, for $\langle IK \rangle$, a loop over j is dependent on its enclosing loop over i. As for $\langle JI \rangle$, it involves a loop over k that is also dependent on an enclosing loop over i. Even more problematically, $\langle II \rangle$ contains two nested loops that are both dependent on an enclosing loop over i.

4 Further optimization over blocks ⟨IK⟩, ⟨JI⟩ and ⟨II⟩

In each case of a loop dependency and to alleviate the dependency of the inner loop on its enclosing loop over i from i_0 to $i_1 - 1$ (corresponding to row stripe I), we apply the same methodology as in previous Section, but at a local level, i.e. within the row stripe. In substance, we subdivide row stripe I into some substripes of rows and replace equivalently the stripe-local loop over i from i_0 to $i_1 - 1$ with two nested loops: an outer loop over the substripes of the row stripe and an inner loop over the rows within each substripe.

If we choose $S' < S$ as the maximal size of a row substripe, the number of substripes within stripe I of range $[i_0, i_1[$ is given by

$$N' \leftarrow \begin{cases} (i_1 - i_0)/S' & \text{if } S' \text{ divides } (i_1 - i_0) \\ (i_1 - i_0)/S' + 1 & \text{else} \end{cases} \tag{7}$$

and the index I' of the substripe has range $[0, N'[$. Also, the range of a row index within substripe I' is $[i'_0, i'_1[$, with $i'_0 \leftarrow i_0 + I' \times (i_1 - i_0)/N'$ and $i'_1 \leftarrow i_0 + (I' + 1)\times (i_1 - i_0)/N'$ (still assuming integer arithmetic).

Following up with previous section's methodology, we replace for $\langle IK \rangle$ the loop over j from $i + 1$ to $i_1 - 1$ with a sequence of two loops, one over j from $i + 1$ to $i'_1 - 1$ and the other over j from i'_1 to $i'_1 - 1$:

$$\textbf{for } j \leftarrow i+1, \ldots, i_1-1 : \left[\ldots \right. \longrightarrow \begin{cases} \textbf{for } j \leftarrow i+1, \ldots, i'_1-1 : \left[\ldots \right. \\ \textbf{for } j \leftarrow i'_1, \ldots, i_1-1 : \left[\ldots \right. \end{cases} \tag{8}$$

That way, loop dependency still occurs only for the first of these inner loops and the second one becomes part of a set of three mutually independent loops that will then be implemented most efficiently with *cublasDgemm*. Further, since the range $[i+1, i'_1[$ is in general substantially smaller than $[i'_1, i_1[$ (assuming that there is at least an order of magnitude between S' and S), the remaining non-optimized part will remain much smaller than the optimized part.

Similarly, for $\langle JI \rangle$, the loop over k from i to $i_1 - 1$ is replaced with one over k from i to $i'_1 - 1$ and another over k from i'_1 to $i_1 - 1$.

Figure 6 shows the resulting sub-partitioning of the rectangular block of $\langle IK \rangle$ and of each of the blocks of $\langle JI \rangle$. Notice that for $\langle JI \rangle$, this sub-partitioning actually occurs over the columns (k is a column index such that $k \geq i$).

We now present in Fig. 7 the resulting optimized pseudocode for the elimination operations over the blocks of $\langle IK \rangle$ and $\langle JI \rangle$.

In order to achieve complete optimization, we must also consider area $\langle II \rangle$, even though its size is much less than that of the other areas (at least for very large linear systems). The difficulty of this area, which we have come to consider as a sort of "meta-pivot", is that its elimination step contains a set of three nested loops with the inner ones (over j and k) being simultaneously dependent on the outer one (over i).

Actually, the generalization of our methodology to this case is quite straightforward provided that we follow the sub-partitioning of Fig. 8 with sub-areas $\langle A \rangle$, $\langle B \rangle$, $\langle C \rangle$ and $\langle D \rangle$.

Figure 9 provides the optimized pseudocode for $\langle II \rangle$ with the four sections corresponding to each of the sub-areas.

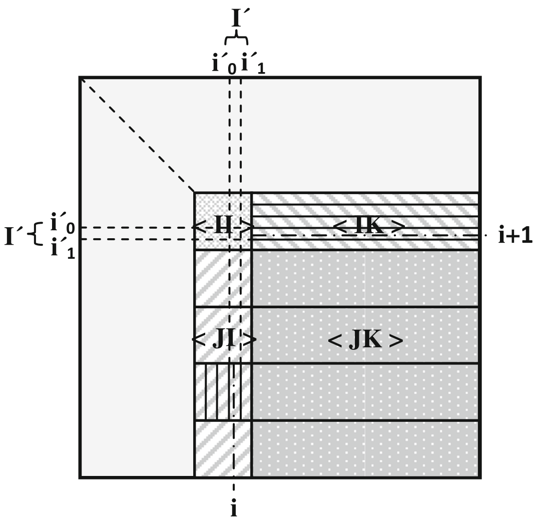


Fig. 6 Substripes of rows—subpartitioning of areas $\langle IK \rangle$ and $\langle JI \rangle$

Fig. 7 Optimized pseudocodes for ⟨*IK*⟩ and ⟨*JI*⟩

```
elimination block < IK >
for $I' \leftarrow 0, \ldots, N' - 1$ :
    $i'_0 \leftarrow i_0 + I' \times (i_1 - i_0) / N'$
    $i'_1 \leftarrow i_0 + (I' + 1) \times (i_1 - i_0) / N'$

    for $i \leftarrow i'_0, \ldots, i'_1 - 1$ :
        for $j \leftarrow i + 1, \ldots, i'_1 - 1$ :
            for $k \leftarrow i_1, \ldots, n - 1$ :
                $A_{jk} \leftarrow A_{jk} - R_{j-j0,\, i-i0} \times A_{ik}$

    for $i \leftarrow i'_0, \ldots, i'_1 - 1$ :
        for $j \leftarrow i'_1, \ldots, i_1 - 1$ :
            for $k \leftarrow i_1, \ldots, n - 1$ :
                $A_{jk} \leftarrow A_{jk} - R_{j-j0,\, i-i0} \times A_{ik}$
```

```
elimination block < JI >
for $I' \leftarrow 0, \ldots, N' - 1$ :
    $i'_0 \leftarrow i_0 + I' \times (i_1 - i_0) / N'$
    $i'_1 \leftarrow i_0 + (I' + 1) \times (i_1 - i_0) / N'$

    for $i \leftarrow i'_0, \ldots, i'_1 - 1$ :
        for $j \leftarrow j_0, \ldots, j_1 - 1$ :
            $R_{j-j0,\, i-i0} \leftarrow A_{ji} / A_{ii}$
            $B_j \leftarrow B_j - R_{j-j0,\, i-i0} \times B_i$

        for $j \leftarrow j_0, \ldots, j_1 - 1$ :
            for $k \leftarrow i, \ldots, i'_1 - 1$ :
                $A_{jk} \leftarrow A_{jk} - R_{j-j0,\, i-i0} \times A_{ik}$

    for $i \leftarrow i'_0, \ldots, i'_1 - 1$ :
        for $j \leftarrow j_0, \ldots, j_1 - 1$ :
            for $k \leftarrow i'_1, \ldots, i_1 - 1$ :
                $A_{jk} \leftarrow A_{jk} - R_{j-j0,\, i-i0} \times A_{ik}$
```

5 Local partial pivoting scheme

A critical issue with Gauss elimination method is its numerical instability: successive divisions by pivot values gradually amplify the truncature errors that even double-precision floating point arithmetic operations unavoidably create. The classical approach to limiting this instability is the so-called partial pivoting method: instead of directly dividing matrix elements A_{ik} by A_{ii} as in Fig. 1, we search in the part of column below A_{ii} for the element of highest amplitude. If such an element is found, then we swap rows so that this element now occupies the place of A_{ii} and is conse-

Fig. 8 Substripes of rows—subpartitioning of area ⟨*II*⟩ into sub-areas ⟨*A*⟩, ⟨*B*⟩, ⟨*C*⟩ and ⟨*D*⟩

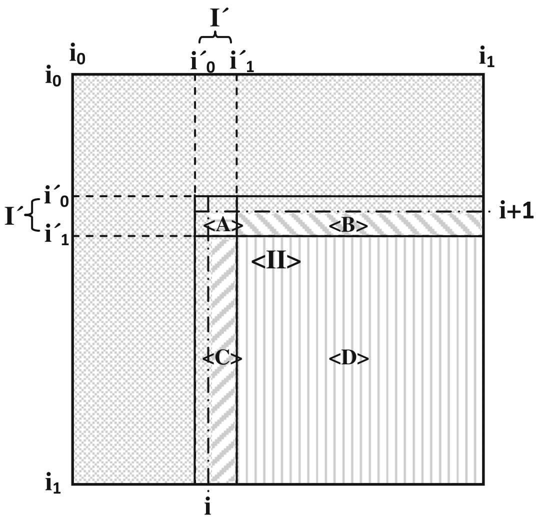


Fig. 9 Optimized pseudocode for ⟨*II*⟩

```
elimination block < II >
  for $I' \leftarrow 0, \ldots, N' - 1$ :
    $i'_0 \leftarrow i_0 + I' \times (i_1 - i_0) / N'$
    $i'_1 \leftarrow i_0 + (I' + 1) \times (i_1 - i_0) / N'$

    for $i \leftarrow i'_0, \ldots, i'_1 - 1$ :
      for $j \leftarrow i + 1, \ldots, i_1 - 1$ :
        $R_{j-j0, i-i0} \leftarrow A_{ji} / A_{ii}$
        $B_j \leftarrow B_j - R_{j-j0, i-i0} \times B_i$

      for $j \leftarrow i + 1, \ldots, i'_1 - 1$ :        // < A >
        for $k \leftarrow i, \ldots, i'_1 - 1$ :
          $A_{jk} \leftarrow A_{jk} - R_{j-j0, i-i0} \times A_{ik}$

      for $j \leftarrow i + 1, \ldots, i'_1 - 1$ :        // < B >
        for $k \leftarrow i'_1, \ldots, i_1 - 1$ :
          $A_{jk} \leftarrow A_{jk} - R_{j-j0, i-i0} \times A_{ik}$

      for $j \leftarrow i'_1, \ldots, i_1 - 1$ :        // < C >
        for $k \leftarrow i, \ldots, i'_1 - 1$ :
          $A_{jk} \leftarrow A_{jk} - R_{j-j0, i-i0} \times A_{ik}$

    for $i \leftarrow i'_0, \ldots, i'_1 - 1$ :
      for $j \leftarrow i'_1, \ldots, i_1 - 1$ :        // < D >
        for $k \leftarrow i'_1, \ldots, i_1 - 1$ :
          $A_{jk} \leftarrow A_{jk} - R_{j-j0, i-i0} \times A_{ik}$
```

quently used for the divisions. Figure 10 shows the resulting pseudocode (with p index of the element of highest amplitude).

A practical drawback of partial pivoting is that it requires for each value of i a search among $n - i$ values, with row index $q \in [i, n[$. In our case, the extra time required for these searches would be felt all the more since we have already heavily parallelized the arithmetic operations of Gauss method.

Actually, other less cumbersome pivoting strategies exist, in particular pairwise pivoting which consists in comparing (and eventually swapping) only couples of possible pivot values. However, pairwise pivoting is known to be less stable numerically than partial pivoting [1,22].

Another alternative pivoting strategy is the so-called local partial pivoting, in effect a weaker version of partial pivoting. It consists in limiting the partial pivoting search to within just a fraction of the row index interval $[i, n[$. Originally presented in [2,6] as a way to avoid extra communications within a distributed-memory multiprocessor (by localizing the search within the matrix part stored in the local memory of a given processor), local partial pivoting also appears well suited to the stripe-based structure of our code.

Indeed, we may consider localizing the search within stripe I since it is currently in cache memory L_{PC}, whereas most other stripes are in L_{SSD} and thus not immediately accessible (see Sect. 7). Actually, we have decided for two reasons to perform local partial pivoting only within the range of row substripe I', i.e. with row index $q \in [i, i'_1[$ instead of $[i, i_1[$. First, the inclusion of pivoting within the optimized pseudocode of Fig. 9 is then straightforward. Second, localizing the pivot search over the full range $[i, i_1[$ offered by stripe I would take excessive time in practice (at least for $n < 100{,}000$, see Sect. 9.4 and Fig. 25), however limited that range already is.

Figure 11 shows, as a generalization of Fig. 9 code, the inclusion of local partial pivoting within the optimized pseudocode for $\langle II \rangle$.

Fig. 10 Sequential pseudocode of triangulation phase with partial pivoting

```
triangulation
  for i ← 0 , ... , n - 1 :
      p ← index of max( |A_qi| ) with q ∈ [ i , n [
      if p ≠ i :
          for q ← i , ... , n - 1 :
              A_iq ↔ A_pq
          B_i ↔ B_p
      for j ← i + 1 , ... , n - 1 :
          R_j ← A_ji / A_ii
          B_j ← B_j - R_j × B_i
      for j ← i + 1 , ... , n - 1 :
          for k ← i , ... , n - 1 :
              A_jk ← A_jk - R_j × A_ik
```

Fig. 11 Optimized pseudocode for ⟨*II*⟩ with local partial pivoting

```
elimination block < II >
for $I' \leftarrow 0, \dots, N'-1$ :
    $i'_0 \leftarrow i_0 + I' \times (i_1 - i_0) / N'$
    $i'_1 \leftarrow i_0 + (I'+1) \times (i_1 - i_0) / N'$

    for $i \leftarrow i'_0, \dots, i'_1 - 1$ :

        $p \leftarrow$ index of $\max(|A_{qi}|)$ with $q \in [\, i, i'_1 [$

        if $p \neq i$ :
            for $q \leftarrow i, \dots, i_1 - 1$ :    // within < II >
                $A_{iq} \leftrightarrow A_{pq}$

            for $q \leftarrow i_1, \dots, n - 1$ :    // within < IK >
                $A_{iq} \leftrightarrow A_{pq}$

            $B_i \leftrightarrow B_p$

            for $q \leftarrow i_0, \dots, i - 1$ :
                $R_{i-i0,\, q-i0} \leftrightarrow R_{p-i0,\, q-i0}$


        for $j \leftarrow i+1, \dots, i_1 - 1$ :
            $R_{j-j0,\, i-i0} \leftarrow A_{ji} / A_{ii}$
            $B_j \leftarrow B_j - R_{j-j0,i-i0} \times B_i$

        for $j \leftarrow i+1, \dots, i'_1 - 1$ :    // < A >
            for $k \leftarrow i, \dots, i'_1 - 1$ :
                $A_{jk} \leftarrow A_{jk} - R_{j-j0,\, i-i0} \times A_{ik}$

        for $j \leftarrow i+1, \dots, i'_1 - 1$ :    // < B >
            for $k \leftarrow i'_1, \dots, i_1 - 1$ :
                $A_{jk} \leftarrow A_{jk} - R_{j-j0,\, i-i0} \times A_{ik}$

        for $j \leftarrow i'_1, \dots, i_1 - 1$ :    // < C >
            for $k \leftarrow i, \dots, i'_1 - 1$ :
                $A_{jk} \leftarrow A_{jk} - R_{j-j0,\, i-i0} \times A_{ik}$


    for $i \leftarrow i'_0, \dots, i'_1 - 1$ :
        for $j \leftarrow i'_1, \dots, i_1 - 1$ :    // < D >
            for $k \leftarrow i'_1, \dots, i_1 - 1$ :
                $A_{jk} \leftarrow A_{jk} - R_{j-j0,\, i-i0} \times A_{ik}$
```

Notice that the swapping of rows *i* and *q* is split into the swapping of their parts within ⟨*II*⟩ and the swapping of their parts within ⟨*IK*⟩. Indeed, at the time of the execution of Fig. 11 code, part ⟨*II*⟩ of stripe *I* will be present in GPU memory whereas part ⟨*IK*⟩ will be accessible in PC memory, see Sects. 6 and 7. Further, Fig. 11 code must also swap elements of B as well as parts of rows within matrix R that have already been calculated.

6 Implementation with CUBLAS

Of course, *cublasDgemm* (double-precision BLAS3 level) is at the core of the actual implementation with the CUBLAS library. Nevertheless, to perform all calculations (including the triangular system resolution) within the GPU—and thus limit GPU-PC data transfers—we also use some double-precision BLAS1 and BLAS2 level CUBLAS functions and even a CUDA kernel (as a matter of fact, BLAS1 and BLAS2 do not offer such a high level of performance, especially compared with BLAS3).

For the elimination over blocks $\langle JI \rangle$, R is calculated with *cublasDcopy* and *cublas-Dscal* (both BLAS1 level), whereas B is updated with *cublasDaxpy* (BLAS1).

As for the elimination over block $\langle II \rangle$, we found it more appropriate to define directly a CUDA kernel to calculate R and update B; this kernel corresponding to a 1D partition over index $j \in [i+1, i_1'[$ which is distributed over the 32 cores of a single multiprocessor of the C2050. Indeed, due to the dependency on i of the loop over j, we must access individual values of A_{ii} and B_i within the GPU memory, and that is done most readily inside a kernel.

The local partial pivoting scheme is implemented as follows: the local search for a pivot of highest amplitude is performed with *cublasIdamax* (BLAS1). Further, to swap two rows, we apply *cublasDswap* over their parts that are present in $\mathrm{L_{GPU}}$ (columns in range $[i, i_1[$) whereas their parts that reside only in $\mathrm{L_{PC}}$ (columns in $[i_1, n[$) are swapped directly by the PC, see Sect. 7. *cublasDswap* is also used to swap individual elements of B, present in $\mathrm{L_{GPU}}$, and the already calculated parts of some rows of R, also in $\mathrm{L_{GPU}}$.

Finally, the resolution of the upper triangular system is done progressively over each stripe of the triangular matrix, from the last stripe up to the first one. For each stripe, we first multiply a portion of this stripe with the corresponding part of B using *cublasDscal* (BLAS1) and *cublasDgemv* (BLAS2). Then, we elaborate with *cublasDcopy* and *cublasDaxpy* (both BLAS1) the second term of the small triangular system corresponding to the stripe and solve it with *cublasDtrsm* (BLAS3).

7 Memory system

7.1 Three-level memory system for matrix stripes

The memory of a GPU is typically limited to no more than a few Gibibytes (1 Gibibyte = 2^{30} bytes and 1 Gigabyte = 10^9 bytes). In particular, the Tesla C2050 contains 3 GiB (Gibibytes) of global memory, which allows it to store in its entirety a dense linear system of size at most 19,000 or so. With the PC memory generally an order of magnitude higher, an obvious solution consists in storing within it the whole matrix of the linear system. Then, only the matrix parts that are currently needed in the course of calculations are copied into the GPU memory, processed and sent back to the PC memory. With a typical 24 GiB of PC memory, it becomes possible to store and process dense systems of size 50,000 or so. But this approach does not currently allow the resolution on a PC of dense linear systems whose size is in the hundreds of thousands.

To that end, we have developed an efficient three-level memory system. An array of six Solid-State Devices (240 GB each for a total of 1,440 GB), labeled $\mathrm{L_{SSD}}$, stores the complete matrix. Depending on the calculations, some matrix parts are loaded into a cache memory, labeled $\mathrm{L_{PC}}$, that occupies most of the PC memory. Finally, the matrix parts are loaded from this cache memory into the GPU global memory, labeled $\mathrm{L_{GPU}}$, for actual processing. Then, the processed matrix parts are downloaded back into the cache where they will await further processing. If left untouched for a certain time, they will eventually be stored back into $\mathrm{L_{SSD}}$ to make room in $\mathrm{L_{PC}}$ cache for other parts.

In essence, our code handles a cache within the PC memory to alleviate the bottleneck of access to the SSDs. Indeed, although the SSDs are organized as an array to parallelize access to their data (see Sect. 7.3), the overall bandwidth is still substantially less than that of the PCIe $\times$ 16 bus connecting the PC with the GPU (see Sect. 9.6).

Concerning the definition of matrix parts, tile algorithms are based on the classical assumption that such matrix parts should be tiles, i.e. square submatrices of relatively small size. However, the preceding algorithmic transformation of Gauss method (see Sect. 3) initiated in our comprehending a drift from that assumption. Indeed, we realized that such a transformation requires only a pretty basic partition of the matrix into stripes of rows to be suitable for efficient implementation with CUBLAS. Actually, the dependencies due to the prior calculation of the ratio values A_{ji}/A_{ii} before actual elimination over the remainders of the rows do naturally lead to a row-oriented structuration and partitioning.

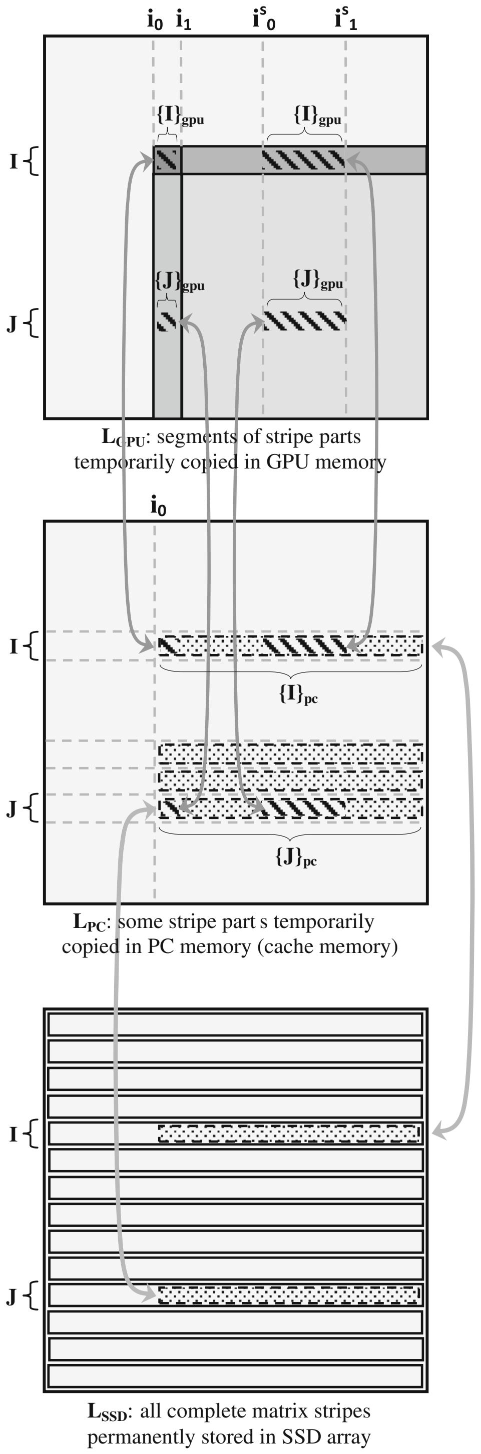
Consequently, as a practical follow-up to these purely algorithmic considerations, we have developed an efficient stripe-based organization of data structures and of data transfers between our three levels of memory. Indeed, each stripe replaces in practice a sequence of tiles and offers much greater granularity.

Each of the N stripes of rows is organized as a monolithic bloc (although it will actually be partitioned in segments for storage within the SSD array). Following a requirement from CUBLAS library (dating back to the days when BLAS was originally developed for Fortran), the elements within this bloc are listed in column-major order.

Coincidentally, this turns out to be well suited to the fact that we need, for an elimination step over a stripe, to transfer from $\mathrm{L_{SSD}}$ to $\mathrm{L_{PC}}$ only the part of stripe that starts with column index i_0 and extends to its rightmost extremity. Indeed, assuming that the bloc of memory storing stripe I (with $(i_1 - i_0) \times n$ elements) has a start pointer A_I, then column-major order implies that the part of stripe to be transferred corresponds to the $(i_1 - i_0) \times (n - i_0)$ contiguous elements starting from pointer $A_I + (i_1 - i_0) \times i_0$. Similarly, for any stripe J of start index A_J, the $(j_1 - j_0) \times (n - i_0)$ contiguous elements starting from pointer $A_J + (j_1 - j_0) \times i_0$ are to be transferred.

Further, the limited GPU memory being unable to store entire stripes, only segments of stripes of maximal length $S^{\mathrm{s}} = 20{,}000$ are loaded and processed in the GPU at a time.

Fig. 12 Transfers of stripes between L_{SSD}, L_{PC} and L_{GPU} memory levels



i0 i1 is0 is1
{I}gpu
{I}gpu
I
{J}gpu
{J}gpu
J
LGPU: segments of stripe parts temporarily copied in GPU memory
i0
I
{I}pc
J
{J}pc
LPC: some stripe part s temporarily copied in PC memory (cache memory)
I
J
LSSD: all complete matrix stripes permanently stored in SSD array

```
triangulation
  for $I \leftarrow 0, \dots, N-1$ :
    $i_0 \leftarrow I \times n / N$ ; $i_1 \leftarrow (I+1) \times n / N$

    $\{I\}_{gpu} \leftarrow$ segment [ $i_0, i_1$ [ of $\{I\}_{pc}$
    elimination block < II > over $\{I\}_{gpu}$
    segment [ $i_0, i_1$ [ of $\{I\}_{pc} \leftarrow \{I\}_{gpu}$

    $N^s \leftarrow$ number of segments of max. length $S^s$ in [ $i_1$, n [
    for $I^s \leftarrow 0, \dots, N^s - 1$ :
      $i^s_0 \leftarrow i_1 + I^s \times (n - i_1) / N^s$
      $i^s_1 \leftarrow i_1 + (I^s + 1) \times (n - i_1) / N^s$

      $\{I\}_{gpu} \leftarrow$ segment [ $i^s_0, i^s_1$ [ of $\{I\}_{pc}$
      elimination block < IK > over $\{I\}_{gpu}$
      segment [ $i^s_0, i^s_1$ [ of $\{I\}_{pc} \leftarrow \{I\}_{gpu}$


    if $I < N - 1$ :

      for $J \leftarrow I + 1, \dots, N - 1$ :
        $j_0 \leftarrow J \times n / N$ ; $j_1 \leftarrow (J + 1) \times n / N$

        $\{I\}_{gpu} \leftarrow$ segment [ $i_0, i_1$ [ of $\{I\}_{pc}$
        $\{J\}_{gpu} \leftarrow$ segment [ $i_0, i_1$ [ of $\{J\}_{pc}$
        elimination block < JI > over $\{J\}_{gpu}$ using $\{I\}_{gpu}$
        segment [ $i_0, i_1$ [ of $\{J\}_{pc} \leftarrow \{J\}_{gpu}$

        for $I^s \leftarrow 0, \dots, N^s - 1$ :
          $i^s_0 \leftarrow i_1 + I^s \times (n - i_1) / N^s$
          $i^s_1 \leftarrow i_1 + (I^s + 1) \times (n - i_1) / N^s$

          $\{I\}_{gpu} \leftarrow$ segment [ $i^s_0, i^s_1$ [ of $\{I\}_{pc}$
          $\{J\}_{gpu} \leftarrow$ segment [ $i^s_0, i^s_1$ [ of $\{J\}_{pc}$
          elimination block < JK > over $\{J\}_{gpu}$ using $\{I\}_{gpu}$
          segment [ $i^s_0, i^s_1$ [ of $\{J\}_{pc} \leftarrow \{J\}_{gpu}$
```

Fig. 13 Overall computational scheme with PC-GPU transfers of stripe segments

```
elimination block < JK >
  for $\Delta i \leftarrow 0, \dots, i_1 - i_0 - 1$ :
    for $\Delta j \leftarrow 0, \dots, j_1 - j_0 - 1$ :
      for $\Delta k \leftarrow 0, \dots, i^s_1 - i^s_0 - 1$ :
        $A^J_{\Delta j, \Delta k} \leftarrow A^J_{\Delta j, \Delta k} - R_{\Delta j, \Delta i} \times A^I_{\Delta i, \Delta k}$
```

$\longrightarrow$

```
cublasDgemm ( 'N' , 'N'
            , $j_1 - j_0$ , $i^s_1 - i^s_0$ , $i_1 - i_0$
            , -1.0
            , R , $j_1 - j_0$
            , $A^I$ , $i_1 - i_0$
            , +1.0
            , $A^J$ , $j_1 - j_0$ )
```

Fig. 14 Actual pseudocode for $\langle JK \rangle$ and its implementation with *cublasDgemm*; A^I, A^J and R are assumed to represent pointers in GPU global memory and the column-major leading dimensions of their blocks are $i_1 - i_0$, $j_1 - j_0$ and $j_1 - j_0$, respectively

Figure 12 summarizes the transfers of stripe parts between L_{SSD}, L_{PC} and L_{GPU}, with the stripes being represented inside the matrix structure and according to the elimination pattern of Fig. 4. $\{I\}_{pc}$ and $\{J\}_{pc}$ (both of size $S \times (n - i_0)$) are the parts of stripes I and J that currently reside in L_{PC} cache, whereas $\{I\}_{gpu}$ and $\{J\}_{gpu}$ (both of size $S \times S$ or $S \times S^s$) are the segments loaded in L_{GPU}, both for $\langle II \rangle$, $\langle JI \rangle$ and $\langle IK \rangle$, $\langle JK \rangle$ elimination tasks (notice that the sizes are actually maximal sizes, to within truncature effects).

Incidentally, we also need to store in L_{GPU} the block R and the parts of B that correspond to stripes I and J (B being stored in PC memory, we will shuttle as well each of its parts between the PC and the GPU).

Assuming that the required stripes are already in L_{PC} cache, Fig. 13 presents an overview of the triangulation scheme with L_{PC}–L_{GPU} transfers of the stripe parts. Again assuming column-major order, the pointers to the diverse blocks within the stripes are obtained trivially with basic pointer arithmetic.

For now, this scheme is described as sequential. Actually, it will be implemented in a multithreaded manner in Sect. 8.

Finally, Fig. 14 details the actual pseudocode for elimination over block $\langle JK \rangle$ in Fig. 13—derived from that of Fig. 5 to integrate stripe structures and stripe segments—as well as its implementation with *cublasDgemm* function.

7.2 Implementation of L_{PC} cache memory

The L_{PC} cache is implemented as an associative memory of cells that represent the currently cached parts of stripes and as a vector V containing the numerical values of these stripe parts. In Fig. 15, we detail the storage in L_{PC} of a stripe part I and of a stripe part J: each stripe part is represented by a cell in associative memory and its numerical values are stored contiguously within V in the range $[p_0,\ p_1[$. The interest of this approach is that it allows us to implement in essence a simple dynamic memory that may later adapt readily to other matrix algorithms.

The associative memory is dimensioned to contain as many cells as there are stripes (N), i.e. up to 200 for $n = 400{,}000$ (assuming $S = 2{,}000$). Also, notice that a great number of dynamic allocations must be performed within V for all stripe parts. However, these allocations are facilitated by the fact that the sizes of the stripe parts keep decreasing over the elimination process, thereby erasing any critical need for a defragmentation.

The cell stores index I (or J) and pointers p_0, p_1 as well as i_0^{store}, column index indicating the beginning of the stripe part. Notice that, although i_0^{store} may coincide initially with i_0 of I, it happens very often that a J stripe is first loaded into L_{PC} cache for elimination using I, then remains in cache until further eliminations using $I + 1$, $I + 2, \ldots$ are performed over it. In such a case, the storage-related index i_0^{store} is different (strictly inferior) from the elimination-related index i_0 of I.

Since every J stripe requires an I stripe for a given partial elimination, the cell of a J stripe must also store its corresponding elimination index I. Further, an I stripe part must remain available, "stuck" in L_{PC} cache (i.e. it must not be allowed to be uploaded back into L_{SSD}) as long as there are some J stripes still requiring it for their partial elimination. To that effect, the cell of I stores the counter *count* that contains

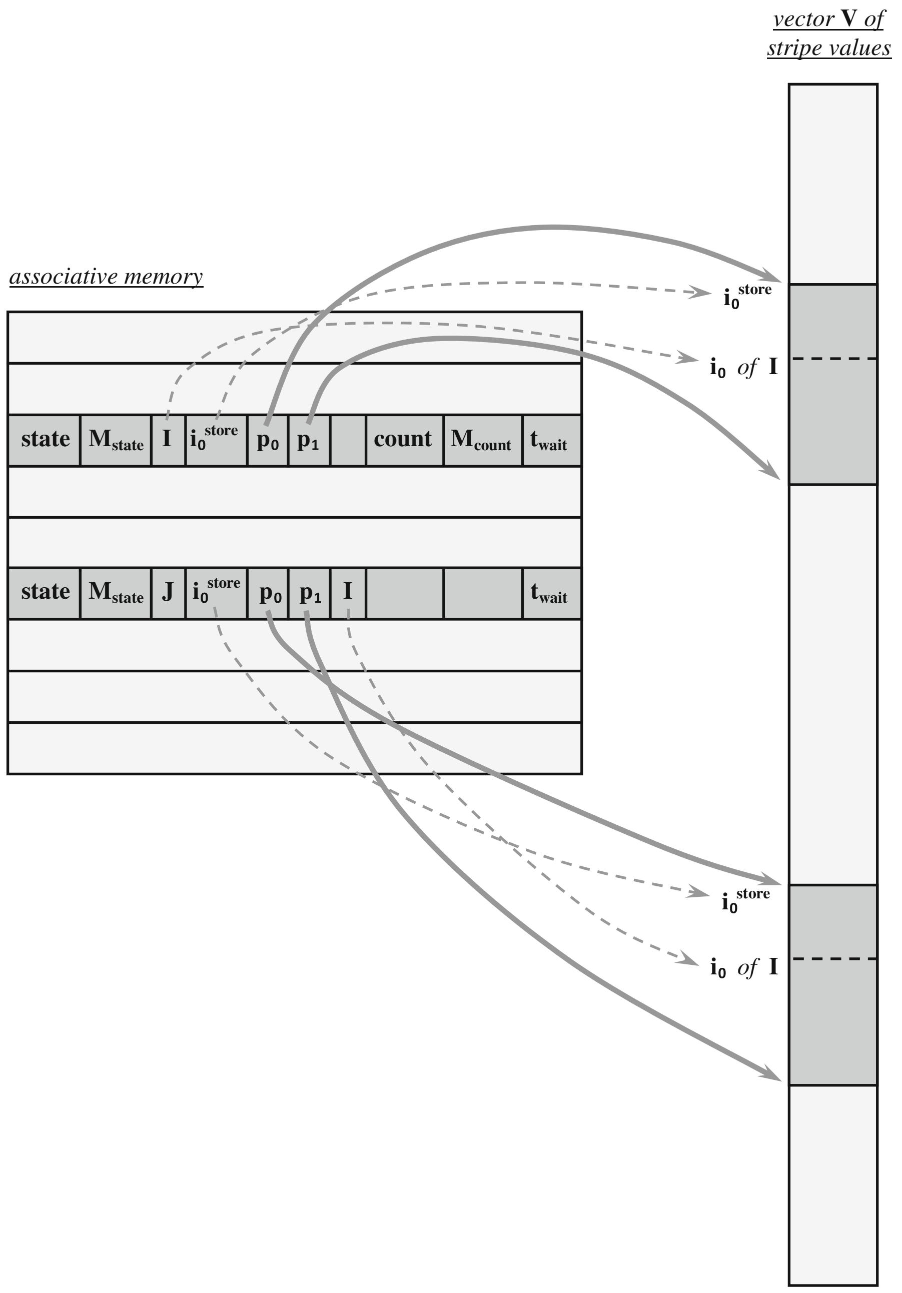


Fig. 15 L$_{PC}$ cache memory: associative memory and vector of stripe values

the number of such *J* stripes. *count* is then decremented each time the elimination step has been performed on one of these *J* stripes. Ultimately, when *count* reaches 0,

I stripe part becomes “unstuck”, i.e. it is liable to upload into $\mathrm{L_{SSD}}$ (as soon as $\mathrm{L_{PC}}$ will require space for other stripe parts).

Further, the cell of a stripe part must contain its current state. *state* describes the operation currently underway over the stripe part or about to be applied to it (see Sect. 8.3). Also, the multithreaded approach of Sect. 8 requires that the cell contains as well the mutexes M_{state} and M_{count} to ensure mutual exclusion locks over *count* and over the stripe operation described by *state*. Finally, the cell stores t_{wait}, time since the stripe part has been waiting idly in $\mathrm{L_{PC}}$ (to implement the cache flushing policy, see Sect. 8.3).

7.3 Storing matrix stripes within SSD array

In order to maximize the transfer rate between $\mathrm{L_{SSD}}$ and $\mathrm{L_{PC}}$, we implement a data striping mechanism over the array of six SSDs to allow parallel access (up to six times faster): each matrix stripe is subdivided into segments (unrelated to those defined in Sect. 7.1) that are interleaved in round-robin fashion over the six SSDs. The size of these segments is adjusted such that interleaving always occurs over eight levels exactly, as described in Fig. 16. Notice that a stripe segment is stored as a file in an SSD and that each SSD consequently contains $8 \times N$ files for all N stripes.

The reason why we chose eight levels of interleaving has to do with the size of the $\mathrm{L_{PC}}$ cache. Our PC being equipped with 24 GiB of RAM, we fixed the cache size so as to contain a maximum of 2.5×10^9 double-precision numerical values. Practically, it means that a matrix of size up to 50,000 can be stored in its entirety in the $\mathrm{L_{PC}}$ cache. This is to be compared with our overarching goal which is to solve a dense linear system of size up to 400,000 (representing 1,280 GB of files in the SSDs).

Assuming n as high as 400,000, it implies that toward the end of the triangulation, more precisely just before the submatrix remaining to be eliminated (see Fig. 2) is small enough to be entirely within $\mathrm{L_{PC}}$ cache, i.e. before its size is down to 50,000,

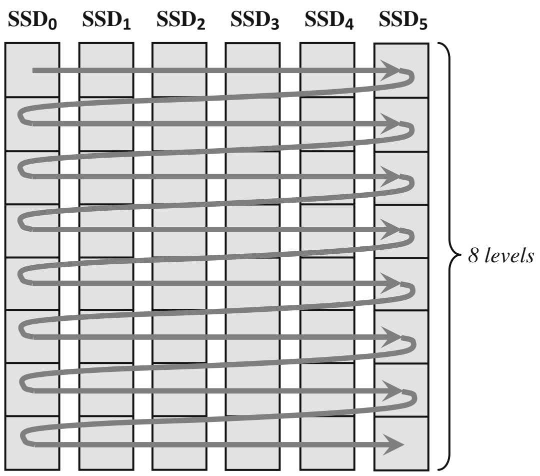


Fig. 16 Interleaving segments of a matrix stripe in round-robin fashion (8 levels) over six SSDs

then there is still one complete round-robin level (since 400,000/8 = 50,000) to ensure a fully parallel access to all six SSDs even for the very last transfers of stripe parts.

At this point, We need to address a truncature issue that occurs because stripe parts of varying size are accessed over the eight levels of interleaving. In the least bad case, a stripe part required for triangulation may be on $6 \times 7 + 1$ segments, then the transfer takes 8/7 = 1.14 times longer than if it were on just 6×7 segments. The actual worst case occurs when the stripe part is on just 6 + 1 segments instead of 6, in which case the transfer takes twice longer...

Consequently, so as to lessen this truncature effect in most cases, we keep a fixed number of eight round-robin levels even for matrices of size (much) smaller than 400,000 (so that the worst case keeps happening relatively infrequently). Of course, it implies that the size of the segments varies in function of n, from 127 MB for n = 400,000 down to 15.9 MB for n = 50,000 (assuming S = 2,000), but this still represents an appreciable granularity for SSD transfers.

Notice that there exists an obvious similarity with RAID 0, and we did consider using this technology. But we eventually decided to develop our own round-robin system and enclose it within our code to remain independent from any specific RAID controller and from the limitations that may come with such or such controller and motherboard.

For example, our motherboard (Gigabyte X58A UD7) possesses as many as ten SATA ports, which should allow an array of up to nine SSDs, in addition to the PC's hard disk. However, up to three SATA controllers (Intel ICH10R, Gigabyte SATA2 and Marvell 9128) impose separate RAID 0 arrays with sizes of 6, 2 and 2, respectively. Using RAID0 (with ICH10R) would thus limit us to our current six SSDs, whereas it is technically possible to add an extra three SSDs. Further, it is not recommended to use SATA3 SSDs in RAID 0 mode with the Marvel 9128 controller.

In addition, our system allows a dynamic adjustment of the size of segments (preferable for the reasons exposed above).

8 Dynamic scheduling for optimized use of PC-based cache

8.1 Dynamic scheduling of elimination tasks over matrix stripes

Tile algorithms were originally developed for parallel execution over traditional multicore architectures [12,20]. The low granularity of the treatment of each tile was indeed better suited to the limited core caches. But, assuming that the processing unit for elimination tasks is now no longer a traditional core but a whole GPU with much more memory, then a stripe partition becomes possible.

Further, since the elimination tasks are now applied over stripes instead of tiles and are thus of much higher granularity, their dynamic scheduling is greatly simplified. Indeed, the Directed Acyclic Graph (DAG) [4,9,12,15,23] that represents the dependencies between elimination tasks over stripes is simpler, much smaller and, in the case of Gauss triangulation, highly predictable, as Fig. 17 shows for $N = 5$ stripes.

For each value of I, a single elimination task is applied over stripe I, then $N - (I + 1)$ elimination tasks over all stripes J such that $J > I$ are executed independently from

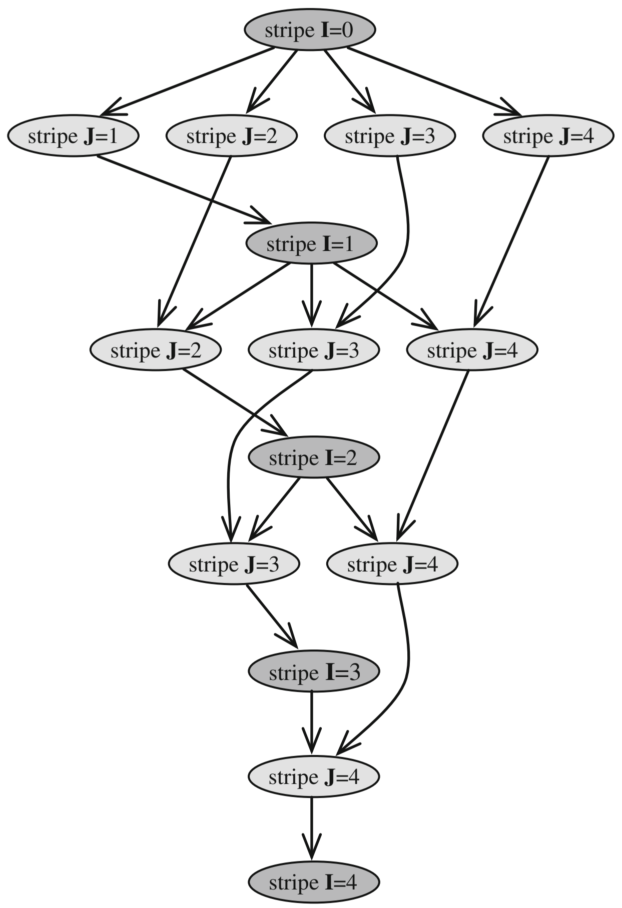


Fig. 17 Directed acyclic graph (DAG) of Gauss triangulation stripe algorithm for $N = 5$ stripes (dependencies between elimination tasks over I stripes and over J stripes)

each other, i.e. potentially in parallel. Once these tasks are over, the same process is repeated with the next value of I.

Actually, it is possible to desynchronize to a certain extent this rigid pattern. Indeed, it is enough to wait for only one of the previous tasks on J, i.e. the one over $J = I + 1$, to terminate to be able to start the next task over $I + 1$. Then, after this task over I is completed, each of the following tasks over J has just to wait for the corresponding previous task over J to terminate.

8.2 PC threads to supervise SSD transfers and GPU-based elimination tasks

We have developed a fully multithreaded PC-based code to supervise in parallel the global tasks of communication with the SSDs and of elimination over the GPU. Figure 18 shows the organization of these threads.

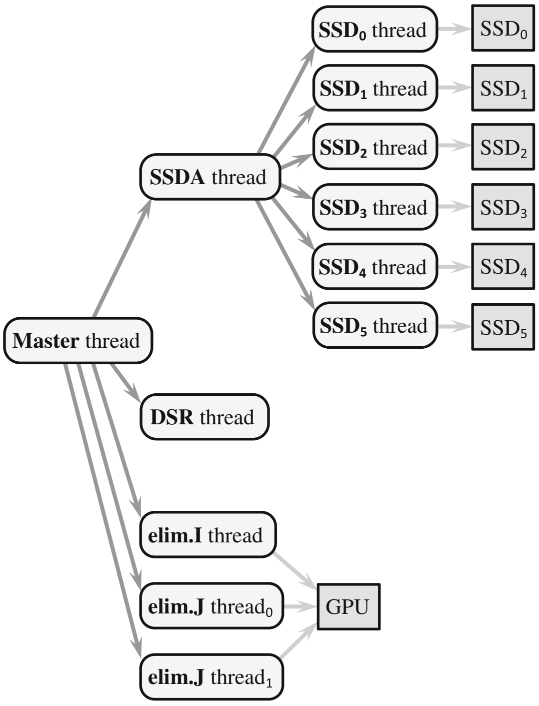


Fig. 18 Organization of PC threads

Along with the *Master* thread running the main code, a dedicated thread supervises the interleaved data transfers with the SSD Array (in accordance with Sect. 7.3). Named *SSDA* thread, it controls SSD$_0$, SSD$_1$, ..., SSD$_5$ threads that are actually in charge of carrying out in parallel input–output operations over the individual SSDs (following the data striping mechanism of Sect. 7.3).

Concerning the elimination tasks over stripe parts, three PC threads are used to direct three concurrent elimination tasks within the GPU: *elim.I* thread directs an elimination task over an I stripe part whereas both *elim.J* thread$_0$ and *elim.J* thread$_1$ direct elimination tasks over J stripe parts (notice that the decomposition of matrix stripes into segments of maximal length $\mathbf{S}^s$ is performed within these elimination threads and is not visible at the dynamic scheduling level).

That way, we mask the overhead cost of PC—GPU data transfers as well as the cost of swapping parts of rows by the PC (for local partial pivoting, see Sects. 5 and 6). Further, the elimination over the next I stripe part can be started asynchronously while some eliminations over J stripe parts using the previous I are still under way (in accordance with Sect. 8.1).

Also, in addition to PC—SSD transfers for the triangulation phase that requires both reading and updating stripes for asynchronous elimination threads, we need for subsequent matrix operations that are directly controlled by the *Master* thread, such as the resolution of the triangular system and the multiplication of the initial matrix with the vector solution (to calculate the residual error), to download a group of stripes for reading only.

Such a flow of stripes must then be downloaded asynchronously with the *Master* thread so as to mask to a certain extent the overhead transfer time. To that avail, the so-called *DSR* thread is used to Download Stripes in a given index Range. Notice that, since these stripes are meant to be read only, they are first labeled as “stuck” in L_{PC} cache (i.e. non flushable). Then, once the *Master* thread has read them, it immediately and directly deletes them from the cache.

All in all, we have a total of 12 PC threads. However, the *DSR* thread is not active during the triangulation phase, whereas elimination threads are inactive during triangular system resolution and matrix–vector multiplication phases. Accordingly, 11 threads are active during the triangulation phase and 9 threads only are active during subsequent phases.

Considering that our PC is equipped with a single hexacore Core i7 microprocessor that offers direct hardware support for up to 12 threads only (with hyper-threading), it appears that our multithreaded system fits neatly within the hardware capabilities (partial experiments did show that adding just one extra thread for *J* elimination led to the complete collapse of the system’s performance). In practice, while the code is using 11 threads, there is still hardware support for one more thread to run the PC’s OS and other tasks (word processing, web browsing,…). These tasks then run relatively slowly, whereas our code’s performance remains virtually unchanged.

Notice that it is possible to limit the number of SSD_x threads by partially sequentializing access to the SSDs, with one thread accessing two (or 3) SSDs in sequence. But that would decrease the bandwidth by a factor 2 (or 3), although Sect. 9.8 results indicate that it should not be too much of a concern.

8.3 N-plicated multithreaded finite-state machine

Every stripe part present in L_{PC} cache may be subject to the following operations: its numerical values may be downloaded from L_{SSD} to be either just read or updated with an *I*-type or *J*-type elimination. Or else the stripe part may be uploaded into L_{SSD} to free space in the cache.

In order to keep track of these operations (or absence of), the cell of the associative memory that represents the stripe part (see Sect. 7.2) contains its current state, defined as either the operation that it is waiting for, or the operation that is currently being performed over it, or else the absence of operation.

If indeed no operation is about to be or is currently applied to it, then the stripe part may be waiting idly in L_{PC} and is liable at any time to be flushed out to L_{SSD}. Or it may be “stuck waiting” in L_{PC}, i.e. it will either remain “unflushable” until it is explicitly unstuck or it will remain in L_{PC} until it is explicitly deleted. That is, in case the stripe part is definitely going to be needed either as *I* stripe for a *J*-type elimination or for triangular system resolution/matrix–vector multiplication.

Notice that if there is no cell in associative memory to represent the stripe part, then its state is implicitly defined as “waiting in L_{SSD}”.

We may represent conceptually the evolution of the states of each stripe part in L_{PC} with a multithreaded finite-state machine whose state diagram is detailed in Fig. 19.

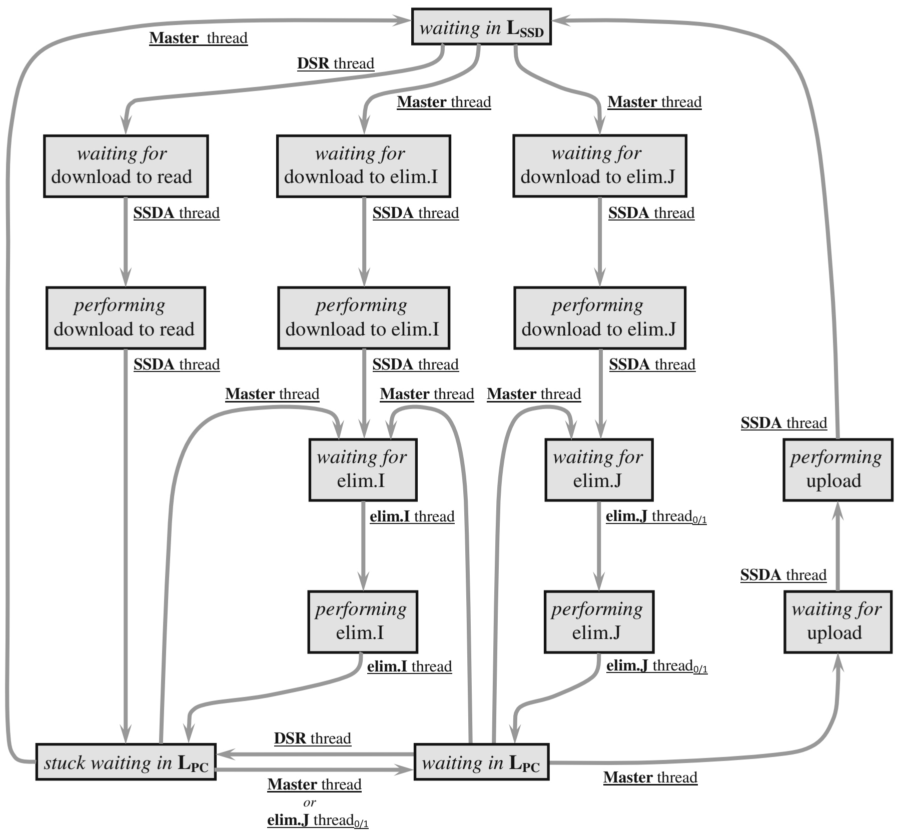


Fig. 19 State diagram for multithreaded state machine

We denote this finite-state machine as multithreaded because all six main threads—namely the *Master* thread, *SSDA* thread, *DSR* thread, *elim.I* thread, *elim.J* thread$_0$ and *elim.J* thread$_1$—carry out concurrently the state transitions (see Fig. 19) and the SSD transfer operations and elimination operations that may be associated with these transitions. Further, we globally see it as an "N-plicated multithreaded finite-state machine" since it is "N-plicated" and implemented simultaneously over all stripe parts present in L_{PC}. For example, the *SSDA* thread may be downloading a stripe part from L_{SSD}, whereas, at the very same moment, the three elimination threads are supervising elimination tasks on three other stripe parts.

Notice that, due to the inherent concurrency between the threads, it is of course crucial to implement mutual exclusion locks over not only the changes of states but also globally over their (eventual) associated operations. For that purpose, we use the mutex M_{state} present in each cell (see Fig. 15).

We may now follow the main paths of the state diagram of Fig. 19.

1. In case a stripe part must be applied an *I*-type elimination, two sub-cases are to be considered:

   1.1. *the stripe part is waiting in* L_{SSD}: it will first wait to be downloaded in L_{PC}, then its download will be performed; after that, it will wait for *I*-type elimination and then this elimination will be performed; finally, the stripe part will end up stuck waiting in L_{PC}.
   1.2. *the stripe part is either stuck waiting or simply waiting in* L_{PC}*(as a previous Jstripe*): it will directly wait for *I*-type elimination and then this elimination will be performed; finally, it will end up stuck waiting in L_{PC}.

   After both sub-cases are completed, the stripe part must remain stuck in L_{PC} since it is needed for subsequent *J*-type eliminations. Then, once *elim.J* $\text{thread}_{0/1}$ has finished the last of them, this thread will realize that *count* of the cell of the *I* stripe part (see Sect. 7.2) has been decremented to 0 and will consequently unstick this stripe part. Actually, for the last stripe $I = N - 1$, since there are no subsequent *J*-type eliminations, it is the *Master* thread that unsticks it.
2. In case a stripe part must be applied a *J*-type elimination, there are two sub-cases similar to those above:
   2.1. *the stripe part is waiting in* L_{SSD}: it will first wait to be downloaded in L_{PC}, then its download will be performed; after that, it will wait for *J*-type elimination and then this elimination will be performed; finally, the stripe part will end up waiting in L_{PC}.
   2.2. *the stripe part is waiting in* L_{PC}: it will directly wait for *J*-type elimination and then this elimination will be performed; finally, it will end up waiting in L_{PC}.
3. In case the *DSR* thread requests a stripe part, there are three sub-cases:
   3.1. *the stripe part is waiting in* L_{SSD}: it will first wait to be downloaded in L_{PC}, then its download will be performed and it will end up stuck waiting in L_{PC}.
   3.2. *the stripe part is waiting in* L_{PC}: it will become stuck waiting in L_{PC}.
   3.3. *the stripe part is already stuck waiting in* L_{PC}.

   After all three sub-cases are completed, the stripe part is read for triangular system resolution/matrix–vector multiplication and explicitly deleted from L_{PC} by the *Master* thread.
4. In case the stripe part has been waiting idly in L_{PC} for too long (actually, if it is the one that has been waiting the longest, based on t_{wait} of its cell, see Sect. 7.2) and if the L_{PC} cache is short of memory, then the stripe part will be flushed out to L_{SSD}: it will first wait for upload, then the upload will be performed and the stripe part will finally be deleted from L_{PC}.

8.4 Strategies to optimize PC-based cache usage

The triangulation scheme was described in Sect. 7.1 and Fig. 13 as a sequential code. We now modify it into a multithreaded code to integrate the multithreaded approach described in Sects. 8.2 and 8.3. Actually, we provide in Fig. 20 three versions of it that correspond to different strategies that we have designed specifically—taking into account the inner workings of our stripe algorithm—to optimize the usage of the L_{PC} cache.

Fig. 20 Pseudocodes of basic, zigzag and meta-stripes-zigzag strategies for optimization of L_{PC} cache usage

basic strategy:

```
triangulation
  for I ← 0 , … , N - 1 :
      start elim. over I
      if I < N - 1 :
          for J ← I + 1 , … , N - 1 :
              start elim. over J using I
```

zigzag strategy:

```
triangulation
  for I ← 0 , … , N - 1 :
      start elim. over I
      if I < N - 1 :
          if I even :
              for J ← I + 1 , … , N - 1 :
                  start elim. over J using I
          else
              for J ← N - 1 , … , I + 1 :
                  start elim. over J using I
```

meta-stripes-zigzag strategy:

```
triangulation
  $N^m$ ← number of meta stripes
  for $I^m$ ← 0 , … , $N^m$ - 1 :
      $I_0 \leftarrow I^m \times N / N^m$ ; $I_1 \leftarrow (I^m + 1) \times N / N^m$

      for I ← $I_0$ , … , $I_1$ - 1 :
          start elim. over I
          if I < N - 1 :
              for J ← I + 1 , … , $I_1$ - 1 :
                  start elim. over J using I

      if $I^m < N^m - 1$ :
          if $I^m$ even :
              for J ← $I_1$ , … , N - 1 :
                  start multiple elim. over J using I ∈ [ $I_0$ ; $I_1$ [
          else
              for J ← N - 1 , … , $I_1$ :
                  start multiple elim. over J using I ∈ [ $I_0$ ; $I_1$ [
```

8.4.1 Basic strategy

The most basic version of the triangulation code corresponds to the so-called "basic strategy": the *Master* thread executes the code which starts the I-type and J-type eliminations by initiating the paths for eliminations present in Fig. 19. To that avail, the *Master* thread changes the states of the stripes and, by doing so, urges the *SSDA* thread and the three elimination threads to proceed along the paths.

Notice that, due to the inherent asynchronicity of the code, both I and J types of elimination can be carried out simultaneously. Indeed, the I-type elimination of the next step can start before all J-type eliminations of the current step are finished (to within the size of the L_{PC} cache), thus implementing the possibility outlined at the end of Sect. 8.1.

In practice, for such a situation to occur, the cache should be able to store two I stripes (of the current and next step) and at least two J stripes (for both *elim.J* thread$_0$ and *elim.J* thread$_1$ to work concurrently over the GPU). Therefore, the L_{PC} cache should be able to store at least four of the initial stripes of full length n to allow the code to perform optimally.

8.4.2 Zigzag strategy

The "zigzag strategy" is an improvement over the basic strategy in that the code now follows a zigzag pattern to start the J-type eliminations: for the first elimination step, the J stripes are scanned downward; then, for the second elimination step, they are scanned upward; then downward again, and so on.

By doing so, the J stripe parts whose index J is close to either $I+1$ or $N-1$ are more likely to be applied two elimination tasks while still in L_{PC} cache, hence diminishing to a certain extent the number of transfers of stripe parts between L_{SSD} and L_{PC}. However, the interest of this strategy can only become marginal as the matrix size becomes much bigger than the L_{PC} cache size (the proportion of J stripe parts remaining in cache over two elimination tasks decreasing accordingly).

8.4.3 Meta-stripes-zigzag strategy

In order to apply more elimination tasks to all stripe parts while they reside in L_{PC} cache, we now apply a formal code transformation, similar to those of Sects. 3 and 4, to the code of the zigzag strategy.

Just as we defined a stripe as a group of at most S rows (and a substripe as a group of at most S' rows), we define a "meta stripe" as a group of at most S^{m} stripes (this paper's experiments have been conducted with only $S^{\mathrm{m}} = 2$). Then the total number of meta stripes is

$$N^{\mathrm{m}} \leftarrow \begin{cases} N/S^{\mathrm{m}} & \text{if } S^{\mathrm{m}} \text{ divides } N \\ N/S^{\mathrm{m}}+1 & \text{else} \end{cases} \tag{9}$$

and the index I^{m} of the meta stripe has range $[0, N^{\mathrm{m}}[$. Also, the range of a stripe index within meta stripe I^{m} is $[I_0, I_1[$, with $I_0 \leftarrow I^{\mathrm{m}} \times N/N^{\mathrm{m}}$ and $I_1 \leftarrow (I^{\mathrm{m}}+1) \times N/N^{\mathrm{m}}$ (using integer arithmetic).

We now replace the loop over I in the zigzag code with two nested loops: an outer loop over the meta stripes and an inner loop over the stripes within each meta stripe:

$$\textbf{for } I \leftarrow 0, \ldots, N-1 : \big[\ldots \quad \longrightarrow \quad \begin{array}{l} \textbf{for } I^{m} \leftarrow 0, \ldots, N^{m}-1 : \\ \big[\; I_0 \leftarrow I^{m} \times N / N^{m} \;\; ; \;\; I_1 \leftarrow (I^{m}+1) \times N / N^{m} \\ \quad \textbf{for } I \leftarrow I_0, \ldots, I_1 - 1 : \big[\ldots \end{array} \tag{10}$$

Then, we replace the loop over J from $I + 1$ to $N - 1$ (or $N - 1$ to $I + 1$) into a loop from $I + 1$ to $I_1 - 1$ and a loop from I_1 to $N - 1$:

$$\textbf{for } J \leftarrow I+1, \ldots, N-1 : \big[\ldots \quad \longrightarrow \left\{ \begin{array}{l} \textbf{for } J \leftarrow I+1, \ldots, I_1 - 1 : \big[\ldots \\ \\ \textbf{for } J \leftarrow I_1, \ldots, N-1 : \big[\ldots \end{array} \right. \tag{11}$$

Further rearrangement yields the "meta-stripes-zigzag strategy" code of Fig. 20. The crucial point is that, in the general case, J-type elimination tasks related to the I stripes of meta stripe I^{m} are now executed in a row by *elim.J* thread$_{0/1}$, i.e. with a loop over I from I_0 to $I_1 - 1$. Of course, it implies that I_0 and I_1 index values must now be stored within the cell of a J stripe (in place of the single I value).

The obvious advantage is that the overall number of $\mathrm{L_{SSD}}$–$\mathrm{L_{PC}}$ transfers of part stripes is reduced by a factor S^{m} or so. However, to achieve optimal performance, the $\mathrm{L_{PC}}$ cache must now be large enough to store all S^{m} stripes of the meta stripe, in addition to at least two J stripes (for *elim.J* thread$_0$ and *elim.J* thread$_1$).

Ideally, to be able as well to start the eliminations over the next meta stripe before the end of all current J-type eliminations (as a generalization of the end of Sect. 8.4.1), we need to store an extra S^{m} stripes in $\mathrm{L_{PC}}$ cache (we admit that the advantage might be somewhat marginal compared with the memory cost).

Overall, good performance requires the $\mathrm{L_{PC}}$ cache to store at least $S^{\mathrm{m}} + 2$ full-length stripes (i.e. 4 if $S^{\mathrm{m}} = 2$). However, peak performance requires the $\mathrm{L_{PC}}$ cache to store at least $2S^{\mathrm{m}} + 2$ full-length stripes (i.e. 6 if $S^{\mathrm{m}} = 2$).

Notice that, whether the cache stores $S^{\mathrm{m}} + 2$ or $2S^{\mathrm{m}} + 2$ stripes of full length n, its required size is bound to grow only linearly with n, whereas the matrix size grows quadratically with n. Consequently, the required cache size will grow much more slowly than the matrix size. In practice, if we want to upgrade our computer system to solve bigger dense linear systems, then the PC's RAM will have to be expanded much more gradually than the SSDs' capacity: each time the SSDs' capacity will be multiplied by four, the PC's RAM should be doubled only.

9 Experiments

9.1 Experimental conditions

The hardware configuration used for the experiments consists in a PC running Windows 7 with an Intel Core i7-980X hexacore at 3.33 GHz, a Gigabyte X58A UD7 motherboard and 24 GiB of DDR3 RAM.

This PC hosts two Tesla C2050 GPUs with 3 GiB of DDR5 RAM (ECC mode being disabled). All experiments are implicitly performed on only one C2050, except in Sect. 9.3.2 where the configuration with two C2050s is put to the test.

Also, the PC is equipped with six Kingston HyperX 3K SSDs with 240 GB each, for a total of 1,440 GB. The SSDs are plugged to the motherboard through six SATA 3 Gb/s connectors.

In order to conduct truly general experiments, we generate randomly a number of dense linear systems: for each system, the values of the elements of matrix A and vector B are chosen randomly with a uniform distribution (similarly to [1]) in range $[-0.5; +0.5]$. Notice that, as far as residual errors are concerned, the amplitude of this interval does not really matter since it is applied over both A and B elements and since floating-point representation of numerical values is used.

We will use in the following sections a sequence of random dense linear systems ranging in size from $n = 25{,}000$ to 400,000.

At this point, we need to expose the methodology followed for our presentation: we first describe the parameter settings, taken as default settings, that we have finally chosen through lengthy trial and error experiments. Then, as an a posteriori—and partial—justification, we will test in the following sections some variations over some of these settings.

The default settings correspond first to the choice of the meta-stripes-zigzag strategy with the maximal size of a meta stripe taken as $S^{\mathrm{m}} = 2$. Then, the L_{PC} cache size is chosen to contain up to 2.5×10^9 numerical values. Consequently, we organize eight levels of interleaving over the six SSDs (see Sect. 7.3). Also, the segments of stripes are of maximal length $S^{\mathrm{s}} = 20{,}000$, whereas the maximal size of a stripe of rows is $S = 2{,}000$ and that of a row substripe is $\mathbf{S}' = 200$.

Finally, notice that all the results reported in this paper correspond to double-precision floating point arithmetic calculations. Also, we will present separately the performances of the triangulation phase and of the triangular system resolution.

9.2 Experiments on stripes and substripes

Our first experiments consist in varying—from the default settings—the maximal stripe size S around value 2,000, along with the substripes optimization being either enabled or disabled. Figure 21 shows, for a linear system of size $n = 200{,}000$, the performance in Gflops of the triangulation phase in function of S without and with row substripes optimization ($\mathbf{S}' = 200$).

It appears clearly that the substripes optimization substantially boosts the performance. With it, the performance peaks at 199.07 Gflops for $S = 2{,}000$ (which justifies our choice of this value).

Notice that the choice of the value of $\mathbf{S}'$ also influences the local partial pivoting scheme (since it is performed within a substripe, see Sect. 5). Accordingly, we will justify in Sect. 9.4 the choice of $\mathbf{S}' = 200$ primarily in relation with our pivoting scheme.

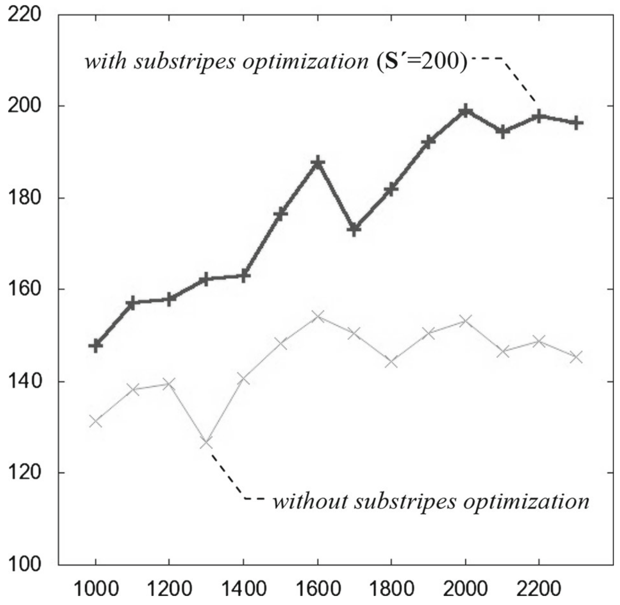


Fig. 21 With one C2050, performance (in Gflops) of the triangulation phase in function of *S* (max. size of stripes) without and with row substripes optimization ($\mathbf{S}' = 200$) for linear system size $n = 200{,}000$

9.3 Experiments on strategies for L_{PC} cache optimization

9.3.1 Using one C2050 GPU

We now compare the performances obtained for the triangulation phase with each of our three optimization strategies for L_{PC} cache usage. Figure 22 shows the curves for the three strategies of the performance in Gflops (with one C2050) in function of the linear system size *n* (ranging from 25,000 to 400,000).

From size 25,000 to 50,000, the three curves are obviously quasi-identical since the linear system is then entirely stored within the L_{PC} cache, thereby avoiding any L_{SSD}–L_{PC} transfers. Beyond size 50,000, the performance with basic strategy remains relatively constant within 154.2–168.7 Gflops.

As we had predicted in Sect. 8.4.2, the zigzag strategy first offers substantially more performance with a peak of 182.1 Gflops at size 100,000, but then the performance gradually diminishes until it rejoins that of basic strategy around size 350,000.

As for the meta-stripes-zigzag strategy, its performance grows steadily until a peak of 201.0 Gflops at size 250,000 and a following slight decrease until size 300,000. Then, beyond that, it decreases markedly until it reaches a low of 149.4 Gflops at $n = 400{,}000$, definitely below the first two strategies.

The analysis at the end of Sect. 8.4.3 helps to explain the curve of the meta-stripes-zigzag strategy. We first calculate that it is possible to store up to six full stripes in L_{PC} cache until $n = 208{,}333$ (since we then have $n \times 2{,}000 \times 6 = 2.5 \times 10^9$). Similarly, up to four full stripes can be stored in cache until $n = 312{,}500$.

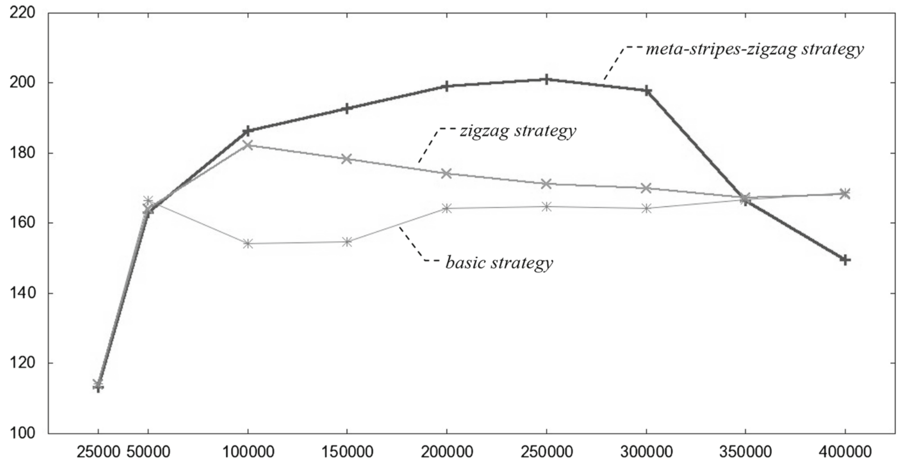


Fig. 22 With one C2050, performance (in Gflops) of the triangulation phase in function of n (linear system size) for basic, zigzag and meta-stripes-zigzag optimization strategies of L_{PC} cache usage

Overall, according to Sect. 8.4.3 analysis, the L_{PC} cache optimization is excellent until $n = 208{,}333$ and remains good until $n = 312{,}500$. Beyond that, we should expect a serious drop in performance since there is not enough place in L_{PC} cache to store simultaneously the two I stripes of the meta stripe and the two J stripes (required to mask the cost of PC-GPU data transfers and the cost of swapping row parts by the PC, see Sect. 8.2).

Tentatively, we may surmise that the meta-stripes-zigzag curve exhibits an upward trend which is gradually thwarted between size $n = 208{,}333$ and $n = 312{,}500$ by the fact that the cache usage is in that interval slightly sub-optimal and keeps degrading as more stripe parts of lower and lower length can no longer be stored in L_{PC} cache in groups of six at least. Hence the maximum at 250,000 as a result of these two opposing trends.

Assuming this explanation, then Sect. 8.4.3 analysis does indeed neatly explain the curve of the meta-stripes-zigzag strategy.

From a practical standpoint, it appears that we need more PC RAM to maintain optimal performance until $n = 400{,}000$, but then how much more? Six full stripes of length 400,000 correspond to 4.8×10^9 numerical values, amounting to an L_{PC} cache size of 35.7 GiB. Consequently, a PC equipped with 40 GiB or so of RAM should be amply sufficient. Unfortunately, the Gigabyte X58A UD7 motherboard cannot support more than 24 GiB of RAM.

9.3.2 Using two C2050 GPUs

Although the paper is primarily focused on a single-GPU configuration, we also present in Fig. 23 the triangulation performances for the three L_{PC} cache optimization strategies, but this time with two C2050s. Notice that, because our PC offers physical support for only 12 threads, we have been able to assign only one *elim.J* thread per

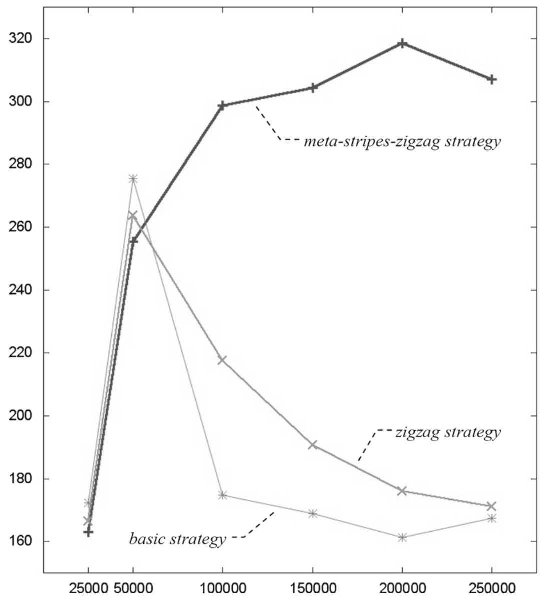


Fig. 23 With two C2050s, performance (in Gflops) of the triangulation phase in function of n (linear system size) for basic, zigzag and meta-stripes-zigzag optimization strategies of L_{PC} cache usage

GPU. Consequently, we should assume that the performances of Fig. 23 are slightly curtailed.

Comparing Figs. 22 and 23 reveals a striking difference with regard to the performance increase from the basic strategy up to the meta-stripes-zigzag strategy. Indeed, for n = 250,000, the increase is a mere 22 % with one C2050, whereas, with two C2050s, it is as much as 84 %. Obviously, more computing power requires a priori a higher PC-SSD bandwidth. The meta-stripes-zigzag strategy has then a more critical role to play in buffering out this increased bandwidth. As such, it is a vindication of our most evolved cache strategy.

9.4 Experiments on local partial pivoting

Our next series of experiments aims at studying the influence of our local partial pivoting scheme over both the numerical accuracy of the resolution and the performance in Gflops. To that avail, we define the residual error as the maximum of the absolute values of the elements of vector $AX - B$.

Figure 24 presents the residual errors (with logarithmic scale) in function of n, first without any pivoting scheme, then using our local partial pivoting scheme with $\mathbf{S}'$ = 50 and $\mathbf{S}'$ = 200. Indeed, since we localize partial pivoting within each substripe, $\mathbf{S}'$ defines the local pivoting range and has obviously an influence over the numerical accuracy. Notice that we also include in Fig. 24 the residual errors produced by a basic implementation of the code with full partial pivoting of Fig. 10, at least until n = 50,000 (due to the limited amount of RAM of our PC).

It appears that the absence of pivoting yields a solution of very poor quality, resulting in a loss of close to four orders of magnitude of accuracy with respect to the standard

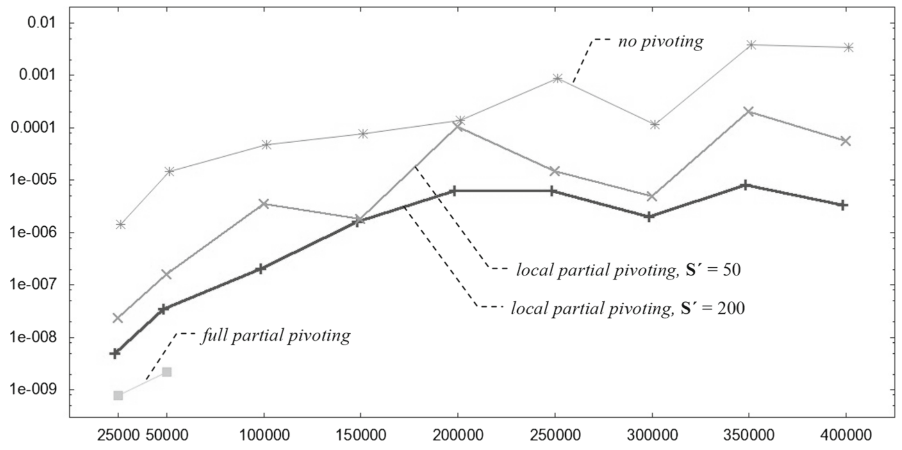


Fig. 24 Residual error in function of n (linear system size) without any pivoting scheme, then using our local partial pivoting scheme with $\mathbf{S}' = 50$ and $\mathbf{S}' = 200$. Results for basic sequential code with full partial pivoting (Fig. 10) also shown until $n = 50{,}000$

full partial pivoting code, at least until $n = 50{,}000$, and an unacceptable residual error of nearly 0.004 beyond $n = 300{,}000$.

Compared with the absence of pivoting, our local partial pivoting scheme does improve substantially the numerical accuracy. Also, a higher value for $\mathbf{S}'$ (i.e. a larger local pivoting interval) logically improves further the accuracy: the curve for $\mathbf{S}' = 200$ is definitely below that of $\mathbf{S}' = 50$ and is also more regular and predictable. In particular, it remains within the same order of magnitude beyond $n = 100{,}000$. By contrast, the curve for $\mathbf{S}' = 50$ becomes quite jagged beyond $n = 100{,}000$.

Overall, $\mathbf{S}' = 200$ consistently yields our best results, with a residual error below 10^{-5} for all values of n tested. Actually, since this error is an order of magnitude or so above the error with full partial pivoting for both $n = 25{,}000$ and 50,000, we wonder if it remains the same for higher values of n (in particular, do we still lose for $n = 400{,}000$ only one order of magnitude relative to full partial pivoting?).

It is quite possible that higher values of $\mathbf{S}'$ might improve further the accuracy. However, $\mathbf{S}'$ being as well the parameter of the substripes optimization, that might also end up degrading noticeably the overall performance in Gflops (and that might not even be worth it if the curve for $\mathbf{S}' = 200$ remains just an order of magnitude or so above that of full partial pivoting).

We now study the cost of our local partial pivoting scheme. Figure 25 exposes the curves of the performance in Gflops for the triangulation phase in function of n, first without any pivoting, then using our local partial pivoting scheme with $\mathbf{S}' = 50$ and $\mathbf{S}' = 200$.

Up to $n = 100{,}000$, local partial pivoting definitely takes a heavy toll on the performance. But as n increases, the cost of our pivoting scheme gradually dwindles until it becomes nearly negligible beyond $n = 250{,}000$. Further, there is a certain cost to increasing $\mathbf{S}'$ from 50 to 200, but it dwindles as well when n increases.

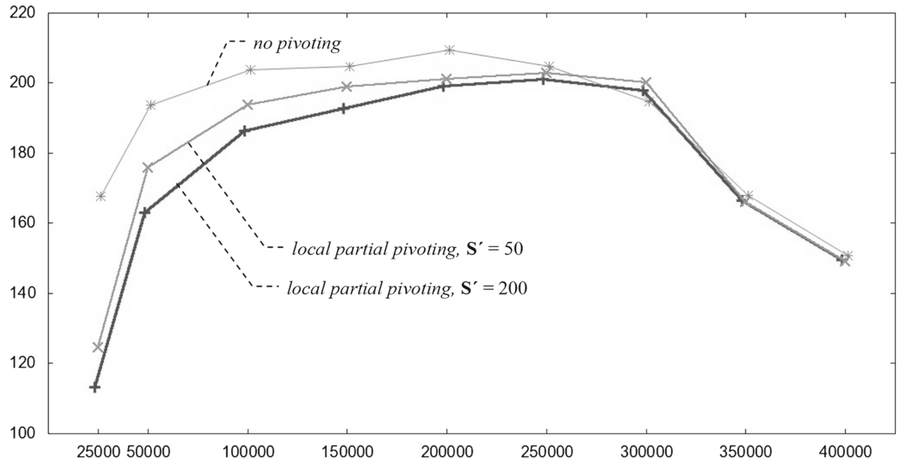


Fig. 25 With one C2050, performance (in Gflops) of the triangulation phase in function of *n* (linear system size) without any pivoting scheme, then using our local partial pivoting scheme with $\mathbf{S}' = 50$ and $\mathbf{S}' = 200$

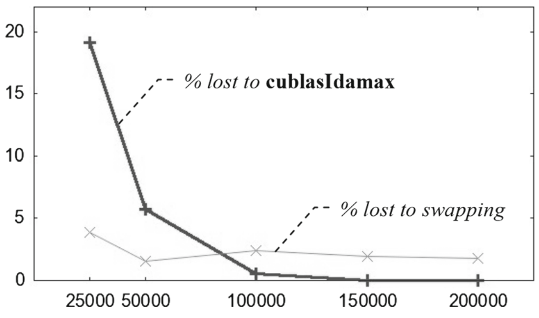


Fig. 26 With one C2050, percentages of reduction in performance due to *cublasIdamax* searches and to row swappings in function of *n* (linear system size)

In order to evaluate the respective costs of the two operations required for pivoting, that is the highest-magnitude pivot search performed with *cublasIdamax* and our explicit swapping of rows, we perform the following experiments.

Firstly, for $\mathbf{S}' = 200$, we duplicate the call to *cublasIdamax* and we measure differentially the Gflops lost to the search. Secondly, we perform row swapping operations two extra times (these two extra swappings neutralizing themselves) to measure, again differentially, the Gflops lost to swapping. Figure 26 exposes the percentages of reduction in performance due to *cublasIdamax* searches and to row swappings in function of *n*.

It appears that the cost of searches with *cublasIdamax* may be initially substantial, such as 19 % for $n = 25{,}000$, but it also happens to become negligible beyond $n = 50{,}000$. This result demonstrates the practicality of local partial pivoting within the range of a row substripe (see Sect. 5).

Further, the cost of our explicit swapping remains quite limited, on the order of 1–2 % beyond $n = 25{,}000$. As a matter of fact, swapping is often not performed explicitly for fear of its cost, but indirectly through a permutation vector. Unfortunately, this solution would have disrupted the inherently parallel structure of our code.

Actually, the reason why the cost of our explicit swapping remains limited is that the biggest (by far) parts of rows to be swapped reside only in PC memory and are swapped by the PC itself (see Sects. 5 and 6). And the multithreaded approach described in Sect. 7 does allow the PC's CPU to perform this task in parallel with the arithmetic calculations within the GPU, in effect masking to a great extent the cost of explicit swapping. Notice that [1] report using a GPU kernel to perform "all permutations within a block at once". Although that obviously allows a parallel implementation, we personally prefer to assign (most of) the swapping to the PC so as to devote entirely the GPU to actual arithmetic calculations.

In order to try to explain the curves of Fig. 26, we assume that all the elimination tasks are globally performed in $O(n^3)$, whereas all the local searches are done in $O(n)$ and the row swappings in $O(n^2)$. Therefore, in all due logic, the proportion of searching should diminish in $1/n^2$ and that of swapping should diminish in $1/n$. Consistently with that, Fig. 26 shows that the searches' cost decreases rapidly. However, the swappings' cost remains close to 2 %.

9.5 About the triangular system resolution

The triangulation phase being performed in $O(n^3)$ and the triangular system resolution in just $O(n^2)$, we may deduce that the cost of this second phase is negligible with respect to that of the triangulation. However, it also implies that the parallel implementation of the triangulation is much more efficient (see Sects. 2 and 3). So, how much negligible the relative cost of the triangular system resolution really is?

Figure 27 presents, in function of n, the performance in Gflops for the resolution of the triangular system as well as the ratio $\text{time}_{\text{triangulation}}/\text{time}_{\text{resolution}}$.

Until $n = 50{,}000$, i.e. as long as the whole system can be stored in L_{PC} cache, the performance is close to a disappointing 0.6 Gflops, a testament to the relative

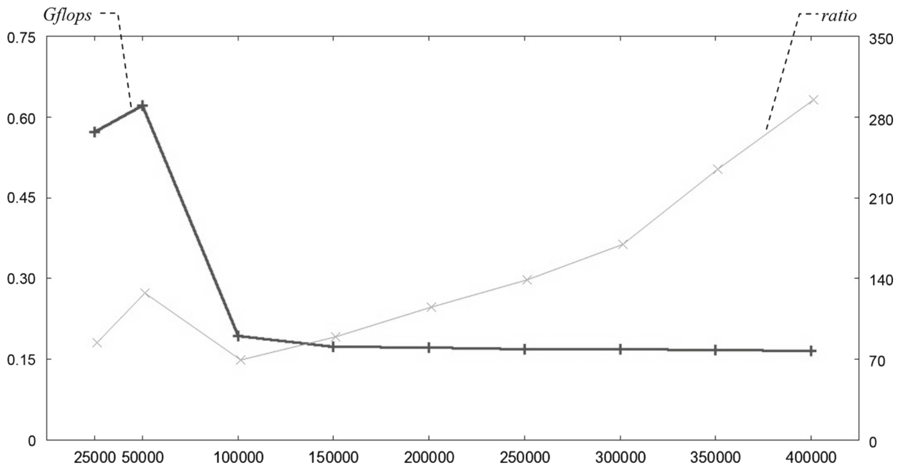


Fig. 27 Performance (in Gflops) for the resolution of the triangular system and ratio $\text{time}_{\text{triangulation}}/\text{time}_{\text{resolution}}$ in function of n (linear system size)

inefficiency of *cublasDgemv* which operates in O(n^2) only. Beyond n = 50,000, the performance is even worse, around 0.16 Gflops, due to the numerous L_{SSD}–L_{PC} stripe transfers required (the data transferred being in O(n^2) as well).

Nevertheless, Fig. 27 also shows that the triangulation phase, however efficient its parallel implementation is, still requires much more time than the triangular system resolution, from 69.5 up to 295 times more. And that ratio keeps increasing past n = 100,000.

9.6 Analysis of memory system performance

To offer a glimpse into the inner workings of our three-level memory system, and to probe further its efficiency, we carry out a limited experimental analysis of the activity of its main threads. We also consider the peak L_{SSD}–L_{PC} and L_{PC}–L_{GPU} data transfer rates.

Note that, for the sake of legibility of the following figure, we limit ourselves in this section to a small-scale experiment, i.e. n = 40,000 with an L_{PC} cache containing 5×10^8 values only.

We present in Fig. 28 the chronograms (over the whole triangulation phase) showing the periods of activity of *elim.I* thread, *elim.J* thread$_0$, *elim.J* thread$_1$ as well as the SSD to PC, PC to SSD, PC to GPU and GPU to PC data transfer activities.

It appears that there are far less L_{SSD}–L_{PC} transfer operations than L_{PC}–L_{GPU} transfer operations. This is the direct consequence of our L_{PC} cache and of its optimized usage. Further, after 250 s, there are no more L_{SSD}–L_{PC} transfers since all stripe parts that are still necessary now remain stored within the L_{PC} cache. We may also notice that the GPU is constantly kept busy with at least one active elimination thread (two most of the time).

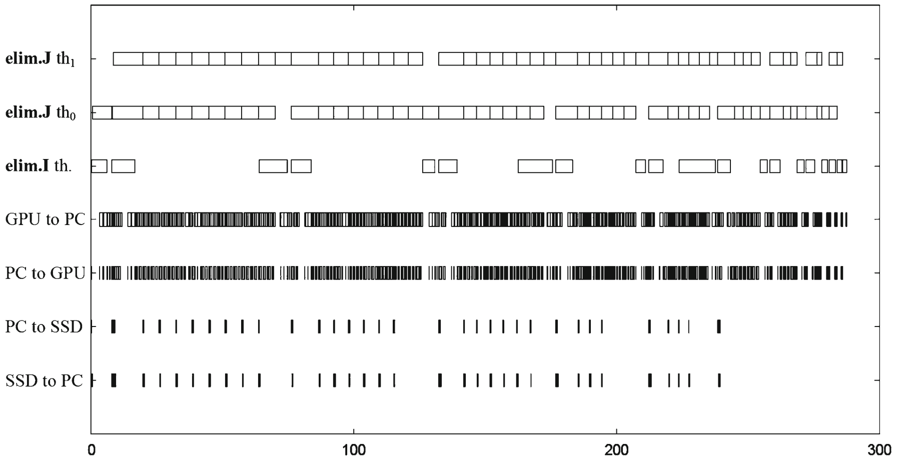


Fig. 28 For n = 40,000 and one C2050, with L_{PC} cache containing 5×10^8 values, chronograms over triangulation phase (time in seconds) of activities of *elim.I* thread, *elim.J* thread$_0$, *elim.J* thread$_1$ and of SSD to PC, PC to SSD, PC to GPU, GPU to PC data transfers

Concerning the L_{SSD}–L_{PC} and L_{PC}–L_{GPU} data transfers, we may consider the peak transfer rates based on hardware specifications.

Thus, the maximal transfer rates for the 240-GB Kingston HyperX 3K SSDs are 555 MB/s in reading and 510 MB/s in writing. However, these transfer rates are limited to no more than 375 MB/s by the transfer rate of a SATA 3 Gb/s connection. Overall, the parallel access to the six SSDs might ideally offer us a transfer rate of 375 MB/s $\times$ 6 = 2.25 GB/s. But, because of the truncature effect over the eight levels of interleaving (see Sect. 7.3), we should expect in the worst case a transfer rate as low as about 1 GB/s.

As for the L_{PC}–L_{GPU} data transfers, their maximal transfer rate ideally corresponds to that of the PCIe $\times$ 16 gen2 bus to which the GPU is plugged, i.e. 8 GB/s.

Overall, the bandwidth between L_{SSD} and L_{PC} is about four times lower than between L_{PC} and L_{GPU}. Thus, the L_{SSD}–L_{PC} data transfers constitute a potential bottleneck. Hence the necessity of our L_{PC} cache and of the meta-stripes-zigzag strategy to limit data transfers between L_{SSD} and L_{PC}.

9.7 Parameterizing the system to improve its portability

In order to facilitate the portability of our program to other future platforms, we now present a tentative parameterization of the system that includes parameters describing both hardware and program configurations. Table 1 lists the full set of parameters with, in particular, their mode of obtention as well as their actual values in our present configuration (with one C2050). Notation *NV* describes a capacity expressed as a number of double-precision floating point values (stored on 8 bytes). Notation *S* describes the size of a stripe, substripe or meta stripe expressed as a number of rows (or stripes).

The fixed hardware parameters correspond to given characteristics of the hardware. As for the program parameters, some had their values chosen in the previous sections to optimize performance (we expect them to remain relatively independent from the platform). The other parameters must be calculated based on certain relations and constraints.

Thus, the capacity of the array of SSDs is obviously calculated as follows:

$$\mathrm{NV}_{\mathrm{LSSD}} = \mathrm{NV}_{\mathrm{1SSD}} \times N_{\mathrm{SSD}} \tag{12}$$

The number of threads required for the elimination is obtained with the following formula (the first 1 corresponds to the master thread, the second to the SSDA thread and the third one to the single *elim.I* thread):

$$N_{\mathrm{th}} = 1 + (1 + N_{\mathrm{SSD}}) + (1 + N_{\mathrm{JthGPU}} \times N_{\mathrm{GPU}}) \tag{13}$$

A critical constraint is that all these threads, together with an extra thread (to keep running the PC's OS and other tasks), should not exceed the number of threads physically supported by the PC hardware:

$$N_{\mathrm{th}} + 1 \leq N_{\mathrm{thPC}} \tag{14}$$

Table 1 Hardware and program parameters of the system

Parameter name	Description	Mode of obtention	Value in our config. (one C2050)
Hardware parameters			
$\mathrm{NV}_{1\mathrm{SSD}}$	Capacity of a single SSD	Fixed	30×10^9
N_{SSD}	Number of SSDs	Fixed	6
$\mathrm{NV}_{\mathrm{LSSD}}$	Capacity of the array of SSDs	Calculated	180×10^9
$\mathrm{NV}_{\mathrm{LPC}}$	Capacity of the L_{PC} cache	Fixed	2.5×10^9
$\mathrm{NV}_{\mathrm{GPU}}$	Capacity of the GPU's global memory	Fixed	ECC off: 0.40×10^9 ECC on: 0.35×10^9
N_{GPU}	Number of GPUs hosted by the PC	Fixed	1
N_{thPC}	No. of PC threads physically supported	Fixed	12
Program parameters			
$n_{\max}$	Maximal matrix size	Calculated	Capacity: 424,264 good perf.: 312,500 peak perf.: 208,333
N_{JthGPU}	No. of *elim.J* threads per GPU	Chosen	2
N_{th}	No. of threads required for elim. phase	Calculated	11
S	Max. size of a stripe (see Fig. 21)	Chosen	2,000
$\mathbf{S'}$	Max. size of a substripe (Figs. 21, 24, 25)	Chosen	200
S^{s}	Max. length of stripe segment	Calculated	20,000
S^{m}	Max. no. of stripes in a meta stripe	Chosen	2
N_{RRlevel}	No. of round-robin levels	Calculated	8

Another constraint is that a stripe segment should not exceed the capacity that is reserved for it within a GPU's global memory. Indeed, this memory must contain one I stripe segment for the *elim.I* thread as well as an I stripe segment and a J stripe segment for each *elim.J* thread. Notice that we neglect the other data structures within the GPU (R, B) since they are much smaller:

$$S^{\mathrm{s}} < \mathrm{NV}_{\mathrm{GPU}}/(1 + 2N_{\mathrm{JthGPU}})/S \tag{15}$$

Taking into account only the capacity of the array of SSDs, we should simply have

$$n_{\max} = \mathrm{sqrt}(\mathrm{NV}_{\mathrm{LSSD}}) \tag{16}$$

However, the meta-stripes-zigzag strategy imposes some constraints on $n_{\max}$: to obtain a good performance, we should be able to store at least $S^{\mathrm{m}} + 2$ full stripes in L_{PC} and, for peak performance, at least $2S^{\mathrm{m}} + 2$.

Overall, for good performance, $n_{\max}$ is calculated with

$$n_{\max} = \min(\mathrm{sqrt}(\mathrm{NV}_{\mathrm{LSSD}}), \mathrm{NV}_{\mathrm{LPC}}/(S^{\mathrm{m}} + 2)/S) \tag{17}$$

Peak performance requires

$$n_{\max} = \min(\mathrm{sqrt}(\mathrm{NV}_{\mathrm{LSSD}}), \mathrm{NV}_{\mathrm{LPC}}/(2S^{\mathrm{m}} + 2)/S) \tag{18}$$

As for N_{RRlevel}, we should have at least one complete round-robin level just before the remaining submatrix can be stored entirely within L_{PC} (see Sect. 7.3); therefore,

$$N_{\mathrm{RRlevel}} \geq n_{\max}/\mathrm{sqrt}(\mathrm{NV}_{\mathrm{LPC}}) \tag{19}$$

Preferently, N_{RRlevel} value should remain limited so as to maintain a high-enough granularity for the PC-mass storage transfers.

Notice that we may in the future choose to adjust N_{RRlevel} dynamically, in function of the current matrix size n, with the following formula:

$$N_{\mathrm{RRlevel}} = Q \times n/\mathrm{sqrt}(NV_{\mathrm{LPC}}) \tag{20}$$

$Q \sim 3$–10 or so would then limit, even in the worst cases, the truncature effect to within a factor of $(Q + 1)/Q \sim 1.33$–1.1.

9.8 Diverse hardware-related experiments

This section presents the results of some experiments over the hardware configuration (using the meta-stripes-zigzag strategy). Firstly, enabling the ECC automatic error correcting mechanism of the GPU does not induce any noticeable performance change. It does decrease the available GPU memory by 12.5 % (see Table 1), but the value of S^{s} had initially been chosen small enough that it still satisfies Eq. 15.

In stark contrast, the deactivation of hyper-threading within the PC brings our code to a standstill by halving N_{thPC}, in effect forcing Eq. 14 to be no longer satisfied.

Another experiment consists in grouping the six SSDs into an actual RAID 0 array, in place of our custom-made round-robin system. As a result, we observe a virtually identical performance. Admittedly, any eventual difference in bandwidth between the two solutions would be most likely hidden by our PC-based cache. Thus, practical hardware considerations such as the ones listed at the end of Sect. 7.3 should be prominent in deciding which solution to adopt.

Finally, using our round-robin system and a constant $N_{\mathrm{RRlevel}} = 8$, we gradually lower the number of SSDs in order to decrease the bandwidth. From 6 down to 2 SSDs, there is no noticeable change in performance. Clearly, it indicates that our PC-based cache is capable of buffering out to a great extent any variations in performance balance between the mass storage and the rest of the system. Ultimately, with only one SSD left, down from two, we do observe a decrease in performance of 22 % for $n = 100{,}000$ and of 37 % for $n = 150{,}000$.

10 Comparison with related work

10.1 About our algorithmic approach

Similarly to our work, the well-established LAPACK library [17] is also based on BLAS library and operates on shared memory systems (see ScaLAPACK [21] for distributed memory systems). A priori, any available parallel implementation of BLAS may be used, including of course CUBLAS, such as in CULA [8,13].

A block algorithm for LU factorization implemented in LAPACK [16] shows great similarity between its progression of panels *B* and *C* being factorized over the trailing matrix *E* (see Fig. 6 of [16]) and our partitioning of Fig. 4, with *B* corresponding to $\langle II \rangle$ and $\langle JI \rangle$ areas, *C* to $\langle IK \rangle$ area and *E* to $\langle JK \rangle$ area. However, below the overall partition into these general areas, we do operate an underlying partitioning of the matrix into stripes of rows.

Actually, we also subpartition (only at the algorithmic level and not for the data structures) the loops over index *i* that belong to the elimination codes for $\langle IK \rangle$, $\langle JI \rangle$ and $\langle II \rangle$. Most interestingly, this substripes optimization, as detailed in Sect. 4, bears a similarity with the recursive panel factorization of [9]. They both address the vexing issue of matrix areas that, although of much lower superficy than the main part to process (i.e. $\langle JK \rangle$ or *E*), are inherently more difficult to process efficiently in parallel. In both cases, the areas are subpartitioned to concentrate the problem at a lower scale, i.e. over an even smaller area.

Concerning our structuration of the matrix into stripes, it appears to be reminiscent of the so-called Hierarchical Tiling methodology [5] which consists in establishing a recursive hierarchy of tiles in order to distribute data and organize calculations over the diverse levels of memory units (from registers to mass storage) and processing units. The levels of the tile hierarchy correspond to groups of tiles of diverse granularities.

More generally than matrix tile algorithms that assume squarish tiles, a tile may be with this approach any small group of related data to be processed together.

Whereas [5] presents a general methodology to define a single hierarchy of tiles that matches any given hardware, we have in fact elaborated in a most instantiated manner a two-pronged hierarchical clusterization of data to be processed: on the one hand, from a purely algorithmic standpoint, meta-stripes, stripes and substripes correspond to three levels of processing with decreasing granularities. On the other hand, regarding actual data structures, complete stripes, stripe parts and segments of stripe parts constitute three levels of data grouping that are well fitted to our three-level memory system and to Gaussian elimination applied over stripes of rows (see Fig. 12).

10.2 Pivoting

Concerning pivoting, LAPACK uses the standard partial pivoting scheme, which is put into perspective by [1,12,15] with the approach often used for tile algorithms, namely block-pairwise pivoting. This last scheme is an adaptation of pairwise pivoting to the tile approach: practically, pairwise pivoting is carried out either within a single tile or within two tiles at a time.

Table 2 Ratios of residual errors for block-pairwise pivoting and for local partial pivoting with 13 tiles/stripes and with 25 tiles/stripes

	13 tiles/stripes	25 tiles/stripes
Ratio for block-pairwise pivoting	10.9	23.9
Ratio for local partial pivoting	6.3	16.2

Thus, block-pairwise pivoting proceeds over a limited range, similarly to local partial pivoting. We should then expect block-pairwise pivoting to be less stable than standard pairwise pivoting, itself being less stable than partial pivoting [22]. On the other hand, local partial pivoting is itself less stable than standard partial pivoting. So, how do local partial pivoting and block-pairwise pivoting compare with each other?

In relation to this question, it appears clearly from [1,12,15] that the numerical stability of block-pairwise pivoting is still a pretty open issue. Because of that, [1] conducted an experimental study of the residual errors incurred by block-pairwise pivoting. Tentatively, we use this study to draw a preliminary comparison with the residual errors from our code.

Agullo et al. [1] considered some matrices randomly generated with a uniform distribution (like our own test matrices) with sizes up to 10,240. Figure 8 of [1] presents for each matrix the ratio of the residual errors with block-pairwise pivoting over those with standard partial pivoting. Empirically, the ratio appears to depend mainly on the number of tiles in a column and not on the tiles' size.

With regard to our approach, and assuming that our stripe size is equivalent to the tile size (with respect to numerical errors), we use the results of Sect. 9.4 (with $\mathbf{S}' = 200$) to calculate the ratios of the errors with local partial pivoting over those with standard partial pivoting for $n = 25{,}000$ (13 stripes) and $n = 50{,}000$ (25 stripes). We then measure in Fig. 8 of [1] the ratios for the biggest matrix (size 10,240) with 13 and 25 tiles per column. All these ratios are reported in Table 2.

Assuming that the ratios of [1] depend only on the numbers of tiles, we may then tentatively compare them with our own ratios. For both 13 and 25 tiles/stripes, the ratio of [1] is slightly above our ratio. Overall, at least for matrices of limited size, the numerical stability of our code with local partial pivoting seems to be comparable with that of block-pairwise pivoting, with both losing an order of magnitude of accuracy with respect to standard partial pivoting. Of course, it is impossible to predict how such a comparison would extend to linear systems of much bigger sizes, especially as high as 400,000.

10.3 Performance in Gflops

As for the evaluation of our approach's performance in Gflops, it appears that [1] is of crucial interest for us. Indeed, that work presents the performance of a tile algorithm (originally developed in [4]) over a system based on C2050 GPUs. Since our results were also obtained on a C2050, it allows an exact quantitative comparison of our stripe algorithm with a tile algorithm.

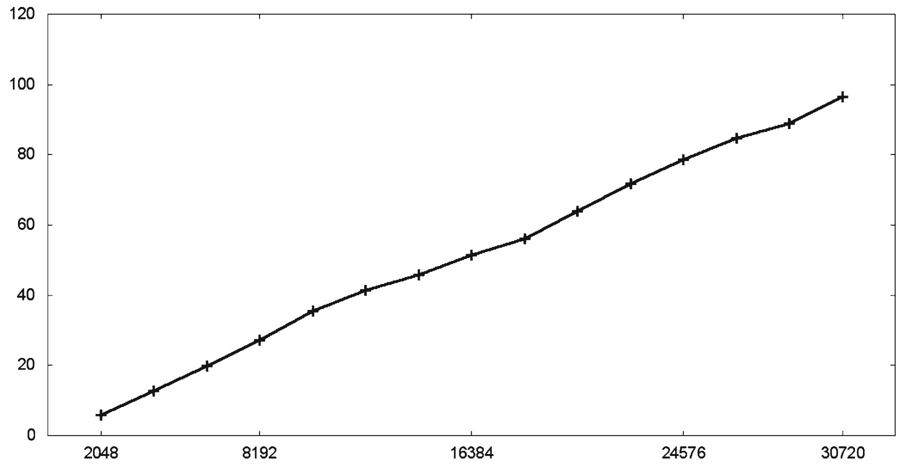


Fig. 29 With one C2050, performance, as a percentage of perf. in Fig. 9 of [1], of the triangulation phase in function of n (linear system size)

The actual basis for comparison is Fig. 9 of [1] which exposes performances with double-precision floating point arithmetic and in function of the linear system size n. We have extracted as accurately as possible from the curves of Fig. 9 of [1] some numerical values of performances to compare them with our values exposed in Fig. 22 for the meta-stripes-zigzag strategy. Of course, since they were deduced from a graphic, the values referenced below as originating from [1] may be very slightly different from the values originally measured by the authors of [1].

A preliminary remark is that [1] presents some results limited to linear system sizes of less than 32,768, whereas our results extend until $n = 400{,}000$. On the other hand, our results were obtained on one or two C2050s, but [1] experimented with up to 3 C2050s working in parallel and also tested using the 12 cores of its dual-CPU system to perform a fraction of the arithmetic calculations.

The performance of [1] with a single C2050 and no CPU cores used for calculations peaks at 169 Gflops (for n around 16,384) then gradually drops to 140 Gflops at 30,720. It is not clear if it would stabilize around 140 Gflops for bigger values of n or if it would continue diminishing. Concerning our approach, Fig. 22 shows that, although lagging behind for $n = 25{,}000$, the performance then steadily progresses until it reaches a peak of 201.0 Gflops at $n = 250{,}000$, i.e. some 33 Gflops higher than the peak of [1]. And the following decrease in performance, beyond 250,000, was shown in Sect. 9.3 to be most likely due to the insufficient amount of RAM of our PC (only 24 GiB, whereas the system of [1] has 48 GiB of RAM).

Figure 29 shows, with only one C2050 and for values of n covered by Fig. 9 of [1], the performance of our approach as a percentage of the performance in Fig. 9 of [1]. For these limited values of n, it appears that our approach is initially quite less efficient (in particular because of the initial cost of our local partial pivoting scheme, see Sect. 9.4). But it gradually gains pace until it reaches the same level of performance as the tile algorithm around $n = 30{,}720$.

Overall, the high granularity of stripes (versus tiles), together with our local partial pivoting scheme, appears to be a limitation below a size of 30,000, but it has definitely an increasingly beneficial impact for larger linear systems. We should stress that, with either 1, 2 or 3 C2050s, the performance of [1] always degrades beyond a size of 20,000–25,000 (at least without using their CPUs as a boost).

Consequently, we may then wonder how that discrepancy in granularity between stripes and tiles entails in practice such a discrepancy in scalability. Does our performance keep scaling up with n, whereas theirs is degrading because of their handling of too many low-granularity tiles that somewhat overwhelm their whole system, in particular their dynamic task scheduler and maybe also their PCIe bus?

As the possible beginning of an answer, [12] stresses indeed that "the size of a DAG considerably increases with the number of tiles", whereas our own DAG remains simple and of limited size (see Fig. 17), if only because we have as few as 200 stripes for $n = 400{,}000$.

Overall, a substantial advantage of [1] is that it uses a hybrid approach (see also about it [13,24,23]) and makes use of the CPU(s) of the system to provide additional computing power for arithmetic calculations. We surmise that it is the additional boost that allows [1] to offset the degradation due to the excessive number of tiles for big linear system sizes. Indeed, Fig. 9 of [1] shows that involving the two CPUs (12 cores) of their systems together with their three GPUs yields a performance that—at last—keeps steadily improving with the size.

This hybrid approach is best known through the MAGMA library. MAGMA [18] reports a performance of up to 240 Gflops for LU factorization on a system with a C2050 GPU and a quadcore GPU with a peak performance of 40 Gflops. Notice that [1] was actually a work intended to prepare the generalization of MAGMA (initially developed for single-GPU systems) to multi-GPU architectures.

The work presented in this paper does not yet include such a hybrid approach (with the exception of the row swapping performed by the CPU) and yet provides, thanks to the high granularity of stripes, acceptable performances of up to 200 Gflops or so. Indeed, our focus has been up to now on the development of an efficient multi-level memory system to solve dense linear systems far bigger than what can be stored within the RAM of a PC.

10.4 Overview of our approach's contributions to the field

In a field where tile algorithms are quite prevalent, we have shown that our concept of a "stripe algorithm" offers, if not a paradigm shift, at least an advantageous alternative for matrices of huge size. The higher granularity of stripes entails not only higher GPU performance for big enough matrices, but also the opportunity to develop an efficient multilevel memory system coupled with a highly scalable dynamic scheduling strategy. For these reasons, our approach allows the efficient resolution of huge dense linear systems on affordable PC-based platforms.

Notice that multilevel memory systems are quite common in computing (cache system of a CPU, virtual memory,…). However, what is particular in our work is that we have developed our memory system and the associated scheduling strategy in total symbiosis with the overall structure and characteristics of our stripe algorithm, so as to stress all possible synergies that we could think of. A most surprising of such synergies is that we have been able to use our methodology of formal code transformations (central to our stripe algorithm approach) and to generalize the concept of stripe to that of meta stripe to increase locality within the L_{PC} cache (meta-stripes-zigzag strategy).

Concerning the scheduling mechanism itself, our N-plicated multithreaded finite-state machine has turned out to be a formal, generic and highly adaptive tool (as a general matter, we strongly recommend the use of a state diagram for similar applications since it allows formal, fast and robust code development).

Regarding the issue of pivoting, we have developed for our stripe algorithm a local pivoting scheme that seems to be a plausible alternative to the commonly used block-pairwise pivoting. Actually, it has been shown to yield acceptable residual errors until at least $n = 400{,}000$.

11 Future developments

Although local partial pivoting seems a natural choice for our stripe algorithm, it is by no means the only option. In particular, block-pairwise pivoting may be adapted to our stripe structure, with pairwise pivoting occurring either within the range of stripe I or within the combined ranges of stripes I and J. Of course, our code might then become slightly faster without the overhead of the local partial pivoting search, at least below $n = 50{,}000$ (see Fig. 26). But that should be weighted against the risk of it being less stable numerically. The issue obviously warrants an extensive comparative study of the residual errors of block-pairwise pivoting versus local partial pivoting for dense linear systems of huge size (which, as we know, is still quite uncharted territory).

Also, our work has so far focused on the parallelization of the traditional Gauss method which solves the linear system $AX = B$ for a given B. However, some applications may require a number of resolutions with the same A and different values of B. For that purpose, the so-called LU factorization expresses A as the product of lower and upper triangular matrices. Once this factorization is established, the linear system can then be solved readily for diverse values of B.

Transforming our parallel code for Gauss method into LU factorization should be quite straightforward. Most notably, the zero values created by elimination within the parts of stripes should be replaced with the corresponding ratio values of R. Then, these stripe parts would be shuttled back toward PC memory just as in the current version of the code. A priori, their transfer cost should remain strictly the same. We, therefore, expect the overall performance to remain quasi-unchanged.

In order to boost performance, we are considering generalizing our code with a hybrid GPU-CPU approach, similarly to MAGMA, so that the CPU cores may also contribute to the arithmetic calculations. But we should be very cautious in doing so, especially with a single-CPU system: indeed, our CPU cores are already quite busy with row swapping and the whole mechanism of our three-level memory system. To overly distract them from these performance-critical tasks might trigger a crash in performance.

Regarding hardware developments, the latest generation of GPUs (Kepler) is substantially faster than that of the C2050 (Fermi), by a factor 2.5 or so (for double-precision floating point arithmetic). A multi-GPU system might also be considered.

Of course, such developments are likely to put the stress on the communication bottleneck between $\mathrm{L}_{\mathrm{SSD}}$ and L_{PC}. Fortunately, our meta-stripes-zigzag strategy for optimization of L_{PC} cache usage has—in our view—not yet shown its full potential since it has been used so far only with $S^{\mathrm{m}} = 2$. Increasing the size S^{m} of a meta stripe to 5 may indeed diminish further the amount of data transferred between $\mathrm{L}_{\mathrm{SSD}}$ and L_{PC} by a factor 2.5 or so. That might eventually suffice to accommodate a Kepler card with full performance (provided that the host PC has enough RAM with respect to the SSDs' capacity).

We may even look beyond and consider a cluster of GPU-equipped PCs (eventually with SSDs). Then, assuming the linear system is distributed over these PCs, our three-level memory system—and the tools developed for it—might be generalized to such a distributed memory (most particularly our caching methodology).

Ultimately, our initial work might be expanded into a practical and efficient matrix computation library adapted to diverse hardware configurations based on GPUs, similarly to [14].

Finally, we are considering applying our dense linear solver to some case studies in computational electromagnetics or computational quantum mechanics to show that it is feasible to solve large-scale problems in these fields using a single PC and without having to resort to an iterative resolution.

12 Conclusion

From a purely algorithmic standpoint, we have shown that formal transformations of the basic pseudocode of Gauss method lead to a pseudocode based on stripes of rows that is suitable for efficient implementation with CUBLAS. Interestingly, a crucial code optimization consists in repeating at a smaller scale, i.e. over substripes, the same sort of code transformation that was initially applied over the stripes. Further, applying again similar formal transformations—this time over meta stripes—yields a code that optimizes the use of our L_{PC} cache (meta-stripes-zigzag strategy).

At the level of data structures, our so-called stripe algorithm implies to store the matrix as a set of row stripes, thereby offering much higher granularity than the decomposition in tiles. This higher granularity of storage and treatment has allowed us to develop an "N-plicated multithreaded finite-state machine" that efficiently handles our three-level memory system, thus enabling us to solve with high performance some huge dense linear systems of size up to 400,000.

By contrast, we believe that the low granularity of tiles, and, therefore, their much bigger number, confines tile algorithms to relatively small linear systems. That should not come as a surprise considering that the number of tiles grows quadratically with n, whereas the number of stripes grows only linearly with n (thus allowing an easier handling by the system).

Concerning our extensive use of solid-state devices, a commonly overstated worry with SSDs is that they offer a limited number of read-write cycles before wearing out. In effect, our system has been in near-continuous use for a period of 6 months, with non-stop runs of up to 80 h or so (for $n = 400{,}000$). And yet the model of SSD used (3K) is not the most endurant one available on the market. Therefore, SSDs can be considered to expand the capability of a PC system to carry out heavy numerical calculations.

We are now looking forward to applying our algorithmic approach to more evolved computing systems, such as a cluster of GPU-equipped PCs which would allow us to solve even larger dense linear systems.

Eventually, we wish to turn our stripe algorithm into a practical library for dense matrix computations.

References

1. Agullo E, Augonnet C, Dongarra J, Faverge M, Langou J, Ltaief H, Tomov S (2011) LU factorization for accelerator-based systems. AICCSA' 11 conference, pp 217–224
2. Angelaccio M, Colajanni M (1993) Unifying and optimizing parallel linear algebra algorithms. IEEE Trans Parallel Distrib Syst 4(1):1382–1397
3. BLAS—basic linear algebra subprograms. www.netlib.org/blas
4. Buttari A, Langou J, Kurzak J, Dongarra J (2009) A class of parallel tiled linear algebra algorithms for multicore architectures. Parallel Comput 35:38–53
5. Carter L, Ferrante J, Flynn Hummel S, Alpern B, Gatlin KS (1996) Hierarchical tiling: a methodology for high performance. Technical Report, UCSD CS96-508
6. Cosnard M, Tourancheau B, Villard G (1987) Gaussian elimination on message passing architecture. In: Proceedings of 1st International Conference on Supercomputing. Athens. Lecture Notes in Computer Science, vol 297, pp 611–628
7. CUBLAS—implementation of BLAS on top of the NVIDIA CUDA runtime. http://docs.nvidia.com/cuda/cublas/index.html
8. CULA tools, GPU Accelerated Linear Algebra. http://www.culatools.com
9. Dongarra J, Faverge M, Ltaief H, Luszcsek P (2011) Achieving numerical accuracy and high performance using recursive tile LU factorization. Technical Report, University of Tennessee Computer Science ICL-UT-11-08 (also Lawn 259)
10. Evans F, Skiena S, Varshney A (1996) Optimizing triangle strips for fast rendering. IEEE Vis 96:319–326
11. Hadri B, Ltaief H, Agullo E, Dongarra J (2010) Tile QR factorization with parallel panel processing for multicore architectures. In: IEEE international symposium on parallel and distributed processing, pp 1–10
12. Haidar A, Ltaief H, YarKhan A, Dongarra J (2011) Analysis of dynamically scheduled tile algorithms for dense linear algebra on multicore architectures. Technical Report, University of Tennessee Computer Science UT-CS-11-666 (also Lawn 243)
13. Humphrey JR, Price DK, Spagnoli KE, Paolini AL, Kelmelis EJ (2010) CULA: hybrid GPU accelerated linear algebra routines. SPIE Conference Series 7705
14. Igual FD, Chan E, Quintana-Ortí ES, Quintana-Ortí G, Van de Geijn RA, Van Zee FG (2012) The FLAME approach: from dense linear algebra algorithms to high-performance multi-accelerator implementations. J Parallel Distrib Comput 72:1134–1143
15. Kurzak J, Ltaief H, Dongarra J, Badia RM (2010) Scheduling dense linear algebra operations on multicore processors. Concurrency and Computation: Practice and Experience 22:15–44
16. LAPACK—derivation of a block algorithm for LU factorization. http://www.netlib.org/utk/papers/siam-review93/node13.html
17. LAPACK—linear algebra PACKage. http://www.netlib.org/lapack
18. MAGMA—matrix algebra on GPU and multicore architectures. http://icl.cs.utk.edu/graphics/posters/files/SC11-MAGMA.pdf
19. Marquardt D (1963) An algorithm for least-squares estimation of nonlinear parameters. SIAM J Appl Math 11(2):431–441
20. PLASMA—parallel linear algebra software for multicore architectures. http://www.netlib.org/plasma

21. ScaLAPACK—scalable linear algebra PACKage. http://www.netlib.org/scalapack
22. Trefethen LN, Schreiber RS (1990) Average-case stability of Gaussian elimination. SIAM J Matrix Anal Appl 11:335–360
23. Tomov S, Dongarra J, Baboulin M (2010) Towards dense linear algebra for hybrid GPU accelerated manycore systems. Parallel Comput 36:232–240
24. Tomov S, Nath R, Ltaief H, Dongarra J (2010) Dense linear algebra solvers for multicore with GPU accelerators. In: IEEE international symposium on parallel and distributed processing, pp 1–8
25. Volkov V, Demmel JW (2008) Benchmarking GPUs to tune dense linear algebra. In: ACM/IEEE conference on supercomputing SC '08
26. Zhenjie D, Yan C (2009) An optimization load balancing algorithm design in massive storage system. In: ESIAT 2009 conference, vol 3, pp 310–313