—Technical Report INCO2-2012-01—
Parallel Two-Stage Least Squares algorithms for Simultaneous Equations Models on GPU

Carla Ramiro[a,1], Jose J. López-Espín[b], Domingo Giménez[c], Antonio M. Vidal[a]

[a]*Departamento de Sistemas Informáticos y Computación, Universidad Politécnica de Valencia, 46022 Valencia, Spain*
[b]*Centro de Investigación Operativa, Universidad Miguel Hernández, 03202 Elche, Spain*
[c]*Departamento de Informática y Sistemas, Universidad de Murcia, 30071 Murcia, Spain*

Abstract

Today it is usual to have computational systems formed by a multicore together with one or more GPUs. These systems are heterogeneous, due to the different types of memory in the GPUs and to the different speeds of computation of the cores in the CPU and the GPU. To accelerate the solution of complex problems it is necessary to combine the two basic components (CPU and GPU) in the heterogeneous system. This paper analyzes the use of a multicore+multiGPU system for solving Simultaneous Equations Models by the Two-Stage Least Squares method with QR decomposition. The combination of CPU and GPU allows us to reduce the execution time in the solution of large SEM.

Keywords: Simultaneous Equations Models, Two-Stage Least Squares, QR decomposition, Parallel Computing, Heterogeneous systems, GPU

1. Introduction

Simultaneous Equations Models (SEM) are often used in large scale problems. In that case, the use of parallel computation is indispensable due to the large amount of runtime required. Heterogeneous systems formed by multicores and GPUs deliver high performance with low cost, but to achieve the highest acceletation in the solution of problems it is necessary to exploit the heterogeneity of the multicore+multiGPU configuration. In this paper, parallel algorithms for the SEM have been developed, with the focus on exploiting the characteristics of each element of the heterogeneous system.

SEM are a statistical technique which has traditionally been used in the economic world [1] although nowadays it is widely used in an increasing number of fields [2, 3]. SEM can be solved through a variety of methods such as Indirect Least Squares (ILS), Two-Stage Least Squares (2SLS), Three-Stage Least Squares (3SLS), etc [1]. 2SLS is one of the most used methods because it can be used in all identified equations [1] (ILS can be used only in a particular case of equations) and is computationally less expensive than 3SLS [4]. Matrix decompositions and parallel computing are useful in obtaining efficient versions [4]. The wide diffusion of multicore and GPU systems allows efficient heterogeneous multicore+multiGPU algorithms of the 2SLS method with QR decomposition to be developed so reducing the execution time of previous parallel implementations [4].

The rest of the paper is organized as follows. In section 2 the ideas on SEM and the 2SLS method with QR decompositon are summarized. Section 3 describes the model of the considered computational system. In section 4 the algorithms developed are analyzed, and section 5 gives the experimental results. Finally, section 6 shows the conclusions of the paper.

Email addresses: cramiro@dsic.upv.es (Carla Ramiro), jlopez@umh.es (Jose J. López-Espín), domingo@um.es (Domingo Giménez), avidal@dsic.upv.es (Antonio M. Vidal)
[1]Corresponding author

2. Algorithms for SEM

Consider N interdependent variables (endogenous variables) which depend on K independent variables (exogenous variables). Suppose that each endogenous variable can be expressed as a linear combination of the other endogenous variables, the exogenous variables, and white noise which represents stochastic interference. The relation between the exogenous and endogenous variables can be expressed [1] through the following matrix equation:

$$\mathbf{Y} = \mathbf{Y}\mathbf{B}^{\mathbf{T}} + \mathbf{X}\mathbf{\Gamma}^{\mathbf{T}} + \mathbf{u} \tag{1}$$

where $\mathbf{Y} \in \mathbb{R}^{d\times N}$, $\mathbf{X} \in \mathbb{R}^{d\times K}$ and $\mathbf{u} \in \mathbb{R}^{d\times N}$ are matrices with N endogenous variables, K exogenous variables and N white noise variables respectively, d is the sample size, and elements $\mathbf{B_{ii}} = 0$.

Solving a SEM is equivalent to obtaining $\mathbf{B}$ and $\mathbf{\Gamma}$ in (1), from a representative sample of the model (a set of values of the data variables $\mathbf{X}$ and $\mathbf{Y}$) in order to explicitly ascertain a matrix equation which represents the relationship between both sets of variables.

2.1. Two-Stage Least Squares

The endogenous variables are correlated with the random variables, making it impossible to solve a Least Squares problem in each equation in (1). To solve this problem, 2SLS obtains a set of variables, called *proxy* variables ($\hat{\mathbf{Y}}$), which are close to the endogenous variables. The *proxy* variables are highly correlated with the exogenous variables but uncorrelated with the error ones. An estimation of the *proxy* variables can be obtained by expressing them in relation to the exogenous variables (equivalent to take the matrix $\mathbf{B} = 0$ in (1)) and finding $\hat{\mathbf{\Gamma}}^{\mathbf{T}}$ in such a way that

$$\|\mathbf{Y} - \mathbf{X}\hat{\mathbf{\Gamma}}^{\mathbf{T}}\| = \min_{\mathbf{\Gamma^T}} \|\mathbf{Y} - \mathbf{X}\mathbf{\Gamma}^{\mathbf{T}}\|.$$

Thus, the *proxy* variables are given by the expression $\hat{\mathbf{Y}} = \mathbf{X}\hat{\mathbf{\Gamma}}^{\mathbf{T}}$, and (1) can be approximated by substituing endogenous variables by *proxy* variables as

$$\mathbf{Y} = \mathbf{X}\mathbf{\Gamma}^{T} + \hat{\mathbf{Y}}\mathbf{B}^{T} + \mathbf{u}. \tag{2}$$

or by expressing (2) by columns:

$$\mathbf{y_i} = [\mathbf{X}|\hat{\mathbf{Y}}]\begin{bmatrix} \mathbf{\Gamma^T} \\ \mathbf{B^T} \end{bmatrix}_i + \mathbf{u}_i, \qquad i = 1 : N \tag{3}$$

where $\begin{bmatrix} \mathbf{\Gamma^T} \\ \mathbf{B^T} \end{bmatrix}_i$ indicates the column i of $\begin{bmatrix} \mathbf{\Gamma^T} \\ \mathbf{B^T} \end{bmatrix}$.

Consider the QR decomposition of the full column rank matrix $\mathbf{X} \in \mathbb{R}^{d\times K}$, with $K \leq d$, $\mathbf{X} = \mathbf{QR} = \mathbf{Q}\begin{pmatrix} \mathbf{R_1} \\ \mathbf{0} \end{pmatrix}$ where $\mathbf{Q} \in \mathbb{R}^{d\times d}$ is orthogonal and $\mathbf{R_1} \in \mathbb{R}^{K\times K}$ is an upper triangular matrix [5, chap.5]. If we denote $\tilde{\mathbf{Y}} = \mathbf{Q^T Y} = \begin{pmatrix} \tilde{\mathbf{Y}}_\mathbf{1} \\ \tilde{\mathbf{Y}}_\mathbf{2} \end{pmatrix}$, with $\tilde{\mathbf{Y}}_\mathbf{1} \in \mathbb{R}^{K\times N}$ and $\tilde{\mathbf{Y}}_\mathbf{2} \in \mathbb{R}^{(d-K)\times N}$, $\hat{\mathbf{\Gamma}}$ can be computed by solving the upper triangular system $\mathbf{R_1}\hat{\mathbf{\Gamma}}^{\mathbf{T}} = \tilde{\mathbf{Y}}_\mathbf{1}$ and *proxy* variables can be expressed as $\hat{\mathbf{Y}} = \mathbf{X}\hat{\mathbf{\Gamma}}^{\mathbf{T}} = \mathbf{Q}\begin{bmatrix} \mathbf{R_1} \\ \mathbf{0} \end{bmatrix}\mathbf{R_1^{-1}}\tilde{\mathbf{Y}}_\mathbf{1} = \mathbf{Q}\begin{bmatrix} \tilde{\mathbf{Y}}_\mathbf{1} \\ \mathbf{0} \end{bmatrix}$. Now, each equation in (3) can be expressed as

$$\mathbf{y_i} = \left[\mathbf{Q}\begin{bmatrix} \mathbf{R_1} \\ \mathbf{0} \end{bmatrix} | \mathbf{Q}\begin{bmatrix} \tilde{\mathbf{Y}}_\mathbf{1} \\ \mathbf{0} \end{bmatrix}\right]\begin{bmatrix} \mathbf{\Gamma^T} \\ \mathbf{B^T} \end{bmatrix}_i + \mathbf{u}_i = \mathbf{Q}\left[\begin{array}{c|c} \mathbf{R_1} & \tilde{\mathbf{Y}}_\mathbf{1} \\ \mathbf{0} & \mathbf{0} \end{array}\right]\begin{bmatrix} \mathbf{\Gamma^T} \\ \mathbf{B^T} \end{bmatrix}_i + \mathbf{u}_i \tag{4}$$

For each equation in (3) a matrix $[\mathbf{R_{i,1}}|\tilde{\mathbf{Y}}_\mathbf{i,1}]$ can be constructed, bearing in mind that some components of $\mathbf{B}$ and $\mathbf{\Gamma}$ may be zero. Construction of $[\mathbf{R_{i,1}}|\tilde{\mathbf{Y}}_\mathbf{i,1}]$ consists of eliminating those columns in matrix $[\mathbf{R_1}|\tilde{\mathbf{Y}}_\mathbf{1}]$ which are multiplied by null components of $\begin{bmatrix} \mathbf{\Gamma^T} \\ \mathbf{B^T} \end{bmatrix}_i$ in (4). Thus, (3) becomes

$$\mathbf{y_i} = \mathbf{Q}[\mathbf{R_{i,1}}|\tilde{\mathbf{Y}}_\mathbf{i,1}]\eta_\mathbf{i} + \mathbf{u}_i, \qquad i = 1 : N \tag{5}$$

with $\eta_\mathbf{i} \in \mathbb{R}^{k_i+n_i-1}$ being a vector formed by those non-null components of column i in matrix $\begin{bmatrix} \mathbf{\Gamma^T} \\ \mathbf{B^T} \end{bmatrix}$, n_i the number of non zero elements in $(\mathbf{B^T})_\mathbf{i}$, and k_i the number of non zero elements in $(\mathbf{\Gamma^T})_\mathbf{i}$. We assume adequate conditions for the existence of solutions in (5) ($n_i + k_i - 1 \leq K$, order condition, see [1]).

Now, each $\eta_\mathbf{i}$ can be estimated by solving the LS problem

$$\|\mathbf{Q}[\mathbf{R_{i,1}}|\tilde{\mathbf{Y}}_\mathbf{i,1}]\hat{\eta}_\mathbf{i} - \mathbf{y_i}\| = \min_{\eta_\mathbf{i}} \|[\mathbf{R_{i,1}}|\tilde{\mathbf{Y}}_\mathbf{i,1}]\eta_\mathbf{i} - \mathbf{Q^T y_i}\|. \tag{6}$$

QR decomposition of matrix $[\mathbf{R_{i,1}}|\tilde{\mathbf{Y}}_\mathbf{i,1}]$ can be used to solve this problem. Thus, if $[\mathbf{R_{i,1}}|\tilde{\mathbf{Y}}_\mathbf{i,1}] = \tilde{\mathbf{Q}}_\mathbf{i}\tilde{\mathbf{R}}_\mathbf{i,1}$, vector $\hat{\eta}_\mathbf{i}$ can be obtained by solving the upper triangular linear system $\tilde{\mathbf{R}}_\mathbf{i,1}\hat{\eta}_\mathbf{i} = \tilde{\mathbf{Q}}_\mathbf{i}^\mathbf{T}\tilde{\mathbf{y}}_\mathbf{i,1}$.

This procedure is summarized in Algorithm 1, which shows a scheme for the 2SLS method with the QR decomposition.

Algorithm 1 $2SLS_{QR}$ algorithm

Input: $\mathbf{X} \in \mathbb{R}^{d\times K}$, $\mathbf{Y} \in \mathbb{R}^{d\times N}$ and zero pattern of $\mathbf{B}$ and $\mathbf{\Gamma}$
Output: $\mathbf{B} \in \mathbb{R}^{N\times N}$ and $\mathbf{\Gamma} \in \mathbb{R}^{N\times K}$

```
Obtain Q, R and Ỹ such that X = QR (QR decomposition of X) and Ỹ = Q^T Y
for i=1...N do
    if i-th equation is identified (i.e. it can be solved [1]) then
        [R_{i,1}|Ỹ_{i,1}] ← Select columns from [R_1|Ỹ_1]
        Obtain Q̃_i, R̃_{i,1} and ỹ̃_{i,1} such that [R_{i,1}|Ỹ_{i,1}] = Q̃_i R̃_{i,1} and ỹ̃_{i,1} = Q̃_i^T ỹ_{i,1}
        Solve R̃_{i,1} η̂_i = ỹ̃_{i,1}
    end if
end for
```

3. Heterogeneous computational models

One of the most decisive concepts for successfully programming a computer that uses GPU and multicore processors is the underlying model of the parallel computer. Traditionally, a GPU card has been considered as an isolated parallel computer fitting a SIMD model, and connected to a sequential computer. From this point of view, the GPU card can be seen as a set of up to 512 (depending on model) processors, running the same instruction simultaneously, each on its own set of data. A realistic performance model should consider the host system and graphic card as a whole, and the host computer as another parallel computer, at the same level as the GPU. This leads us to the heterogeneous parallel computer model. A similar model is used in [6], where the machine is a set of computing elements with varying characteristics connected via independent links to a shared global memory.

Following this idea, a system with a GPU or an accelerator card (see Figure 1) consists of a set of two (or more) parallel computers, with different speeds, each with access to different types of memory, which also implies different memory access times for each processor.

Such a model would be characterized by the number and type of processors and different access time of each processor to the different types of memory. For example, a system comprising a multicore CPU plus two GPUs is considered in Figure 1. This system has a first-level cache and a main memory, shared by all cores, and two accelerator manycore cards with different types of memory (global memory, constant memory, shared memory). In this case the CPU can write on and read from global and constant memory of the GPU, and the GPU can write on and read from their global memory; but can only read from constant memory.

Performance and programming of this model depend on the type of parallel computer (MIMD for the CPU, SIMD for the GPU), the clock speed of CPU and GPU, the access time to each type of memory and the amount of memory in each memory class. The performance of a GPU in a system of this kind is difficult to evaluate as an isolated component. The best metric in this case may be to compare the speed of the system with and without the accelerator card. A simultaneous use of the GPU and CPU should be allowed (and even encouraged), and the performance obtained should be compared when they act together and when eliminating the use of the GPU to solve a concrete problem. This approach is much more realistic and is used, for example, in numerical linear algebra libraries.

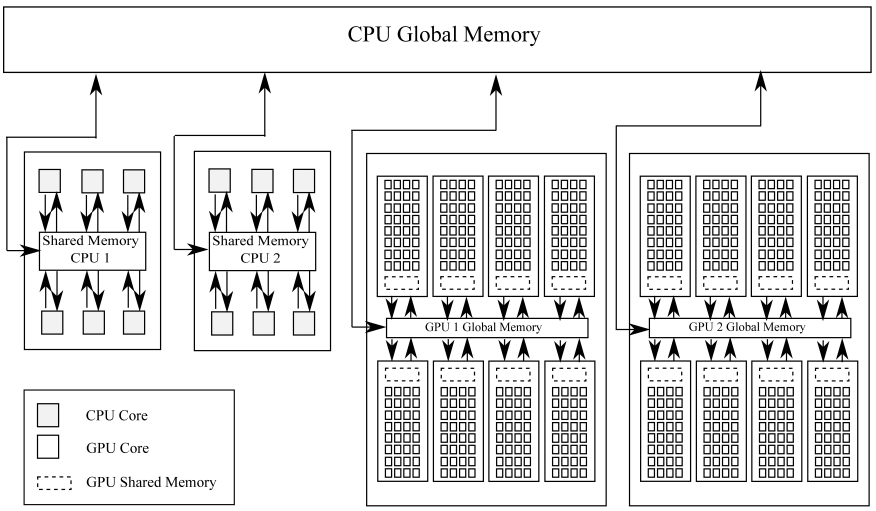

Figure 1: Heterogeneous parallel computer system

4. Parallel Algorithms for 2*SLS*

Now, we analyze different possibilities of parallelization of Algorithm 1 in a heterogeneous parallel computer. The concrete model corresponding to our target machine consists of a multicore CPU unit with two GPU cards. It can be seen that most of the computational cost of the algorithm relies on the calculation of one QR decomposition for each equation. The parallelization of the problem can be addressed in two ways. The first is based on dividing the set of equations to be solved among the various computational elements of the machine. The second idea is to parallelize the computation of the QR decomposition, by using Givens rotations, and taking advantage of the structure of the matrix $[R_{i,1}|\tilde{Y}_{i,1}]$. Algorithms which combine both types of parallelization can be also devised.

4.1. Parallelism by equations

As it can be seen in Algorithm 1, to solve the SEM problem we need solve N equations. These equations are independent, therefore we can parallelize their computation. We used the OpenMP API [7], which supports multi-platform shared memory multiprocessing programming in C and which can be used with CUDA.

We can create several threads. Some of these threads (depending on the number of GPUs the platform has) can work on the GPUs and are responsible for transferring data to the global memory graphic card and processing the data back, while the other threads can work on the CPU cores in parallel.

4.2. Parallel QR decomposition on GPU

To calculate the QR decomposition on the GPU, we have not used kernels available in libraries, for example MAGMA [8] or CULA [9], because these libraries do not take into account the structure of the matrix. Therefore we have implemented a parallel algorithm in CUDA, based on the triangularization of the matrix $[\mathbf{R_{i,1}}|\tilde{\mathbf{Y}}_{\mathbf{i,1}}]$ by using a parallel scheme proposed in [10], which simultaneously nullifies elements placed in several main diagonals. The only necessary condition to calculate and apply a number of Givens rotations simultaneously is that the rows involved in its computation are disjoint. Figure 2 shows an example of triangulation of a 8×4 matrix using Givens rotations. Note that ten sequential stages (NS) are required for triangularizing the matrix in this case.

$$\begin{bmatrix} x & x & x & x \\ 7 & x & x & x \\ 6 & 8 & x & x \\ 5 & 7 & 9 & x \\ 4 & 6 & 8 & 10 \\ 3 & 5 & 7 & 9 \\ 2 & 4 & 6 & 8 \\ 1 & 3 & 5 & 7 \end{bmatrix}$$

Figure 2: Example of the triangularization by using a parallel scheme with 10 sequential stages

The explicit construction of the orthogonal matrix Q has been avoided. For computing $\tilde{\tilde{\mathbf{y}}}_{\mathbf{i,1}} = \tilde{\mathbf{Q}}_{\mathbf{i}}^{\mathbf{T}} \tilde{\mathbf{y}}_{\mathbf{i,1}}$, we create a matrix of the form $[\mathbf{R_{i,1}}|\tilde{\mathbf{Y}}_{\mathbf{i,1}}|\tilde{\tilde{\mathbf{y}}}_{\mathbf{i,1}}]$ and the calculated rotations are applied to the column $\tilde{\tilde{\mathbf{y}}}_{\mathbf{i,1}}$. Algorithm 2 computes the QR decomposition of a matrix $\mathbf{Z}$ and applies $\mathbf{Q^T}$ on another matrix $\mathbf{W}$ by using this method.

Algorithm 2 QR-SPAN

Input: $[\mathbf{Z} \mid \mathbf{W}]$
Output: $\mathbf{R}, \widetilde{\mathbf{W}}$

1: **for** $s = 1, \ldots, NS$ **do**
2: Compute index rotations, row i and column j, needed for triangulation matrix $\mathbf{Z}$ for stage s in CPU (see Figure 2).
3: Copy index rotations from CPU main memory to constant memory in GPU and $[\mathbf{Z} \mid \mathbf{W}]$ to GPU global memory.
4: The threads of the GPU calculate in parallel the rotations for the stage s. (see Algorithm 3)
5: The threads of the GPU apply in parallel the rotations calculated in the previous step to the matrices $\mathbf{Z}$ and $\mathbf{W}$. (see Algorithm 4)
6: **end for**
7: Copy $[\mathbf{R} \mid \widetilde{\mathbf{W}}]$ from GPU global memory to CPU main memory

Before starting the QR decomposition, the matrices $\mathbf{Z}$ and $\mathbf{W}$ are copied into the GPU global memory. For each stage, index rotations are copied into the GPU constant memory. In a first kernel, each thread block is in charge of calculating a group of rotations. The rotations are stored in compact way [5, chap.5] at the bottom of the matrix to minimize the memory requirements as shown in Algorithm 3.

Algorithm 3 Calculation of the one rotation by the qth thread

1: Get one of the index rotations (i, j) from constant memory
2: Compute Givens rotation cos and sin for $\mathbf{Z_{i-1,j}}$ and $\mathbf{Z_{i,j}}$
3: $\mathbf{Z_{i-1,j}} \leftarrow cos \times \mathbf{Z_{i-1,j}} - sin \times \mathbf{Z_{i,j}}$
4: Store cos and sin in a compact way in $\mathbf{Z_{i,j}}$

After the calculation of the rotations for one stage is finished, a new kernel is launched. Now one thread by row reconstructs the compact rotation in shared memory for faster access. Then the rest of threads apply this rotation in the rest of elements in the row, as shown in Algorithm 4.

Algorithm 4 Application of the one rotation by the qth thread

1: Get one of the index rotations (i,j) from constant memory
2: **if** $threadIdx.x = 0$ **then**
3: Reconstruct rotation $\mathbf{Z_{i,j}}$ and store cos and sin in shared memory
4: **end if**
5: Sync barriers
6: $\mathbf{Z_{i-1,q}} \leftarrow cos \times \mathbf{Z_{i-1,q}} - sin \times \mathbf{Z_{i,q}}$
7: $\mathbf{Z_{i,q}} \leftarrow sin \times \mathbf{Z_{i-1,q}} + cos \times \mathbf{Z_{i,q}}$

4.3. Implemented algorithms

Following the two schemes previously seen, we have implemented several algorithms. The objective of this study is to see which algorithm is best suited to the platform. In Algorithm 5 we can see the number of threads and parallelization scheme depending on the algorithm used.

Algorithm 5 $2SLS_{QR}$ algorithms parallelized

Input: $\mathbf{X} \in \mathbb{R}^{d\times K}$, $\mathbf{Y} \in \mathbb{R}^{d\times N}$ and zero pattern of $\mathbf{B}$ and $\mathbf{\Gamma}$
Output: $\mathbf{B} \in \mathbb{R}^{N\times N}$ and $\mathbf{\Gamma} \in \mathbb{R}^{N\times K}$

Obtain $\mathbf{Q}$, $\mathbf{R}$ and $\tilde{\mathbf{Y}}$ such that $\mathbf{X} = \mathbf{QR}$ (QR decomposition of $\mathbf{X}$) and $\tilde{\mathbf{Y}} = \mathbf{Q^T Y}$ using Algorithm 2 except the $1CPUs$ algorithm that computes this step in sequential.
switch (algorithm)
  case ($pCPUs$)
    Launch p threads: p threads calculate the step 15 with a sequential algorithm in the CPU.
  case ($pCPUs + nGPU(S)$)
    Launch $p + n$ threads: p threads calculate the step 15 with a sequential algorithm in CPU and n with a sequential algorithm in GPU.
  case ($nGPU(P)$)
    Launch n threads: n threads calculate the step 15 using Algorithm 2 in GPU.
  case ($pCPUs + nGPU(P)$)
    Launch $p + n$: p threads calculate the step 15 with a sequential algorithm in CPU and n threads with parallel Algorithm 2 in GPU.
end switch
IN PARALLEL: N equations are distributed among all threads using schedule dynamic. Once a particular thread finishes its allocated iteration, it returns to get another one from the iterations that are left.
if i-th equation is identified (i.e. it can be solved [1]) **then**
  $[\mathbf{R_{i,1}}|\tilde{\mathbf{Y}}_{\mathbf{i,1}}] \leftarrow$ Select columns from $[\mathbf{R_1}|\tilde{\mathbf{Y}}_{\mathbf{1}}]$
  Obtain $\tilde{\mathbf{Q}}_{\mathbf{i}}$, $\tilde{\mathbf{R}}_{\mathbf{i,1}}$ and $\tilde{\tilde{\mathbf{y}}}_{\mathbf{i,1}}$ such that $[\mathbf{R_{i,1}}|\tilde{\mathbf{Y}}_{\mathbf{i,1}}] = \tilde{\mathbf{Q}}_{\mathbf{i}}\tilde{\mathbf{R}}_{\mathbf{i,1}}$ and $\tilde{\tilde{\mathbf{y}}}_{\mathbf{i,1}} = \tilde{\mathbf{Q}}_{\mathbf{i}}^{\mathbf{T}}\tilde{\mathbf{y}}_{\mathbf{i,1}}$
  Solve $\tilde{\mathbf{R}}_{\mathbf{i,1}}\hat{\eta}_{\mathbf{i}} = \tilde{\tilde{\mathbf{y}}}_{\mathbf{i,1}}$
end if
END PARALLEL

- In the algorithm ***pCPUs***, N equations are distributed among the threads to be solved independently and these threads are run exclusively on the CPU cores.
- The version denoted as ***pCPUs + nGPU(S)*** uses the cores available in the heterogeneous system and distributes the solution of the equations among p cores of the CPU and of the n GPUs. Each GPU launches a kernel and each thread computes a QR decomposition. The distribution (see Figure 3) is performed dynamically with clause *schedule dynamic* and the parameter "chunk" is used to determine the number of contiguous iterations that are allocated to a thread at one time.

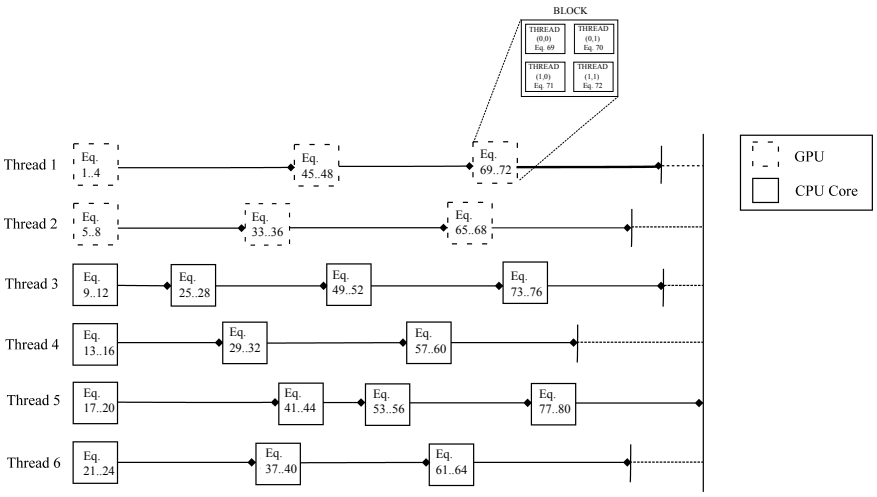

Figure 3: Example of load-balancing for an execution with 80 equations, 6 threads and chunk 4

- Two versions with the second parallelization scheme have been implemented. The calculation of the QR decomposition is parallelized following Algorithm 2. Version ***1GPU(P)*** runs on one GPU. Version ***2GPU(P)*** distributes dynamically, with the OpenMP scheduling clause *schedule dynamic*, the equations between the two GPUs, and each GPU applies Algorithm 2 on its set of equations.

- The ***pCPUs + nGPU(P)*** version distributes the solution of the equations among the computational elements of the heterogeneous system, and the GPU applies QR-SPAN to solve the equations. The distribution is performed dynamically. In this case, the threads are executed by all the elements of the machine, i.e. by all the cores of the CPU and by one or both GPU cards. ***11CPUs + 1GPU(P)*** uses a single GPU, ***12CPUs + 2GPU(P)*** and ***10CPUs + 2GPU(P)*** uses two GPUs with dynamic load-balancing. The distribution (see Figure 4) is performed dynamically with clause *schedule dynamic*. In this case, the threads are executed by all the elements of the machine, i.e. by all the cores of the CPU and by the two GPU cards.

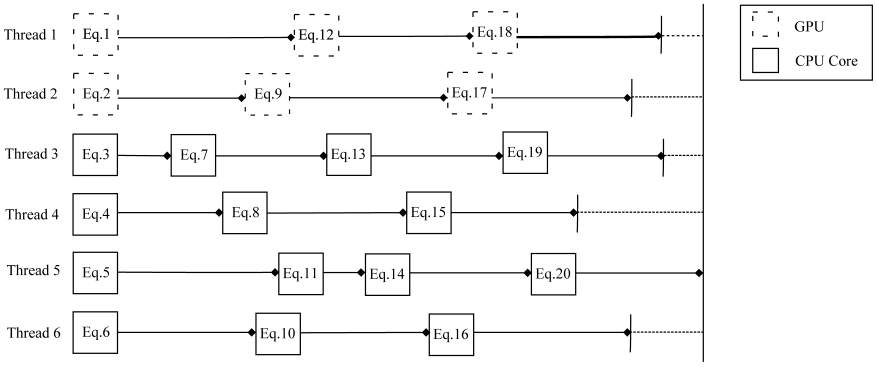

Figure 4: Example of load-balancing for an execution with 20 equations and 6 threads

5. Experimental Results

The computer used in our experiments, has two Intel Xeon X5680 processors at 3.33 GHz and 96 GB of GDDR3 main memory. Each one is an hexacore processor with 12 MB of cache memory. It contains two Nvidia Tesla C2070 GPU with 14 stream multiprocessors (SM). Each Fermi SM includes 32 cores, 16 load/store units, four special-function units, a 32K-word register file, 64K of configurable RAM, and thread control logic. Each core has both floating-point and integer execution units. Floating-point operations follow the IEEE 754-2008 floating-point standard. Each core can perform one single-precision fused multiply-add operation in each clock period and one double-precision FMA in two clock periods. The core frequency is 1.15 GHz and each GPU has 6 GB of GDDR5 global memory. The installed CUDA toolkit is 4.0 and it has also libraries like MKL 10.3.

Since this paper aims to obtain fast algorithms to solve SEM, the data used for evaluation have been randomly generated, bearing in mind the temporal performance of algorithms.

Table 1: Execution time of ***1CPUs***, ***12CPUs***, ***12CPUs+2GPU(S)***, ***10CPUs+2GPU(S)***, ***1GPU(P)***, ***2GPU(P)***, ***12CPUs+2GPU(P)***, ***11CPUs+1GPU(P)*** and ***10CPUs+2GPU(P)***. N, K and d variable.

N	400	400	400	400	800	800	800	800	1000	1000	1200	1200
K	400	400	600	600	800	800	1000	1000	1200	1200	1200	1200
d	1000	1500	1000	1500	2000	2500	2000	2500	2500	3000	2500	3000
$1CPUs$	6.08	6.20	13.56	13.59	87.77	89.84	132.78	135.25	213.40	300.59	483.29	443.79
$12CPUs$	0.76	**0.82**	**1.44**	**1.93**	10.19	10.22	17.36	17.58	28.61	48.56	75.43	76.01
$12CPUs + 2GPU(S)$	0.99	1.29	2.23	2.24	12.47	12.49	20.89	21.23	41.38	57.72	90.98	90.91
$10CPUs + 2GPU(S)$	0.93	1.43	2.13	2.46	12.60	12.66	20.85	21.36	33.84	56.90	87.62	88.85
$1GPU(P)$	5.70	5.72	10.96	10.94	47.54	47.91	70.67	71.00	100.97	142.66	189.77	187.77
$2GPU(P)$	3.74	3.82	8.36	8.39	32.42	33.93	48.86	49.53	68.30	96.21	130.53	124.70
$12CPUs + 2GPU(P)$	0.90	1.17	1.72	**1.93**	9.25	**9.22**	14.24	14.98	29.23	35.56	50.47	55.58
$11CPUs + 1GPU(P)$	**0.69**	1.21	1.54	2.21	11.01	10.95	17.54	17.14	25.30	40.57	61.60	60.17
$10CPUs + 2GPU(P)$	0.74	1.30	1.60	1.96	**9.04**	9.32	**13.66**	**13.81**	**20.84**	**31.91**	**47.43**	**47.01**

Table 1 also shows the execution time of the algorithms in double precision and 12 threads. Note that, the performance of GPU in double precision is very poor in comparison to single precision, though the technique SEM requires double data type. Figure 5 shows the speedup of the implemented algorithms with regard to the ***1CPUs*** algorithm.

- The results of the parallelization with ***12CPUs*** are good, and programming is very simple, yet they are far from achieving the theoretical speedup. As can be seen in Figure 5, the maximum speedup is 9; when the problem size increases, the speedup drops to 6.

- When we apply the same parallelization scheme ***10CPUs+2GPU(S)*** without taking into account the specificities of the GPU, we get negative effects on performance. Now, the algorithm makes a bad use of memory. It performs an excessive number of memory accesses because there are many uncooperative threads that work simultaneously. Furthermore, a higher CPU speed with regard to GPU also represents a problem if we try to compare performances of the algorithms in CPU and GPU.

- The use of the GPU as a standalone tool ***1GPU(P)*** and ***2GPU(P)*** provides benefits but does not even reach the performance obtained when using parallelism in the CPU. It can also be seen how the use of two GPUs does not imply reducing the execution time by half. This is because the CPU also performs some support work, for example, by sending data to the GPU. When data have to be sent to two GPU these transferences can not overlap.

- The best results are obtained when using all the CPU + GPU system as a single heterogeneous computer ***11CPUs+1GPU(P)*** and ***10CPUs+2GPU(P)***; the speedup increases as the problem size increases and approaches the theoretical maximum of 12, achieving almost twice that obtained with the ***12CPUs*** version. In this case, the load is dynamically balanced. It was necessary to adapt the programming scheme to each type of element in the computer, using a scheme similar to ***12CPUs*** algorithm for the CPU and a scheme similar to the

1GPU(P)/2GPU(P) algorithm for the GPU. For small sizes, better performances are obtained with a single graphics card ***11CPUs+1GPU(P)***.

- Note that the use of the ***12CPUs*** in ***12CPUs+2GPU(P)*** and ***12CPUs+2GPU(S)*** versions does not achieve better speedup except for some small sizes. This is because two of this cores are busy managing the sending and receiving of data from the GPU and therefore remain occupied for some time.

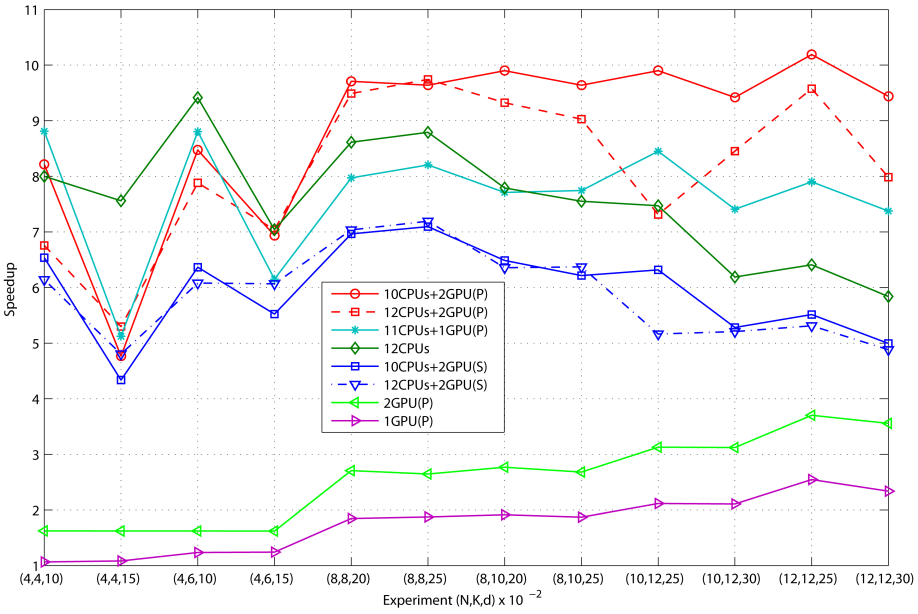

Figure 5: Speedup for ***12CPUs***, ***12CPUs+2GPU(S)***, ***10CPUs+2GPU(S)***, ***1GPU(P)***, ***2GPU(P)***, ***12CPUs+2GPU(P)***, ***11CPUs+1GPU(P)*** and ***10CPUs+2GPU(P)*** with regard to the ***1CPUs***

6. Conclusions

Efficient algorithms for solving Simultaneous Equations Models have been developed. The use of computers with multicore + GPU architecture allows for very competitive performances in large scale problems, with affordable hardware, so the execution time obtained with precious 2SLS algorithms for SEM can be reduced with the combined use of CPU and GPU.

The use of computers with multicore + GPU architecture can be useful when working with big models built from a set of lower dimension models. For example, SEM for the economic variables in Europe can be obtained from the models of the different countries by adding some union equations. So, the total model has a large number of variables. The world model (managed by the LINK project at the University of Toronto [11]) includes the Spanish model (managed by the CEPREDE [12] or Centro de Predicción Económica and by the Institute Laurence R. Klein [13] at the University Autónoma de Madrid) and a set of equations to connect the Spanish variables with the global ones.

We have shown that when we are working on a heterogeneous system it is necessary to design dynamic and hybrid algorithms to exploit the full potential of the machine. In this way we can even approach the theoretical maximum speedup, as seen in Figure 5.

Our contribution shows that we can efficiently exploit the resources of the machine even for dense linear algebra problems of double data type where GPUs do not offer good performance, as occurs in some highly optimized libraries

that use the hybrid programming CPU with GPU, as CULA [14] or MAGMA [15], where the speed up achieved is far from the theoretical one.

However, programming these systems must be performed carefully. Heterogeneity makes it difficult. To obtain optimum performance, problems should be suitable. For example, in the problem studied, QR decomposition algorithm consumes a lot of time and the algorithmic structure does not allow optimal parallelization, due to the lock-step that occurs in each stage.

Although GPUs are a great tool, their benefits are not miraculous, especially in fields such as Numerical Linear Algebra. Compared with the behaviour of multicore architectures, benefits are less scalable, i.e. a linear increase in resources may not mean a linear increase in performance.

Acknowledgements

Supported by: Spanish Ministerio de Ciencia e Innovación (Projects TIN2008-06570-C04-02, TEC2009-13741 and CAPAP-H3 TIN2010-12011-E), Universidad Politécnica de Valencia (PAID-05-10), Fundación Séneca from C.A.Región de Murcia (08763/PI/08) and Generalitat Valenciana (project PROMETEO/2009/013).

References

[1] W. Greene, Econometric Analysis, 3rd Edition, Prentice Hall, 1998.
[2] R. Henry, I. Lu, L. Beightol, D. Eckberg, Interactions between CO2 Chemoreflexes and Arterial Baroreflexes, Am. Journal of Physiology 274 (43) (1998) H2177–H2187.
[3] W. Ressler, M. Waters, Female earnings and the divorce rate: a simultaneous equation model, Applied Economics 32 (2000) 1889–1898.
[4] J. J. López-Espín, A. M. Vidal, D. Giménez, Two-stage least squares and indirect least squares algorithms for simultaneous equations models, Journal of Computational and Applied Mathematics (0) (2011) –. doi:10.1016/j.cam.2011.07.005.
URL `http://www.sciencedirect.com/science/article/pii/S0377042711003918`
[5] G. Golub, C. V. Loan, Matrix Computations, The John Hopkins University Press, 1996.
[6] G. Ballard, J. Demmel, A. Gearhart, Communication bounds for heterogeneous architectures., Tech. Rep. 239, LAPACK W.N (2011).
URL `http://www.netlib.org/lapack/lawnspdf/lawn239.pdf`
[7] OpenMP v3.0, http://www.openmp.org/mp-documents/spec30.pdf (May 2008).
[8] S.Tomov, R.Nath, P.Du, J.Dongarra, http://icl.cs.utk.edu/magma.
[9] C. tools, www.culatools.com.
[10] A. Sameh, D. Kuck, On stable parallel linear system solvers, Journal of the ACM 25 (1) (1978) 81–91.
[11] Project LINK Research Centre, `www.chass.utoronto.ca/link`.
[12] Centro de Predicción Económica, `www.ceprede.com`.
[13] Instituto Universitario de Predicción Económica "Lawrece R. Klein", `web.uam.es/otroscentros/klein`.
[14] CULA GPU Accelerated Linear Algebra, `www.culatools.com/features/performance`.
[15] MAGMA Matrix Algebra on GPU and Multicore Architectures, `www.icl.cs.utk.edu/magma/`.