Available online at www.sciencedirect.com

SciVerse ScienceDirect

Procedia Computer Science 9 (2012) 47 – 56

Procedia Computer Science

International Conference on Computational Science, ICCS 2012

Heterogeneous Computational Model for Landform Attributes Representation on Multicore and Multi-GPU Systems

Murilo Boratto[a,1], Pedro Alonso[b], Carla Ramiro[b], Marcos Barreto[c]

[a]*Colegiado de Engenharia da Computação, Universidade Federal do Vale do São Francisco, Juazeiro, Bahia, Brazil*
[b]*Departamento de Sistemas Informáticos y Computación, Universidad Politécnica de Valencia, Valencia, Spain*
[c]*Laboratório de Sistemas Distribuídos, Universidade Federal da Bahia, Salvador, Bahia, Brazil*

Abstract

Mathematical models are often used to simplify landform representation. Its importance is due to the possibility of describing phenomena by means of mathematical models from a data sample. High processing power is needed to represent large areas with a satisfactory level of details. In order to accelerate the solution of complex problems, it is necessary to combine two basic components in heterogeneous systems formed by a multicore with one or more GPUs. In this paper, we present a methodology to represent landform attributes on heterogeneous multicore and multi-GPU systems using high performance computing techniques for efficient solution of two-dimensional polynomial regression model that allow to address large problem instances.

Keywords: Mathematical Modeling, Landform Representation, Parallel Computing, Performance Estimation, Multicore, Multi-GPU

1. Introduction

Some recent events have encouraged the development of applications that represent geophysical resources efficiently. Among these representations, mathematical models for representing landform attributes stand out, based on two-dimensional polynomial equations [1]. Landform attributes representation problem using polynomial equations had already been studied in [2]. However, distributed processing was not used in that work, which implied the usage of a high degree polynomial, thus limiting the area representation. It occurred because the greater the represented information, the greater computational power is needed. Furthermore, a high degree polynomial is required to represent a large area correctly, which also demands a great computational power.

Among the reasons for fulfilling landform representation, we focus on measuring agricultural areas, having as preponderant factors: plantation design, water resource optimization, logistics and minimization of erosive effects. Consequently, landform representation process becomes a fundamental and needful tool in efficient agriculture operation, especially in the agricultural region located in São Francisco river valley, which stands out as one of the largest viniculture and fruit export areas in Brazil. In addition, one of the main problems that make efficiency use difficult in

Email addresses: murilo.boratto@univasf.edu.br (Murilo Boratto), palonso@dsic.upv.es (Pedro Alonso), cramiro@dsic.upv.es (Carla Ramiro), marcoseb@dcc.ufba.br (Marcos Barreto)
[1]Corresponding author

doi:10.1016/j.procs.2012.04.006

agricultural productivity factors in this areas occurred due to soil erosion associated with inappropriate land use. In this context, the work proposed here contributes to the characterization of soil degradation processes.

Today it is usual to have computacional systems formed by a multicore with one or more Graphics Processing Units (GPUs) [3]. These systems are heterogeneous, due to different types of memory and different speeds of computation between CPU and GPU cores. In order to accelerate the solution of complex problems it is necessary to use the aggregated computational power of the two subsystems. Heterogeneity introduces new challenges to algorithm design and system software. Our main goal is to fully exploit all the CPU cores and all GPUs devices on these systems to support matrix computation [4]. Our approach achieves the objective of maximum efficiency by an appropriate balancing of the workload among all these computational resources.

The purpose of this paper is to present a methodology for landform attributes representation of São Francisco river valley region based on two-dimensional polynomial regression method on heterogeneous multicore and multi-GPU systems. Section 2 briefly describes the mathematical model. Section 3 explains the heterogeneous model on multicore and multi-GPU systems. Section 4 describes the parallel implementation. Section 5 presents the experimental results. Conclusions and future work section closes the paper.

2. Mathematical Model

A mathematical landform model is a computational mathematical representation of a phenomenon that occurs within a region of the earth surface [5]. This model can represent plenty of geographic information from a site, such as: geological, geophysical, humidity, altitude, terrain, etc. One available technique to accomplish this representation is the Regular Grid Model [6]. This model makes the surface mapping with a global fitting using the polynomial regression technique, which is a kind of mathematical modeling that attempts to describe the relation among observed phenomema. This technique fits a two-dimensional polynomial that best describes the data variation form a sample. The key problem here is that the high processing power needed to perform the regression in a large data set makes the process very limited.

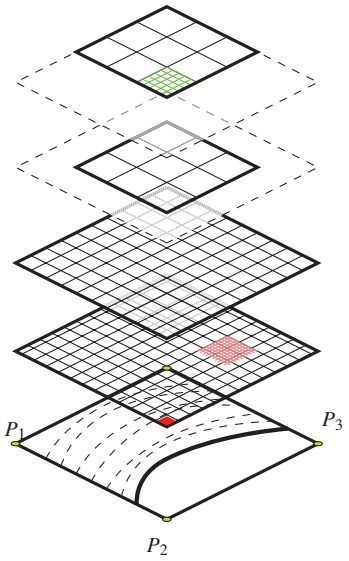

Figure 1: Model for landform attributes representation: Regular Grid.

Figure 1 shows an example of a Regular Grid representation generated from a regular sparse sample that represents information of an area altitude. According to Rawlings [7], modeling can be understood as the development of a mathematical analytical model that describes the behavior of a random variable of interest. This model is used to describe the behavior of independent variables whose relationship with the dependent variable is best represented in a

non-linear equation. The relationship among variables is described by two-dimensional polynomial functions, where the fluctuation of the dependent variable is related to the fluctuation of the independent variable. Particularly, in the case study developed in this project, the non-linear regression has been used to describe the relationship between two independent variables (latitude and longitude) and a dependent variable (altitude). The mathematical model we use provides the estimation of the coefficients of two-dimensional polynomial functions, of different degrees in x and y, which represent terrain altitude variation from any area.

When using mathematical regression models, the most widely used estimation method of parameters is the Ordinary Least Squares [8] that consists of the estimation of a function to represent a set of points minimizing the deviations squared. Given a set of geographic coordinates (x, y, z), taking the estimated altitude as estimation function of these points, a two-dimensional polynomial of degrees r and s in x and y, respectively, can be given as shown in Equation 1.

$$\hat{z}_{ij} = f(x_i, y_j) = \sum_{k=0}^{r} \sum_{l=0}^{s} a_{kl} x_i^k y_j^l \,, \tag{1}$$

A particular form for the polynomial in Equation 1 can be exemplified for $r = s = N$ case as follows,

$$\hat{z}_{ij} = a_{00} x_i^0 y_j^0 + a_{01} x_i^0 y_j^1 + \cdots + a_{NN} x_i^N y_j^N .$$

Let m and n be the length and width of the surface grid, respectively, then the coefficients a_{kl}, for $k = 0, 1, ..., r$ and $l = 0, 1, ..., s$, that minimize the function error of the estimation function $f(x, y)$, can be obtained by solving the following equation, for $c = 0, 1, ..., r$ and $d = 0, 1, ..., s$,

$$\partial \left(\sum_{i=0}^{m} \sum_{j=0}^{n} (z_{ij} - \hat{z}_{ij})^2 \right) / \partial a_{cd} = 0 \,. \tag{2}$$

After some derivations [9] we get that the solution to Equation 2 has the form

$$\sum_{i=0}^{m} \sum_{j=0}^{n} \left[z_{ij} x_i^c y_j^d - \left(\sum_{k=0}^{r} \sum_{l=0}^{s} a_{kl} x_i^{k+c} y_j^{l+d} \right) \right] = 0 \,. \tag{3}$$

The solution drawn in Equation 3 can be summarized in a matrix representation form. In particular, we deal with a square surface ($n = m$). Also, the degrees of the polynomials in both directions in our problem will be the same ($r = s$) so the estimation problem really consists of solving the following linear system

$$Ax = b \,, \tag{4}$$

where matrix $A = [A_{kl}]_{k,l=0,...,(s+1)^2-1}$ and vector $b = [b_k]_{k=0,...,(s+1)^2-1}$ are defined as

$$A_{kl} = \sum_{i=0}^{m} \sum_{j=0}^{m} x_i^{k \,\mathrm{div}(s+1)+1 \,\mathrm{div}(s+1)} y_j^{k \bmod(s+1)+1 \bmod(s+1)} \,, \quad \text{and} \quad b_k = \sum_{i=0}^{m} \sum_{j=0}^{m} z_{ij} x_i^{k \,\mathrm{div}(s+1)} y_j^{k \bmod(s+1)} \,, \tag{5}$$

respectively, and vector $x = [x_l]_{l=0,...,(s+1)^2-1}$ contains the sought-after elements of the polynomial $[a_{cd}]_{c,d=0,...,s}$ so that $x_l = a_{cd}$, being $c = l \,\mathrm{div}(s + 1)$ and $d = l \bmod(s + 1)$. The estimation problem consists of two main parts, 1) the construction of matrix A and vector b by means of the computation in Equations 5, and 2) the solution of system in Equation 4.

3. The Heterogeneous Parallel Model

One of the most decisive concepts to successfully program modern GPU and multicore computers uses GPU and multicore processors is the underlying model of the parallel computer. A GPU card connected to a sequential computer can be considered as an isolated parallel computer fitting a SIMD model, i.e. a set of up to 512 (depending on model) processors running the same instruction simultaneously, each on its own set of data. On the other hand, CPU can be seen as a set of independent computational resources (core) that can cooperate in the solution of a given problem.

Thus, a realistic programming model should consider the host system comprising CPU cores and graphic cards as a whole thus leading to a heterogeneous parallel computer model. We follow a model similar to the one described

in [10], where the machine is considered as a set of computing resources with different characteristics connected via independent links to a shared global memory. Such model would be characterized by the number and type of concurrent computational resources with different access time to reach each resource and the different types and levels of memory. Programming such a heterogeneous environment poses challenges at two different levels. At the first one, the programming models for CPUs and GPUs are very different. The performance of each single host subsystem depends on the availability of exploiting the algorithm's intrinsic parallelism and how it can be tailored to be fitted in the GPU or CPU cores. At a second level, the challenge consists of how the whole problem can be partitioned into pieces (tasks) and how they can be delivered to CPU cores or GPU cards so that the workload is evenly distributed between these subsystems.

The partition of the problem into tasks and the scheduling of these tasks can be based on performance models obtained from previous executions or on a more sophisticated strategy. This strategy is based on small and fast benchmarks representative of the application that allows to predict, at runtime, the amount of workload that should be dispatched to each resource so that it would minimize the total execution time. Currently, we focus our work on how to leverage the heterogeneous concurrent underlying hardware to minimize the time-to-solution of our application leaving the study of more sophisticated strategies of workload distribution to further research.

4. The Parallel Implementation

4.1. The parallel algorithm

In particular, the algorithm for the construction of matrix A of order `N`$= (s+1)^2$ with a polynomial degree s can be seen in Algorithm 1. Arrays `x` and `y` have been previously loaded from a file and stored in such a way that allows to simplify two sums of Equation 5 in only one of length `n`$= m^2$. Routine `matrixComputation` receives as arguments the arrays `x` and `y`, the length of the sum (`n`) and the order of the polynomial (`s`), and returns a pointer to the output matrix `A`.

Algorithm 1 Routine `matrixComputation` for the construction of matrix A.

```
 1: int N = (s+1)*(s+1);
 2: for( int k = 0; k < N; k++ ) {
 3:   for( int l = 0; l < N; l++ ) {
 4:      int exp1 = k/s+l/s;
 5:      int exp2 = k%s+l%s;
 6:      int ind = k+l*N;
 7:      for ( int i=0; i<n; i++ ) {
 8:        A[ind] += pow( x[i], exp1 ) * pow( y[i], exp2 );
 9:      }
10:   }
11: }
```

The construction of matrices A and b (Equation 5) is by far the most expensive part of the overall process. However, there exists a great opportunity for parallelism in this computation. It is not hard to see that all the elements of the matrix can be calculated simultaneously. Furthermore, each term of the sum can be calculated independently. This can be performed to partition the sum into chunks of different sizes. Our parallel program is based on this approach since the usual value for the order of the polynomial s ranges from 2 to 20, yielding matrices of order from 9 to 441 (variable `N`), whereas the length of the sum ranges from $1,3$ to $25,4$ million terms (variable `n`), depending on how fine the grid is desired. We partition the sum into chunks, each one with a given size so they do not be necessarily equal. The application firstly spawns a number of `nthreads` where each one (`thread_id`) will work on a different sum-chunk yielding a matrix `A_[thread_id]`. We consider here `A_[thread_id]` as a pointer to a linear array of length `N`×`N`. The result, i.e. matrix `A`, is the sum of all these matrices so `A=A_[0]+A_[1]+...+A_[nthreads-1]`.

Function `matrixComputation` can be easily modified to work on these particular matrices that present the computation over chunks of arrays `x` and `y`. Everything discussed for the computation of matrix A can be extrapolated to the computation of vector b including its computation in the same routine.

Now, we consider our heterogeneous system consisting of two CPU processors with six cores each and two identical GPUs. The workload consisting of the computation of matrix A and array b has been separated into two

main parts in order to deliver its computation to both the multicore CPU system and two-GPUs system. Algorithm 2 shows the scheme used to partition the workload into these two pieces by means of the `if` branching which starts in line 4. Because it is necessary to have one CPU thread linked to each GPU, we initialize the runtime with a total of `nthreads`, i.e., as many CPU threads as CUDA devices [11] plus a number of CPU threads that will be linked to a CPU core each. This is carried out through an OpenMP [12] pragma directive (lines 1 and 2). The first two CPU threads is binded to two GPUs devices and the rest is binded to CPU virtual processors. The right number of CPU threads can be fewer than the available number of CPU in some cases. Sometimes, we got better results with a number of threads larger than number of cores since we have the Intel Hyper-Threading [13] capability activated. We have employed a static strategy to dispatch data and tasks to the CPU cores and to the GPUs, i.e., the percentage of workload delivered to each system is an input to the algorithm provided by the user. Once given the desired percentage of computation, the size of data is calculated before calling Algorithm 2 so that variable `sizeGPU` stores the number of terms of the sum (lines 7–9 of Algorithm 1) that each one of two GPUs will compute, and variable `sizeCPU` stores the total amount of terms that all the cores of the CPU system will compute. Each system will perform computation if the piece of data assigned is larger than zero.

Algorithm 2 Using multiple GPU devices and cores.

```
 1:   omp_set_num_threads(nthreads);
 2:   #pragma omp parallel {
 3:   int thread_id = omp_get_thread_num();
 4:   if( sizeGPU ) {                 /* GPU Computing */
 5:     int gpu_id = 2*thread_id;
 6:     int first = thread_id * sizeGPU;
 7:     cudaSetDevice(gpu_id);
 8:     matrixGPU(A_[thread_id],b_[thread_id],&(x[first]),&(y[first]),&(z[first]),sizeGPU,s);
 9:   } else {
10:     if( sizeCPU ) {               /* CPU Computing */
11:       int cpu_thread_id = thread_id-2;
12:       int first = 2*sizeGPU+cpu_thread_id*sizeThr;
13:       int size = thread_id==(nthreads-1)?sizeLth:sizeThr;
14:       matrixCPU(A_[thread_id],b_[thread_id],&(x[first]),&(y[first]),&(z[first]),size,s);
15:     }
16:   }
17:  }
```

Matrix *A* and vector *b* are the output data of Algorithm 2. They are the sum of the partial sums computed by each system. We use arrays of matrices `A_` and `b_` described earlier to store these partial results independently whether they were computed by a CPU core or a GPU device. Once the threads are joined (after line 17) a total sum of these partial results is performed by the main thread to form `A` and `b`. Each thread works on a different piece of arrays `x`, `y`, and `z`. Routines `matrixCPU` and `matrixGPU` are adaptations of Algorithm 1 that receive pointers to the suitable location in arrays `x`, `y` and `z` (set in variable `first` in lines 6 and 12, respectively) and the length of the sum, i.e., `sizeGPU` for the each GPU or `size` for each CPU core. These routines also include the computation of vector `b` that was omitted in Algorithm 1. The total amount of work performed by the CPU system (line 13) is divided into equal chunks of size `sizeThr` (`sizeThr=sizeCPU/(nthreads-2)`) except for the last core which is `sizeLth`, i.e., `sizeThr` plus the remaining terms. The GPU devices in our system are identified with integers 0 and 2 (`gpu_id`), which explains line 5.

4.2. The CUDA Kernel

The computation performed by each GPU is implemented in the `matrixGPU` function, called in line 8 of Algorithm 2. This function firstly performs the usual operations of allocating memory in GPU and uploading data from the CPU to the GPU kernel. Thus, it is supposed that arrays `A`, `x` and `y` have been previously uploaded into the card's global memory. For the sake of simplicity we restrict the explanation to the computation of matrix *A* since the computation of vector *b* can be easily deduced.

The construction of matrix *A* through a CUDA kernel has a great opportunity of parallelism. In this case, we exploit both, the fact that all the elements of matrix *A* can be computed concurrently and that each term of the sum

Algorithm 3 CUDA Kernel.

```
1:  #define BLKSZ_X    128
2:  #define BLKSZ_Y    2
3:  #define BLKSZ_Z    2
4:
5:  __global__ void kernel( double *A, double *x, double *y,
6:                          int s, int n, int N ) {
7:    int k = blockIdx.y * blockDim.y + threadIdx.y;
8:    int l = blockIdx.z * blockDim.z + threadIdx.z;
9:    int X = threadIdx.x;
10:   int Y = threadIdx.y;
11:   int Z = threadIdx.z;
12:   double a = 0.0;
13:   __shared__ double sh_x[BLKSZ_X], sh_y[BLKSZ_X];
14:   __shared__ double sh_A[BLKSZ_X][BLKSZ_Y][BLKSZ_Z];
15:
16:   if( k<N && l<N ) {
17:     int exp1 = (k/s)+(l/s);
18:     int exp2 = (k%s)+(l%s);
19:     for( int K=0; K<n; K+=BLKSZ_X ) {
20:       int i = X+K;
21:       if( i<n ) {
22:         if( Y == 0 && Z == 0 ) {
23:           sh_x[X] = x[i];
24:           sh_y[X] = y[i];
25:         }
26:         __syncthreads();
27:         a += pow( sh_x[X], exp1 ) * pow( sh_y[X], exp2 );
28:       }
29:     }
30:     sh_A[X][Y][Z] = a;
31:     __syncthreads();
32:     if( X == 0 ) {
33:       a = 0.0;
34:       for( int i=0; i<BLKSZ_X; i++ ) {
35:         a += sh_A[i][Y][Z];
36:       }
37:       A[k+N*l] = a;
38:     }
39:   }
40:  }
```

is independent of any other one. In order to exploit all this concurrency we used a grid of three-dimensional thread blocks. The thread blocks have dimension `BLKSZ_X×BLKSZ_Y×BLKSZ_Z` whose values are macro definitions in the first three lines of Algorithm 3. Each thread is located in the block through 3 coordinates which are represented by variables `X`, `Y` and `Z` (lines 9–11). The thread blocks are arranged in a three-dimensional grid. The first dimension is 1, and the other two are $\lceil$`N/BLKSZ_Y`$\rceil$ and $\lceil$`N/BLKSZ_Z`$\rceil$, respectively, being `N` the dimension of matrix A and `idiv` an integer function which returns the length of the last two dimensions. The following code is within `matrixGPU` function and shows this arrangement and the call to the kernel:

```
1: dim3 dimGrid( 1, idiv( N, BLKSZ_Y ), idiv( N, BLKSZ_Z ) );
2: dim3 dimBlock( BLKSZ_X, BLKSZ_Y, BLKSZ_Z );
3: kernel<<< dimGrid, dimBlock >>>( A, x, y, s, n, N );
```

The aim is that all the threads within a block calculate concurrently the core computation of line 27. The thread with coordinates (`X`,`Y`,`Z`) is assigned to calculate the terms of the sum `X`+i, with $i = 0$:`BLKSZ_X`: n. This operation is specified by the loop which starts in line 19. The exponents `exp1` and `exp2` depend on the row (`k`) and column (`l`) indexes of the sough-after matrix A. These indexes are calculated in lines 7 and 8, respectively, based on coordinates `Y` and `Z` of the thread. All the threads in the block use data in arrays `x` and `y` so, before calculation in line 27, a

piece of these arrays must be loaded into the shared memory from the global memory. Shared memory is a rapid access memory space that all the threads within a block can access. Each thread in the X dimension with Y=0 and Z=0 performs this load into shared memory copying one element of arrays x and y into arrays sh_x and sh_y, respectively (lines 22–25). These last arrays have been allocated in the shared memory in line 13. Upon completion of the loop in line 29, all the terms of the sum assigned to that thread have been calculated and stored in the register variable a. Now, this value is stored in shared memory (line 30). Therefore, a three-dimensional array sh_A of size BLKSZ_X×BLKSZ_Y×BLKSZ_Z has been allocated in shared memory in line 14.

We need to imagine the shared data sh_A as a three-dimensional cube where each position has a partial sum of the total sum. There is one sum for each element r×c of matrix *A*. In other words, elements sh_A[i][Y][Z], for all i, contain the partial sums corresponding to a given element r×c, taking into account the parity between the matrix element and the thread coordinates set in lines 7–11. Thus, it is necessary now to add all the partial sums in the X dimension for all the sums. This operation (lines 32–38) is performed only by threads such that X=0. Once the sum has been computed, the result is saved in, global memory (line 37).

We use synchronization points within the code (__syncthreads()) to make sure data in shared memory is saved before read. The use of shared memory is restricted to a small size that must be statically allocated (at compilation time). The size of our target GPU is 48KB. The maximum number of threads per block is limited to 1024. However, the total amount of shared memory is what really determines the size of the threads block. Anyway, the limitation in the number of threads per block is easily overcome by the number of blocks that can be run concurrently. Typical values for the threads block dimensions are the ones defined in lines 1–3. We experimentally checked that dimensions Y and Z should be equal. Somehow there are proportional to N (size of matrix A) and dimension X should be related to n (size of arrays x and y) and so much longer. Values of N range from 9 to 441, while values of n, range between $1,3 \times 10^6$ and $25,4 \times 10^6$ in our experiments. Given this relationship between N and n, it is clear that the opportunity for concurrency spreads in the X dimension of the block.

Just as a simply note to say that we chose the first dimension as the "largest" one due to the GPU limits, the last dimension of the threads block to 64, allowing up to 1024 the number of threads in the other two dimensions. In addition, it is possible to say that the three-dimensional grid of blocks has really been limited to an effective two-dimensional grid since the first dimension is set to 1. More blocks in coordinate X means that data computed by each block in that dimension and stored in the shared memory should be also shared among the thread blocks. This can only be done through global memory resulting in a performance penalty.

5. Experimental Results

5.1. Characterization of the Execution Environment

The computer used in our experiments has two Intel Xeon X5680 at 3.33Ghz and 96GB of GDDR3 main memory. Each one is a hexacore processor with 12MB of cache memory. It contains two GPUs NVIDIA Tesla C2070 with 14 stream multiprocessors (SM) and 32 stream processors (SP) each (448 cores in total), 16 load/store units, four special functions, a 32K-word register file, 64K of configurable RAM, and thread control logic. Each SM has both floating-point and integer execution units. Floating point operations follow the IEEE 754-2008 floating-point standard. Each core can perform one double precision fused multiply-add operation in each clock period and one double-memory. The installed CUDA toolkit is version 4.0. We use library MKL 10.3 to perform BLAS operations in the CPU subsystem.

5.2. Landform Attributes Representation Analysis

In order to validate the presented methodology and derived equations, it will be applied computing techiques for efficient solution of two dimensional polynomial regression model to represent the landform attributes of an area of São Francisco river valley region. The data source of the chosen area comes from Digital Terrain Models (DTM) [14], in the form of a regular matrix, with spacing approximately 900 meters in the geographic coordinates. The statistical analyses of the elevations indicate a dispersion from 1 to $2,863$ meters. The DTM with $1,400$ points has only 722 points inside the region, the other points are outside the area. Using all the points representing the landform attributes of the area and, by Equation 5, we estimate the polynomial coefficients for representing terrain altitude variation. The time to estimate such a polynomial in high degree needs great computer power and a long time of processing. However,

the higher the degree of the polynomial is more accuracy in the description of landform attributes representation we have, thus achieving a more satisfactory level of details.

Using the DTM data source, the solution of the model shows an elevation map generated in 3D vision (Figure 2) and a 2D projection in gray tones (Figure 3). It can also be observed that São Francisco river valley region has a heavily uneven topography so a high degree of the adjusted polynomial is needed to attain an accurate representation of the surface. Figure 2 shows the elevation map generated with the coefficients of polynomials with degrees 2, 4, 6 and 20, respectively. Therefore, by fitting a high degree polynomial to the data a better landform attributes representation and a more accurate extrapolation is obtained.

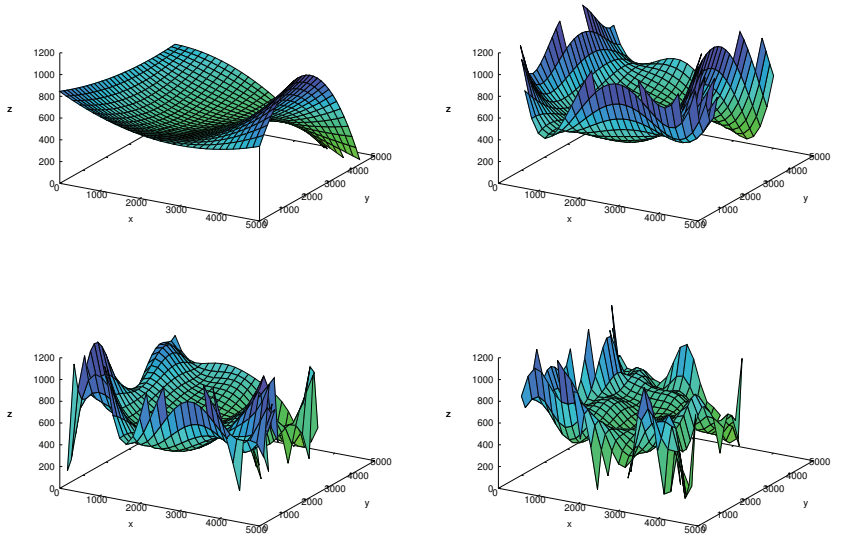

Figure 2: 3D Vision landform attributes representation of São Francisco river valley region for polynomial degrees 2, 4, 6 and 20.

Figure 3: Gray tones landform attributes representation of São Francisco river valley region for polynomial degrees 2, 4, 6 and 20.

5.3. Experiments using Double Precision Data

We have implemented a parallel algorithm in CUDA, based on landform attributes representation by using the parallel scheme proposed in Section 4 to build the surface mapping with a global fitting using the polynomial regression technique. The benchmarks were compiled with *nvcc*. In the experiments, we first increased the number of CPU threads from 1 to 24 (Hyper-Threading is set) to obtain the number that minimizes time. Then we add 1 and 2 GPUs to the number of threads obtained in the former test. The input sizes of the problem (degree of a polynomial) for the experiments were 2, 4, ..., 40. The algorithm performance is analyzed in Figure 4.

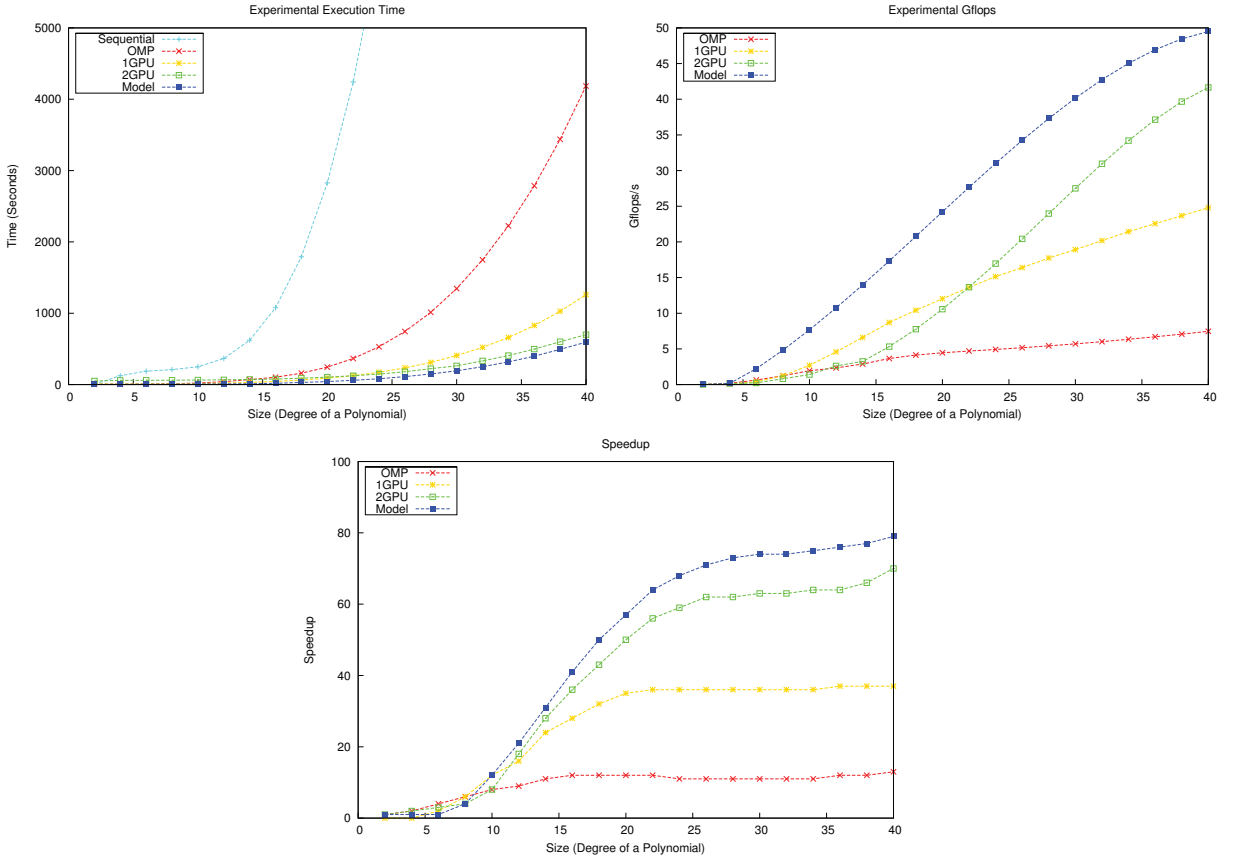

Figure 4: Graphical representation of the execution time (in seconds), Gflops and Speedup rates by varying the size of the problem (degree of the polynomial).

Table 1: Comparative performance analysis of execution time (in seconds).

Degree Polynomial	Sequential	OMP	1GPU	2GPU	Model
8	84.49	12.32	12.44	14.19	13.61
12	386.17	41.85	21.36	19.39	18.04
16	1,166.88	114.55	43.31	31.48	25.53
20	2,842.52	268.32	90.57	57.03	49.29
24	5,916.06	544.93	172.71	101.08	88.86
28	11,064.96	1,011.42	310.53	176.88	156.72
32	24,397.66	1,777.25	521.63	285.07	256.62
36	30,926.82	2,700.00	828.50	450.67	404.25
40	46,812.70	4,252.69	1,261.09	666.77	600.90

The execution with one thread is denoted by "Sequential" in the figure while "OMP" denotes the use of several CPU threads. The OMP version distributes the matrix calculation among the threads and each thread is run exclusively on a CPU core. Versions denoted by 1GPU and 2GPU represent executions in one single and two devices, respectively. The Heterogeneous Model ("Model") uses all cores available in the heterogeneous system. In this model the threads are executed by all the elements of the machine, the suitable number of CPU cores and the two GPUs. The results of the experiments show that the parallel CPU algorithm (OMP) reduces the execution time significantly. As can be seen in Figure 4 the maximum speedup is around 12, matching the number of cores. The second figure shows Gflops, presenting a difference in performance that can be observed more clearly. It must be noted that, for small polynomial degrees, the performance of OMP is larger than the performance with 1 GPU (size <= 10). This is due to the data transfer between CPU and GPU. Similarly, the performance of 1 GPU is larger than the performance with 2 GPUs (size <= 20). In this case, this is due to the setup time needed in the selection of the devices, which is high in our target machine (4.5 sec.) and is not necessary if just one GPU is used. The best result has been obtained with every resource available in the heterogeneous system. The speedup increases with the problem size reaching the theoretical maximum of 78, a number that has been obtained by comparing the computational power of one GPU with the CPU. The use of GPU as a standalone tool provides benefits but does not allow to reach the potential performance that could be obtained by adding more GPUs and/or the CPU subsystem.

6. Conclusions

The experimental results obtained in this work indicate that our approach to the solution of the mathematical model for representing landform attributes is efficient and scalable. The built application exploits the computing power of current GPUs leveraging the intrinsic parallelism contained in the algorithm. Furthermore, our solution is designed so that tasks in which the building matrix is partitioned can be either been dispatched to a GPU or a CPU core. The high computing cost of the application and the way in which we performed the solution in this paper motivate us to extended this algorithm further to other hierarchically higher levels such as clusters of nodes like the one we used in these experiments. To this end, we propose for the future an auto-tuning method to determine the best tile size that will be computed by each subsystem in order to attain load balancing among all possible computational resources available.

References

[1] L. Nogueira, R. P. Abrantes, B. Leal, A methodology of distributed processing using a mathematical model for landform attributes representation, in: Proceeding of the IADIS International Conference on Applied Computing, 2008.
[2] C. Bajaj, I. Ihm, J. Warren, Higher-order interpolation and least-squares approximation using implicit algebraic surfaces, ACM Transactions on Graphics 12 (1993) 327–347. doi:http://doi.acm.org/10.1145/159730.159734.
[3] S. Sengupta, M. Harris, Y. Zhang, J. D. Owens, Scan primitives for GPU computing, in: Proceedings of the 22nd ACM SIGGRAPH/EUROGRAPHICS Symposium on Graphics Hardware, Eurographics Association, Aire-la-Ville, Switzerland, Switzerland, 2007, pp. 97–106.
[4] F. Song, S. Tomov, J. Dongarra, Efficient support for matrix computations on heterogeneous multicore and multi-GPU architectures, Tech. Rep. 250, LAPACK Working Note (Jun. 2011).
[5] L. M. Namikawa, C. S. Renschler, Uncertainty in digital elevation data used for geophysical flow simulation, in: GeoInfo, 2004, pp. 91–108.
[6] I. Rufino, C. G. ao, J. Rego, J. Albuquerque, Water resources and urban planning: the case of a coastal area in brazil, journal of urban and environmental engineering 3 (2009) 32–42.
[7] J. O. Rawlings, S. G. Pantula, D. A. Dickey, Applied Regression Analysis: A Research Tool (Springer Texts in Statistics), Springer, 1998.
[8] G. H. Golub, C. F. V. Loan, Matrix Computations, 2nd Edition, Baltimore, MD, USA, 1989.
[9] L. Nogueira, R. P. Abrantes, B. Leal, C. Goulart, A model of landform attributes representation for application in distributed systems, in: Proceeding of the IADIS International Conference on Applied Computing, 2008.
[10] G. Ballard, J. Demmel, A. Gearhart, Communication bounds for heterogeneous architectures, Tech. Rep. 239, LAPACK Working Note (Feb. 2011).
[11] J. Barnat, P. Bauch, L. Brim, M. Ceska, Computing strongly connected components in parallel on CUDA, in: Proceedings of the 25th IEEE International Parallel & Distributed Processing Symposium (IPDPS'11), IEEE Computer Society, 2011, pp. 544–555.
[12] B. Chapman, G. Jost, R. v. d. Pas, Using OpenMP: portable shared memory parallel programming (scientific and engineering computation), The MIT Press, 2007.
[13] D. T. Marr, F. Binns, D. L. Hill, G. Hinton, D. A. Koufaty, J. A. Miller, M. Upton, Hyper-threading technology architecture and microarchitecture, Intel Technology Journal 6 (1) (2002) 1–12.
[14] M. Rutzinger, B. Hofle, M. Vetter, N. Pfeifer, Digital terrain models from airborne laser scanning for the automatic extraction of natural and anthropogenic linear structures In: Geomorphological Mapping: a professional handbook of techniques and applications, Elsevier, 2011, pp. 475–488.