CO-380

DATABASE MANAGEMENT ON GPU

BARDIERE E.
IGN, SAINT-MANDÉ, FRANCE

1. BACKGROUND AND OBJECTIVES

Nowadays, the well-known law of Moore is not followed anymore in the world of processor makers. Indeed, to follow this empiric ”rule”, in the last decade, the way to increase the power capabilities of a chip has been focused on the multiplication of the number of cores. A specific class of chips, called GPUs, have evolved to become a massively parallel computing device, capable of performing mathematical operations 2 orders of magnitude faster than the fastest CPUs available, leading to the emergence of a new discipline : High Performance Computing, a modern version of Parallel Computing.

Historically, GPU's were used to aid the CPU in very specialized tasks for 3D graphics rendering. In order to follow the continuous demand of high realism in imagery, specialized computation capabilities have evolved towards general purpose computation and flexibility, with hierarchical memory architecture, full double precision support, dedicated APIs, and many more features borrowed to CPUs. The flexibility has been brought by refined programming capabilities with low level languages (ASM like) at the beginning and high level APIs (C++ like) today. A wide range of programming languages can now be used to exploit the high computing power available in a first class GPU.

The problem we address in the present article is to find a solution to manage and display a very dense set of vector data with at least an order of magnitude in performance improvement. Since both the demand and the availability of such data for GIS applications is continuously growing and at a fast rate (due to ever increasing screens and storage sizes) we study how to implement several techniques to achieve such a speedup.

Our main objective is to create and manage a database of geographic vector data "on the fly" directly on a GPU, thus saving a high amount of bandwidth between the CPU and the GPU. Due to the fact that a part of the information treated is dedicated to be displayed on screen (another part is semantic information of each vector object, the last part being indexes or encoding of an access path to vector object data), a storage inside the memory of the Graphic Card would save several copies from main memory to GPU memory.

Our goal is to reach a significant speed-up in loading, querying and displaying large sets of vector data to achieve near real time interactivity. Typical gains reported in literature are between a 10 to 20 times more performance than native CPU implementations for several well-known algorithms. One must also mention that these improvements usually take into account the bottlenecks of transferring data from main memory to GPU memory and reading the results back. As these limitations do not exist when all the processing are done on CPU, one can expect even higher gains if storage, processing and displaying operations were all performed fully on GPU.

By studying one API in details, for instance OpenCL, we have designed the main steps of a methodology to re-write efficient GPUs dedicated algorithms for this API. State of the art algorithms used to store data are based, almost all of them, on structured trees. Nevertheless, they are often implemented using a recursive implementation, which prevents still today a straightforward port on GPUs due to hardware constraints (e.g. no stack, design focused on massively parallel architecture). In order to build these trees with massively parallel techniques, which indeed are the only way to achieve a high performance computing gain, we had to understand in details how to use GPU's features. Each step of the construction is analyzed to be the as efficient as possible on the specific hardware architecture (in particular local memory sizes, memory access constraints and hardware accelerated atomic functions).

The essential feature of a typical modern GPU that will be exploited by our method is the native SIMT architecture: Single Instruction Multiple Thread. This is a combination of SIMD (Single Instruction Multiple Data) and MIMD (Multiple Instruction Multiple Data) architectures. In fact, the smallest computing units, called threads, are grouped by 32 to become a Warp, and a Scalar Processor on a GPU can execute up to 12 Warps simultaneously. These grouping structure names and figures come from the GPU used for our work: an nVidia GTX295. Additional specifications: the GPU has 2 Stream Multiprocessor (SM), each SM contains 30 Scalar Processor (SP) and each SP can manage the execution of 16 groups of 32 threads. Finally, we have a theoretical total of: 2 x 30 x 16 x32 = 30 720 threads can be

executed in parallel. In the method, we insist on one aspect, all processes and steps must be done with the highest factor of parallelization possible.

In the following sections, we first present a kind of generic method to transform recursive algorithms in parallel ones. Some well-known algorithms are then adapted to demonstrate the method and they will finally be used for our final goal.

2. APPROACH AND METHOD

Before the GPU database building method is described, we will present the organization of the chosen data for an example of implementation. During all experiments, a VMAP 1 vector product database will be used.

In order to validate the main steps of this method, we try to demonstrate the whole process of building a static geographic vector data structure dynamically. In GIS applications, experts often consider Kd- tree(s) as one of the best structure to store static data, especially geographic ones, but again, this structure is always built recursively in literature. Specific issues of structure updates (such as insertions or deletions) are left for future enhancements of the work proposed since the method to construct the tree shall be extensively reused for dynamic updates, which is also always implemented using recursivity.

Finally, the data model used consists of a variable number of multidimensional points, geometric primitives description and a highly complex set of semantic attributes, leading to a variable number of dimensions of the tree structure. Yet, the capability to manage such data will naturally lead to a relatively easy extension of the work described below for non-geographic, general purpose databases.

2.1 First part: Adaptations needed to have the benefit of GPUs' specific architecture.

GPUs are organized to manage streams of data and not random memory blocks, stacks or pointers: with currently available APIs, it is impossible to allocate memory as it would be done on conventional CPU memory. All memory management is still today done by the CPU, the GPU simply executes static algorithms on streams as if it were a fully deterministic automat. Data streams define a continuous flow of data, usually provided into buffers, that feed compute units of the GPU, and this is one of the most important concepts required to get the maximum possible throughput from the GPU. All algorithms of the method searched must follow this requirement to be efficient enough in order to reach the performance target.

Historically, these data streams were pixel blocks (such as textures or images) or geometry primitives, commonly called vertices, or primitive definitions such as points, lines or polygons. GPUs used to embed predefined compute pipelines, for example to interpolate a geometry primitive in order to produce fragments that will become pixels at the end of the graphical pipeline. One must also note that GPUs are processors that work natively with 4 components vectors due to their historic origin of vertex and pixel processors: data manipulated has always had 4 components, a quadruplet (x,y,z,h) for 3D geometry and a quadruplet (r,g,b,a) for displaying pixels. Hence, the GPU can execute native vector operations in 1 cycle such as dot products, cross product, or any base arithmetic operation.

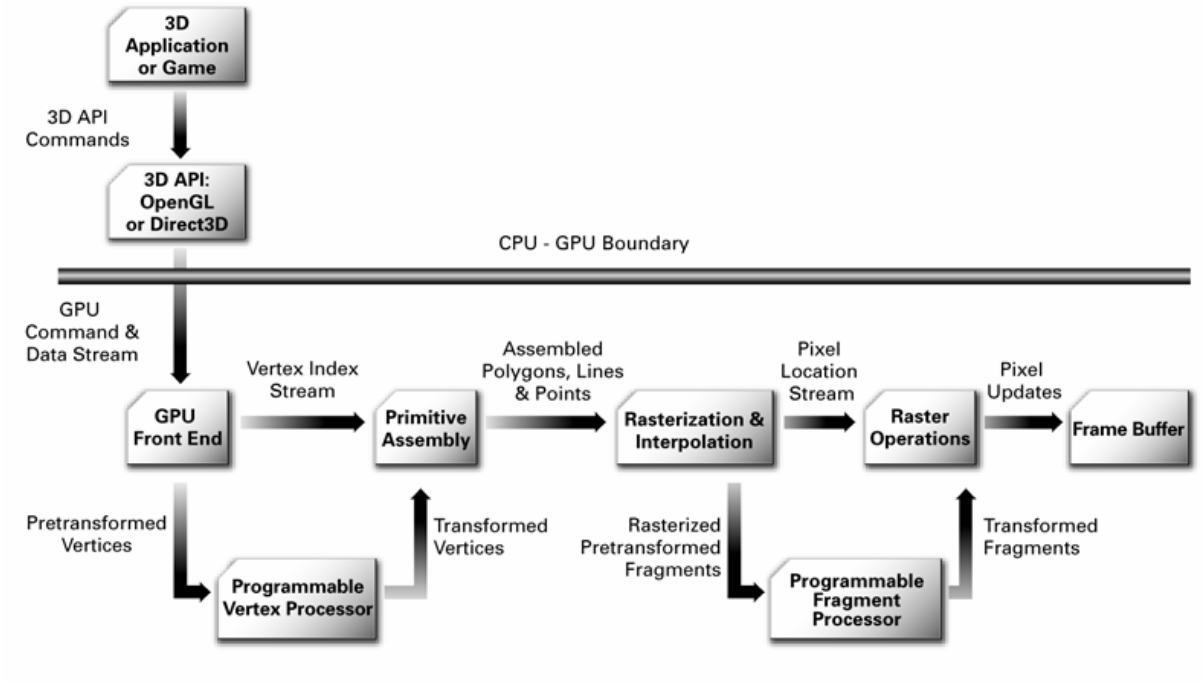

Figure 1 : Pipeline OpenGL

But with few modifications in the hardware it became possible to send data back to the main memory of the computer (for example through textures in early version of the API, or compute buffers today). Progressively, during the past decade, a large number of architectural modifications have transformed the Vertex units, the Rasterization units and the Texture units into unified multipurpose compute units. Furthermore, the GPU driver natively handles the switch between the execution of graphical rendering algorithms and multipurpose computing tasks.

In addition to the requirement of continuous feed of data streams, another key feature to reach computational performance is to optimize the bandwidth inside the GPU. CPU bandwidth on the most powerful available processor is approximately 25 GB/s, this figure has to be compared to 150 GB/s in a high end GPU like the NVidia GTX295 used here. These figures represent the capability of each chip of its class to access data located into the fastest memory level accessible. At first glance, if bandwidth is the bottleneck of an algorithm, in the most favorable possible configuration a GPU can access 6 times more data per second than a CPU on a single stream.

<TODO : schémas architectures mémoire>

As mentioned above, CPU and GPU both have today a comparable hierarchical organization of memory. Some benchmark tools available on the market can precisely measure such a bandwidth for a CPU, but it is more complicated for a GPU since the measurements will include bus transfers, global to local memory transfers and non-coalescent accesses into local memory blocks, which physical properties depend on the GPU architecture version. An important difficulty when writing algorithms is to properly optimize the usage of the different kind of memory blocks: local and global. In particular, different blocks have very different access speed: local memory registers can be written in approximately 8 cycles while global memory requires around 400 cycles according nVidia reference. Furthermore, these access times vastly depend on pattern in which threads use memory addressing

<TODO: schemas des acces memoire dans les differents cas>

Finally the PCI Express bus has a bandwidth of approximately 6Gb/s, which is the bottleneck when moving data from main memory to GPU's memory. This limitation is the second major drawback of the processing chain that consists in fetching data from a vector database (the most favorable case is when data is partly cached into main memory) and transferring it to the GPU for display. Indeed a first level of optimization would be to cache the most frequently accessed data into GPU's memory. But this leaves unaddressed the main issue of querying data and updating the cache in GPU's memory, which is by far the slowest operation when new data need to be retrieved from hard drives.

The bandwidth considerations above naturally require that both the data and the queries be located into GPU's memory in order to achieve the fastest possible display. In order to save as much bandwidth as possible, we decided to transfer the whole database in the global memory to the GPU (indeed the

maximum possible number of tiles if the database size is greater than the GPU memory size, one can easily imagine a background lightweight process responsible of maintaining the most appropriate tiles available on the GPU). For the experiment, we stored a whole database tile by tile on the GPU which memory is 1024Mo, but some GPUs available on the market have as much as 6Go available, and some features also enable to have several GPUs coupled on a single system (such configurations are not presented in this work since it is a hardware scalability evolution that leaves unchanged the interface to the computing device from the API, basically, only more stream processors and more memory are available). One can easily imagine that in a near future, GPU boards will offer enough memory to be compared to a SSD hard drive for example.

Obviously, when trying to address the sum of all the constraints described above, it became clear that the method searched would have to rely on an hybrid system : a CPU that would be responsible of memory management and algorithm selection, and a GPU on which we would rely for its computing performance through a massively parallel architecture. It is commonly admitted that high performance computers have adopted the schema (c.f. top500 supercomputers).

2.2. Second part: Transformation of Algorithms

2.2.1. The API

Before the effective writing of algorithms, we needed to validate the capabilities of the chosen API in order to be able to completely implement a multidimensional generic database on the GPU.

Two APIs were short listed: CUDA, property of nVidia but highly optimized for High Performance Computing and OpenCL, a more versatile API which has the interesting capability to execute a “kernel” on different platforms: several hardware support this API: CPUs, DSPs, GPUs, Cell processor, tomorrow APU, etc... . In OpenCL terminology, “Kernels” are a dedicate name to specify the part of the algorithm source code which must be executed by the compute unit (here the GPU) and is very similar to a function name in C.

Indeed, with CUDA, programmers can only use nVidia compiler, but a complete development environment is available today, fully integrated in IDEs with the most advanced tools such as debuggers or profilers. Nevertheless, OpenCL has been introduced with several common features with CUDA, and could then benefit from strong concepts and foundations of GPU computing laid down by this APIs and previous initiatives.

A simple program using OpenCL has the following organization in its source code:

1. Selection of the most appropriate hardware to execute kernels by choosing the compute unit in the list of supported computing platform listed above. It is also possible to exploit different hardware at the same time in the same application.
2. Initialization of computing devices, environment and execution queues (queues are FIFO stacks of kernels).
3. Creation and build of programs, followed by JIT compilation (just in time) of the kernels.
4. Initialization of data used by kernels: data buffers stored on the GPU and parameters passed to kernels (any type, including pointers to data buffers).
5. Kernels are then sent to the chosen computing units on an execution queue.
6. Query of data results by copying GPU memory buffers to host memory.

The steps described above are all controlled by the CPU, even if the CPU can also be a computing platform from the OpenCL point of view, a hybrid architecture CPU + GPU is mandatory.

2.2.2. From Recursivity to parallelism

The major issue we faced to transform algorithms for the API and the GPU architecture was to migrate purely recursive algorithms to fully parallel ones. The typical concise and elegant writing of recursion on tree algorithms is indeed hiding the complexity of the migration: recursivity defines a strong sequential set of operations whose dependencies (through the implicit stack ordering) make it very difficult to separate algorithm steps into independent one.

Here is a brief presentation of a method for building a tree (a Kd-tree) with the traditional recursive method. Kd-trees are an instance of more general structure called Binary Space Partition (BSP). Quad trees, Octrees and their generalization are another form of BSP, but unlike the Kd-Tree, their structure is not equally balanced in each dimension and for each level. The construction of Kd-trees is done according to the following rules:

- The root contains the whole space, i.e. all elements of the structure can be found under the root element

• A dimension is chosen, called the main dimension, other dimensions are called accessories dimensions. Using a specific heuristic, the space is divided in two sub-spaces containing the same number of elements. The two sub-spaces are usually called right/left node, which means we introduce a relation order in the structure of the tree. In the work presented here, we used spatial ordering of elements, but any semantic attributes of vector object could have been chosen as a partition of space, the number of dimensions can be the number of attributes, and any relation order can be chosen for each dimension.

• If the number of elements in each node is less than a configuration parameter, the building stops and this node is a leaf of the tree. If not we call recursively the step 2 on the two child-nodes, and at each level another dimension of split is selected.

• To finish, the last node (in fact, we call it a leaf) is made by more than one element, we can build an annex structure to store them. The number of elements if the leaves must be selected in order to minimize the constraints for a parallel processing of them.

Then, classic algorithms need to be completely redesigned, and first tentatives are often less efficient when executed on the CPU (which is not a massively parallel execution unit despite several physical and virtual cores). A first approach to transform a recursive algorithm is to remove the dependency to the stack. The classical trick is to use a stack for each step: in a loop (until the stack is empty) we store the next operation or the list of next operations, which in the case of a tree is for example during the processing of each node to store the list of child nodes to process during the next iteration. The hardware stack (that does not exist on a CPU) is replaced by a far more controllable one in a slightly modified version of the same algorithm.

Once all steps are available in the new stack, in some favorable cases (when they can all be processed independently) they could be processed in parallel, for example with the OpenMP API on CPU. But there are still two important gaps in performance:

1. The usage of a stack cannot be ported to the GPU with OpenCL, as it requires dynamic memory allocation or complex fixed buffers management but this is not generic for various data base sizes

2. This parallelism brings improvements on the CPU by a factor proportional to the number of available cores and available steps on the stack, but this is still far from true parallel computation and does not use the performance available on the GPU.

Such methods keep almost unchanged the classic algorithms, which requires a very low development effort with gains up to a factor of 4 in our work

<TODO: present OpenMP improvements>

A step forward in improvement was to completely rewrite algorithms for the construction and exploitation of the database tree and the following section presents the method and algorithm we have selected.

2.3 Third part: Description of the method to build a kd-tree with non-recursive method

As mentioned previously, we decided to create a specific structure to store the elements of a leaf.

Each leaf contains an array of points and not a single point (an n-dimensional point), which produces a smaller tree with less nodes (the ratio of space used by the structure when compared to the data stored is more favorable). A second advantage of the structure is that the size of the lists in the leaves is selected to maximize the usage of local memory. As explained before, local memory of a Scalar Processor has a much higher bandwidth and is rather independent of the number of reads of writes. In addition, using the maximum local memory possible produces less coalescent accesses

<TODO: metre shema accès memoire nVidia>

The creation of the Kd-tree has the following steps:

1. Sort data using a well-known algorithm: Bitonic sort (or Batcher sort). One sort is performed on each dimension. Today, we know (ref KNUTH) that the best sorts have a complexity of o(n.log(n)) in the best case, like Quicksort or insertion sort, but on GPU, the parallel implementation of the Bitonic sort has the same complexity. Efficiency can even be better if the native vectorial nature of GPU processors is also exploited (see Intel implementation of Bitonic Sort)

2. For one specific dimension, find the median, i.e. the value that splits the number of elements into 2 equal sets. The method used to find the median also needs to have a parallel algorithm, because each portion of code that executes sequentially on a data stream can be slower than its equivalent on the CPU (for example because of very different base frequencies)

3. We then split in parallel the data according their position when compared to the median.

4. Repeat from step 2 on another dimension until the number of points of each leaf (the Kd-tree is fully equilibrated) occupies the maximum of the local memory available on the device (in order to maximize the bandwidth).

We have considered that first step of sorting must be the most expensive processing, and then we decided to build the tree with only one sort in each dimension sequentially. No other operation will sort any element of the tree, and after the first step, initial data will not move anymore. Then the difficulty is to split data on a specific dimension without sorting it again.

To solve this issue, the have exploited the most important concept used in this work which consists in using a massively parallel technique known as “scan” (or prefix sum) in all algorithms of each step.

After each initial sorting operation, we store in an additional vector the index of each element according to its position in the sorted vector. Then the structure of the tree consists in a set of n vector of indexes (one by dimension).

<TODO: indiquer le ratio taille du Kd-tree (tous les buffers utilises par rapport à la taille des données>

Now, we consider a primary dimension, other dimensions are called secondary dimensions. In order to

set up the prefix sum algorithm on the current dimension, we use a vector of flags (Ref Blelloch) with a fixed size suited to get the best bandwidth on our development hardware. The values of the flag vector are computed by answering to the following question for each element: “does the index of the sorted vector is less than the median value?”

Below is an example in two dimensions

0 1 2 3 4 5 6 7 indexes of points in the original tile 10.5 29 7.6 35.4 25.1 3.2 13.7 21.4 values on a given dimension 2 6 1 7 5 0 3 4 ranks of the values once sorted

Once the rank of each value is computed, we can perform the split on one dimension : The ranks in the above example are : 2 6 1 7 5 0 3 4 According to a median value of 4, the flag vector has the following values : 1 0 1 0 0 1 1 0 We use an additional trick to compute the split : the flag vector will be used for the first node and a dual vector will be used for the second node, the dual values are then the following :0 1 0 1 1 0 0 1

Then, prefix sum are computed on both flag vector:

VSP = 0 1 1 2 2 2 3 4 WSP= 0 0 1 1 2 3 3 3 + and by adding the median value we obtain: 4 4 5 5 6 7 7 7

The new use the prefix sums computed above: 26175034 21036754

21036754 10.5 7.6 3.2 13.7 29 35.4 25.1 21.4

We compute the prefix sums of the vector and its dual (according to the technique witch consist in splitting big blocks, which is not detailed here but is essential to keep peak performance with constrained local memory sizes). The position of an initial point in the final vector is obtained by the value of its index in the prefix sum Vector. The relative order of the elements in the sub-space is not modified. The second sub-space can then be “normalized” to have only values between 0 and N_elements - 1by subtracting the median as in previous example. 2 1 0 3 6 7 5 4 => 2 1 0 3 | 2(=6-4) 3(=7-4) 1(=5-4) 0(=4-4). 21036754

10.5 7.6 3.2 13.7 29 35.4 25.1 21.4 =) 21032310 10.5 7.6 3.2 13.7 29 35.4 25.1 21.4

The next step consists in passing this splitting and rearranges indexes of secondary dimensions and to transform them to have “normalized” indexes. We still use then the vector flags / scan technique. But this time, instead of using the median heuristic to create the vector of flags, we used another one. It consists in filling a vector which size is the same than the original space and to attribute an 1 at each value taken in the dimension index of the sub-space, and a 0 if not. Note that for the second subspace, a dual vector is easy to build, this can be done by a parallel algorithm. The two prefix sums are now computed and the result is used to rearrange the index of the secondary dimensions (An example is given below).

3. RESULTS

We achieved the building of such a database fully on the on GPU, using at each step a parallel method. We have also used the structure (kd-tree) to make requests on the database. Gains are notables, mainly on the querying part but we can note that building time is only done once for a tile because data are static. In case of reusing the structure the time required to build the structure is null! Global data are stored in a special database organized on the principle of a pyramid of tiles. The first level is the biggest one and contains the whole earth. The file format is the VPF Vector Product Format declined in VMAP0, VMAP1, etc ... All algorithms can be Massively Parallelized. Using vector of flags and computing his dual in the same time improve notably time of computing, building two vectors with one function of the kernel.[F1]

4. CONCLUSION AND FUTURE PLANS

Working with GPU as naturally as it is done today with CPUs is a great challenge. We have to change our way of thinking on algorithms mainly on recursive algorithm used to build typical structures to manage geographic information.

Rather than sorting once by dimension, we could have implemented another algorithm in o(n) to find the median value known as median of median (also a recursive one). But his improvement requires that our structure is modified to be more dynamic.

For example, how can we add elements in the database? How to remove elements? Will the structure stay balanced (which is an essential feature of Kd-trees that must be maintained for parallelism efficiency)? Do we need to have a highly complex structure to be self-balanced? Adaptation of other generalization algorithms like Douglas-Peucker (also a recursive method) is also needed to achieve the whole process. We also present a different technique to compute the bounding box of each "branch" and leaf to avoid empty space in the structure: this is classical disadvantage of equally partitioned space tress, but the structure is then not anymore a BSP.

Finally, to validate that gains are effective, it would be worth testing the method on other platforms. The unification in a single chip like the FUSION project from AMD witch today provides APUs and the new SANDY Bridge or Larrabee (Intel) will provide more and more capability in processing huge amount of points, polygons. Finally, we could make an evaluation of complexity of the algorithms used here on these new unified architectures.

Prefix sums are computed by summing each element with its precedent, assuming the first to be 0.

The last element contains the whole sum of the initial vector execpt the last value.

Ref CUDA scan. (In fact this operation is called a scan because the principle is the same for each binary operator, plus, minus, max, min, etc...), ref :http://http.developer.nvidia.com/GPUGems3/gpugems3_ch39.html

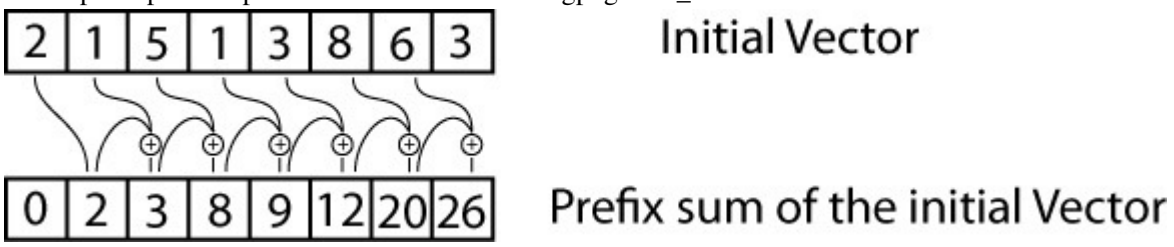

Figure 2 : Presentation of Prefix Sum Principle

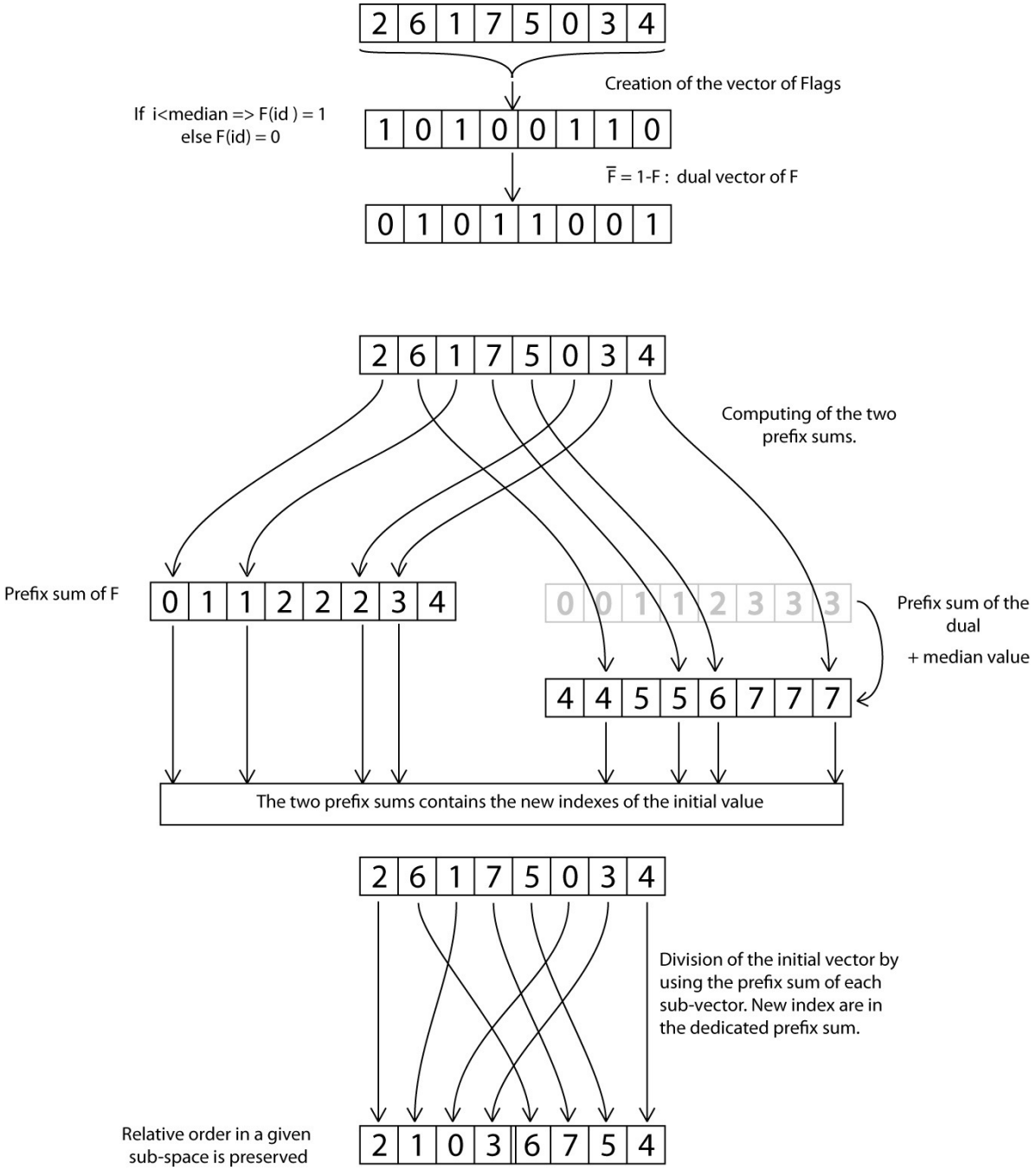

Figure 3 : Splitting on major dimension

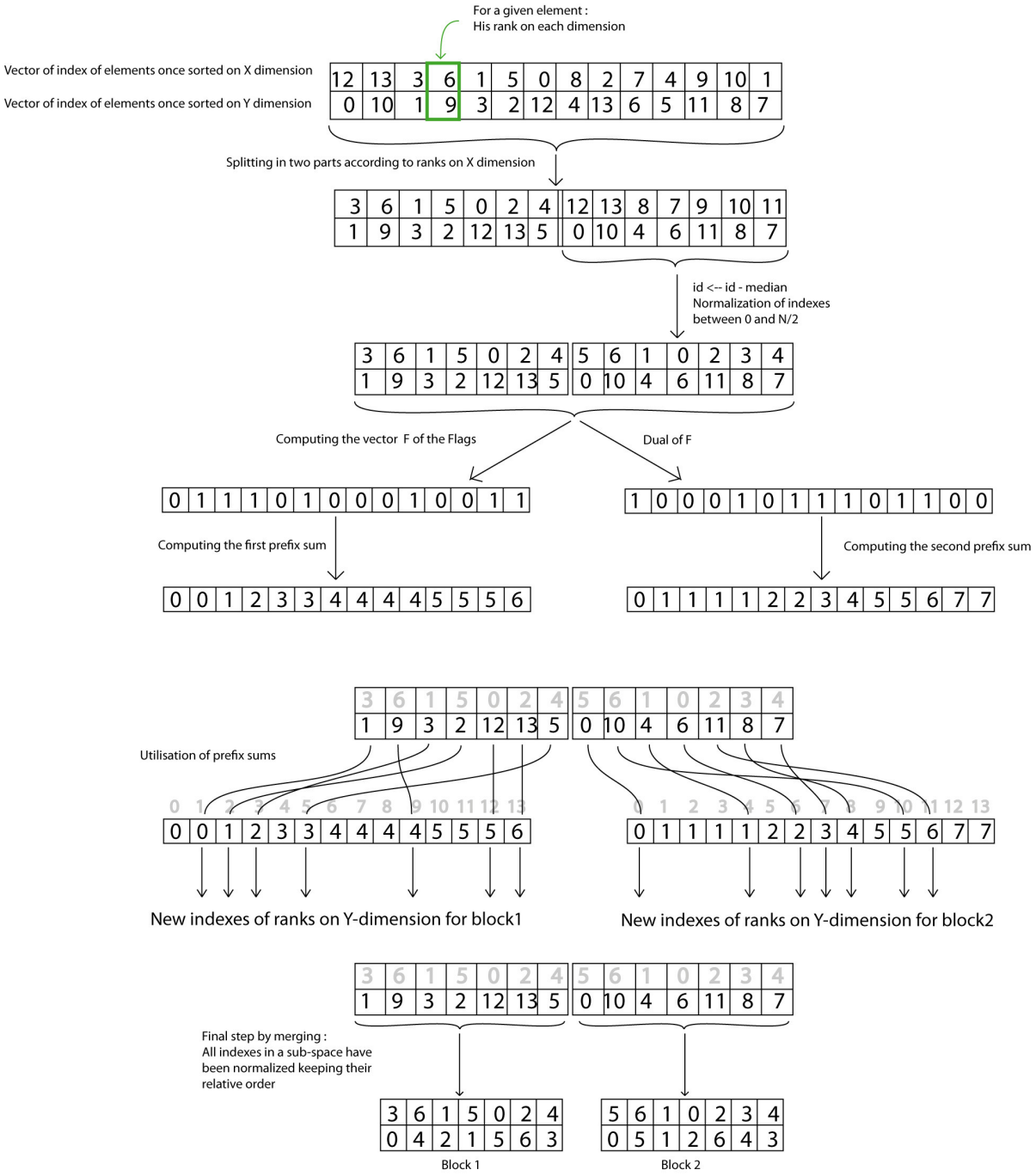

Figure 4 : Other Dimensions repercution

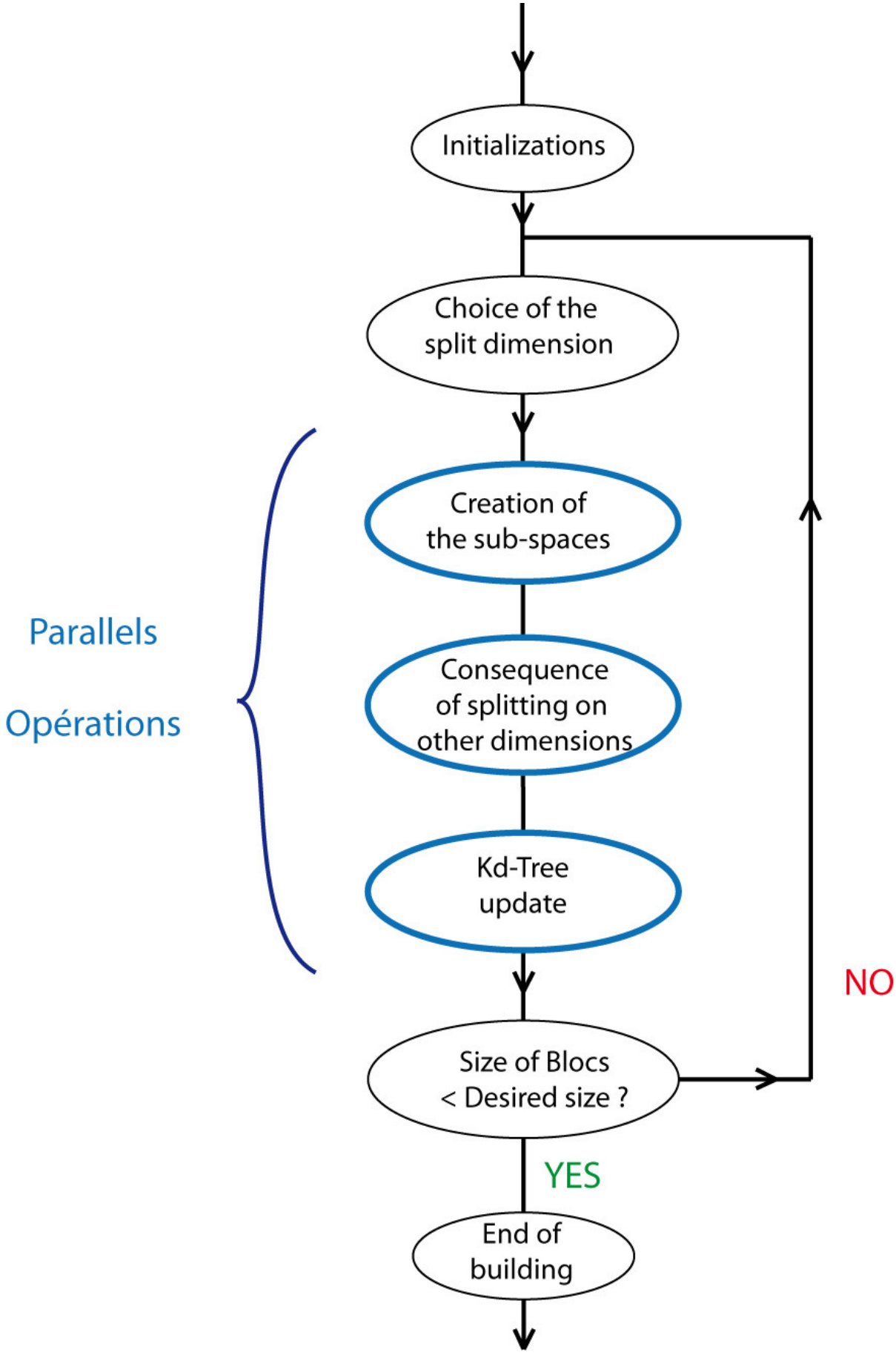

Figure 5 : Operations on building Kd-tree