More scalability at lower costs – Server Architecture for Massive Multiplayer 3D Virtual Spaces powered by GPGPU

Alin Moldoveanu, Florica Moldoveanu, Victor Asavei

***Abstract*—**3D massive multiplayer virtual spaces are getting more and more popular, not only as computer games but as complex simulation and interaction environments, heading to become the next paradigm of multi-user interface. Still their universal adoption is hindered by some serious practical issues, mainly revolving around development costs and scalability limitations. The authors consider that the main cause for these limitations resides in the particularities of server-side software architectures - traditionally designed as clusters of single processor machines. The paper gives a brief overview of current solutions and their limitations and proposes two innovative architectural concepts which have a big potential for creating cheaper and more scalable solutions. We describe a region based decomposition of the virtual space together with supporting middlewares of messaging, distributed control and persistence, which allow an efficient and flexible work effort distribution on server side. The solution allows for both horizontal and vertical scalability. The vertical scalability is then mapped in an innovative manner on the last generation of SIMD-like multi-processor graphics cards. The huge processing power of these cards, with the right architecture, can take over the bulk of the server-side effort. Our prototype tests indicated that the solution is feasible and may represent an important turnaround in the development of more scalable and much cheaper massive multi-player server architectures for various types of virtual spaces.

***Keywords*—**3D, virtual space, massive multiplayer, server scalability, GPGPU, CUDA

I. Introduction

3D massive multiplayer virtual spaces offer rich information delivery, interaction and collaboration possibilities. They are getting more and more popular each day, not only as computer games but also as simulations, training, edutainment or work-oriented applications [8]–[10]. Considering the huge attraction and interest from the users, it is highly probable that not far in the future they will be the standard user-interface paradigm.

Right now, the market adoption of such virtual spaces is hindered by the huge costs involved in their creation, operation and maintenance. Creation costs regard software development and content creation. Content creation is basically open to everyone, with many easy to use tools available, but the software creation is prohibitive.

The architecture for such applications is typically client-server. The server manages the content of the virtual world while clients provide access to it for remote users, through internet.

While client applications are well supported by libraries, graphics, physics, audio, multiplayer etc. engines and RAD tools, this does not stands for the servers for massive multiplayer 3D virtual spaces.

Particularities of the domain are:

- the real time multi-player aspect
- the complexity of interactions
- requirements about persistence and up-time
- (most important,) the short response time (round trip latency) that users expect

All of these, put together, will make the development of a 3D multi-player virtual space a challenging task, accessible only to a selected elite of software developers and, of course, very costly.

Operation and maintenance cost tend to be very high due to the traditional approach to server design for this type of application.

According to our researches, there is no highly scalable accepted solution of 3D massive multiplayer virtual spaces server in the public domain

Judging by the little confidential information that leaked out, it seems that even biggest operators of such spaces don't have a perfect solution and they are running huge operating costs because of this.

The paper proposes a high level architectural solution designed for high scalability, both horizontal (on multiple computers) and vertical (on multi-processor machines). Then we show how the vertical scalability can be mapped to benefit from the huge parallel power of the new generations of graphics processing units.

II. Server Responsibilities and Issues

A. Server responsibilities

The vast majority of 3D massive multiplayer virtual spaces are implemented as client-server applications. This comes

Manuscript received December 15, 2008

All authors are in the Faculty of Automatic Control and Computers, University Politehnica of Bucharest, Bucharest 060042, ROMANIA
http://www.pub.ro/
http://www.acs.pub.ro/

from the competitive and security issues existing in most of them. Even if, theoretically, some virtual space can also have efficient peer to peer solutions, this is excluded in practice, because it will be possible, actually quite probable, that some users will try to exploit the inherent weakness of the architecture by reverse engineering and modifying the client, as is actually happening with the clients of most successful MMORPGS.

Hence, the first role of the server, which force the client-server approach, if to be the unique supervisor and arbiter of the virtual space.

To accomplish this role, the server must [3]–[5]:

- keep a full semantical representation of the internal state of the virtual space
- validate the significant actions of each user according to the virtual space rules, to prevent exploits by reverse engineering the client
- detect patterns of bot-like actions and possible frauds and stop them etc.

Besides this arbiter role, typical responsibilities of the server are:

- characters/avatars creation and changing
- login system
- implementing the non-user controlled aspects of the virtual space logics:
  - events
  - NPCs (non-player characters)
  - changing geography
  - resources
  - mobs creation
  - mobs control
  - quests
  - artificial intelligence
- permanent updating each client about all the events of interest for him

B. Scalability issues for the server

As we will argument below, the scalability of a massive multiplayer virtual space server is substantially different from other classical scalability problems [1].

For example, for applications consisting of highly intensive more or less complex database accesses and queries, the scalability is almost entirely solved by classical techniques at database or file system level. There are no synchronization problems that can't be solved with the basic locking of records or tables. There are no real time challenges. The clients for these applications also have usually small complexity, they just acting at presentation level - GUI.

Other applications, like search engines, rely on specific algorithms for dividing a query in several smaller ones, which is easily distributed over several machines, the results of the independent sub-queries being relatively easy to assemble also.

Most of these applications do not have to address the issues typical for a massive multiplayer virtual space:

- permanent connections to a large number of clients
- high amount of information exchanged in real time (users actions and their effects)
- the absolute demand for a fast, guaranteed response time to users actions (round-trip latency)
- the relative complexity of the logics and interactions in the virtual space, which usually implies:
  - there is no straightforward full hierarchical tree decomposition of the problem
  - even if a decomposition is found bases on some criteria, the resulting subtasks will require some amount of communication between them

1) Low latency

Virtual spaces are mostly used in entertainment (games). From other uses, they also make up good simulation and training environments. In both cases, the users do expect and need the response to their actions and to other users' actions in real time, as fast as possible. A usual value is in the rank of tens to hundreds of milliseconds, anything above affecting the quality of the interaction (is usually called lag and hated by the users).

This is so important that some specific techniques like "guessing" the results of an action or the expected next position during movement, followed by later corrections or adjustments when necessary, are sometimes employed to minimize latency, as observed by the user.

Such techniques are anyway more or less just workarounds or tricks. Above them, is essential the architectural design of the whole system to minimize the latency.

2) Strongly linked multi-user interactions

Unlike other application types, where the work flow is initialized and directed by a single user, virtual spaces support complex interactions, involving many users at the same time. The actions of an individual user and their effects, usually calculated by the server, must be propagated in real time to all those affected by them.

Obviously this requires a communication architecture with support for broadcast. This has no impact over scalability

However, of maximum importance for scalability is the accomplishment of the following thumb-rule:

- the division and distribution of users and tasks over different processing units (processor or computers) must allow very fast retrieval of the necessary information about all the users potentially affected by an action

This requires:

- designing the scenario of the virtual space, the geography of the world, the possible interactions between users and the quests
- the internal data structures on storage
- the server communication middleware

III. Limitations of the Current Solutions

Due to legacy background and education, most massive multiplayer virtual spaces developers are oriented to classical

architectures, with a very small number of processes or threads, since the time of single-processor computers. Hence, first scalability attempts followed this approach, ending to be more or less like some partial work-around for the problem.

A. Virtual world independent instances

Totally independent instances of the virtual world, usually called realms in games terminology, are created and reside on completely independent servers. Users can access any of these instances, but, as any moment in time, the interaction is limited server-wide and generally, character/avatar transfer between worlds is restricted or limited. Sometimes, minimal connection between these world exists though, but they are minimalistic and don't have real time behavior.

B. Instance dungeons

A similar workaround is the instance dungeon concept – a subspace of the virtual space, for which an independent copy is spawned each time a user or a group of users is accessing it. Users in a dungeon can only interact with others in the same dungeon.

From users' point of view, this limits the competition for resources in that area to an acceptable degree.

However, from developers point of view is a tricky way to:

- reduce server workload by limiting the number of users in an dungeon and therefore reducing the overall number of interactions
- divide the server effort, since the dungeons are quasi-independent and they can be hosted on different computers.

C. Static spatial decomposition

Sometimes, the geography of the world can be exploited or, even more, can be designed to be decomposed in several totally-independent regions. For example a virtual world made of planets, or islands, or having some non-passable natural barriers can benefit from this idea. The transition from one region to another is done through some key points and usually is not instantaneous.

Hence, the server workload can naturally be divided by these regions, each being handled by a different processing unit, usually a different computer.

This approach is a nice workaround for many cases; still it has some serious limitations or disadvantages:

- the resulting virtual world lacks total spatial continuity
- some computers corresponding to overcrowded regions may not be able to handle all the workload, resulting in lag, while others, for empty or scarcely populated regions, are not used at their capacity
- to obtain realtime speed, the processes that handle each region are usually defined and statically allocated on different computers; transferring a region process from one computer to another might be costly in terms of runtime speed and is usually avoided - limitation to good load balancing

It is obvious that this solution is still a workaround to the real scalability issue.

IV. Dynamical Spatial Decomposition

We suggest a more powerful approach, which would divide the virtual space into regions dynamically, at runtime.

A. The decomposition method

It is based on some heuristic function that calculates the workload for a region. When the function value exceeds a threshold, the region can be subdivided into smaller regions, and so on.

The heuristic can take into account:

- number of users in the region
- number of mobs in the region
- number of inanimate objects in region
- dynamics of interactions, etc.

The dynamic decomposition will have a tree-like structure. A binary or quad-tree fits perfect this idea.

Quad or oct-trees are common structures use in many 3D applications, exactly for their ability to easily and flexibly decompose the space according to the different criteria of each application.

The decomposition starts by seeing the whole 3D virtual space as a top region, which will be divided into sub-regions, which in turn will be further divided and so on.

The decomposing process should stop when each region computes a value for the heuristic lower than the threshold, or the region dimensions are lower than a given constant, usually depending on the users and mobs perceiving range.

In practice, the algorithm must take care of various aspects like:

- users or objects passing from one region to another
- users or objects situated on the edges of two or more regions

The mechanism is exemplified in Fig. 1: the dots represent the users, the heuristic is very simple, equal to the number of users within the region, quad-tree is used and the decomposing threshold is 4. Basically, a region will be divided when noUsers(region) >= 4.

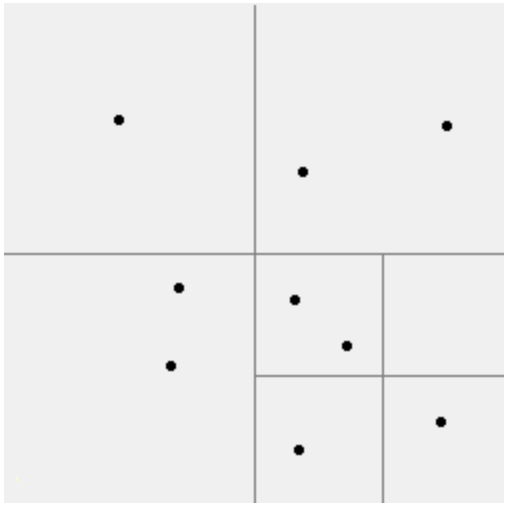

Fig. 1 spatial decomposition

This decomposition allows:

- reduction of the workload, as each users actions targets will be tested and their effects calculated only on the users in its region (or neighboring, if user is on close to border)
- division of the workload across several computing units, each region or groups of regions being assigned to different processing unit

The division of the workload works like this:

– a computing unit handle a number of regions, usually from same branches
– when the estimated workload exceeds its computing capacity, it will delegate some of the regions to a difference computing unit

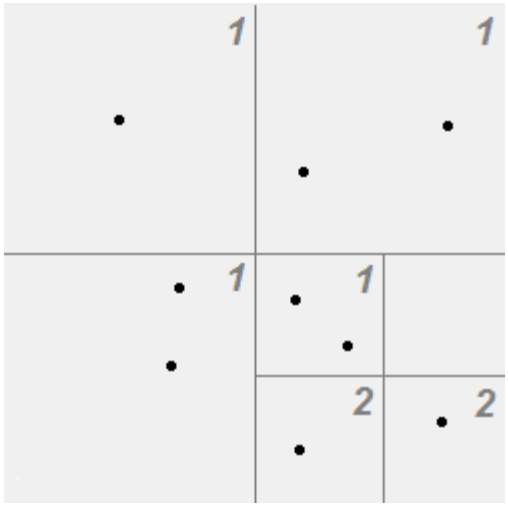

Fig. 2 division of the workload

A computing unit can be either:

– an independent computer from the server network (horizontal scaling)
– a processor from a multi-processor machine (vertical scaling)

Fig. 2 illustrates this process, building on the decomposition from Fig. 1: at the moment of the last division, it is also decided that the processing unit 1 cannot handle all the workload, therefore some of the resulting regions will be assigned to processing unit 2.

The division in regions, and also the assignment of regions to processing units can be highly dynamic.

When the heuristic workload estimation value for a previously divided region goes back under the threshold, its sub-regions can be grouped back into the original region to reduce the tree.

B. Required Middlewares

To support this highly dynamic division and load balancing possibilities, some underlying middlewares are required. They must allow:

– fast transfer of users between the regions processing tasks that run on different computers
– fast transfer of a region data from one computer to another
– scalability

As this paper focus mostly on the decomposition and on scalability issues, we will just give a brief overview of the needed middlewares and our solutions for each of them:

1) Control layer

The conceptual model of a virtual space is highly event based - basically all changes in system state are caused by users' actions.

Therefore, the control layer will be responsible only with initializing the system and supervising the regional division and eventually load balancing [2].

Tasks not handled directly by the regions (e.g. commerce) will be separated from the control layer with a plugin system.

2) Communication layer

Considering the nature of the system and the scalability requirements, we consider that the best choice will be a publish-subscribe messaging system.

This is basically a paradigm of communication through asynchronous messages where transmitters (also names publishers or producers) do not sent the messages to a specific destination. Rather, the messages are classified in classes and have attributes. The subscribers declare their interest for some classes of messages or having some attributes and will receive only the corresponding messages.

Main advantage of this method is the total decoupling between producers and consumers, this usually allowing for great scalability

There are many variants and implementation topologies for this paradigm. Without going into many details, we have decided that best choice is:

– message filtering based on type and content, with the possibility of specifying very strong filters
– domain server topology
– low level networking service tuned to transmit very fast the small sized messages, these being the huge majority of the messages in our case

With publish-subscribe, moving a user from a region to another comes to changing subscriptions [6], updating the domain server and sending a message with essential real time user data.

3) Persistence layer

This layer is used just to save periodically essential system state.

For example, in a MMORPG, the will be saved:

- existing characters
  - name
  - level
  - skills
  - items
  - values for life, mana, etc.
  - progress in various quests
- virtual world geography, is dynamic
- the state of the essential non-player aspects of the system

Traditionally, the persistence layer works more like a backup system: all the above elements are saved periodically to the system database as some fixed time intervals. This approach is due to the slowness of database management systems compared to the realtime requirements of most 3D virtual spaces.

The only advantage is that it simplifies the architecture, by separating the persistency from the real time logic of the system.

There are however two important drawbacks:

- in the case of some system failure, either hardware or software, important realtime data can be lost, only solution being to restore the virtual space to a previous state
- scalability is also affected: considering the workload and system data is distributed over several computers, when one of them stop working because of some hardware failure, then data needed by other modules is lost also so the system as a whole needs to be restored.

We consider that recent advances in database engines mark the time to switch to a full realtime allow its extended use to store real time persistency, with the help of some intelligent caching system.

In such case, data that is used by the regions will also be replicated in the database, thus allowing easy transfer of the

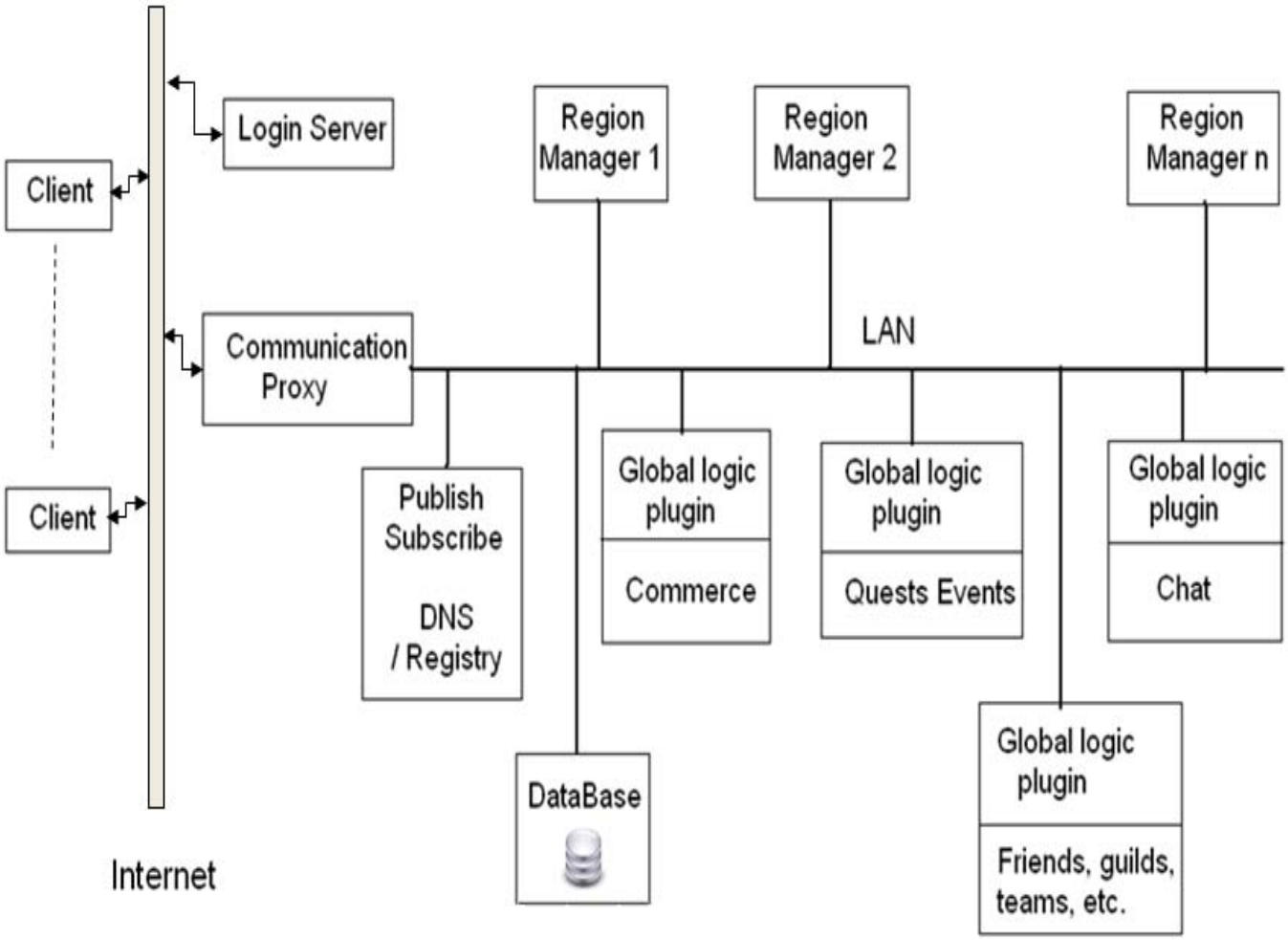

Fig. 3 scalable server for 3D massive multiplayer virtual spaces with spatial decomposition, publish-subscribe messaging and plugins system

regions tasks from one computer to another, either as normal functioning of the system or to take over from the failure of some computing unit.

4) Plugins

Designing the virtual space server as a basic infrastructure for plugins and implementing all the actual logics and functionality would offer huge advantages related to modularity and reusability of the logics.

Plugins can also migrate freely from one computer to another which can increase scalability.

There will be two categories of plugins:

- Basic/predefined, covering all the basic functionality:
  - zone managers
  - virtual world geography
  - in-game resources management (gold, minerals etc.)
  - player (level, skills etc.)
  - inventory
  - combat
  - AI
  - mobs generation
  - mobs control
  - marketing
- particular plugins, specific to each game
  - 3rd party ones, for example:
    - vehicles control
    - special effects
    - weapons
    - common quests
  - Implemented by the developers

Considering their instantiations, plugins can also be divided in two classes:

- global plugins: have an unique instance throughout the system
- regional plugins: have an instance on each region manager

C. System architecture

Considering the elements describes above, a high level view of the system architecture is illustrated in Fig. 3.

The structure of a region manager, with regional plugins, is shown in Fig. 4.

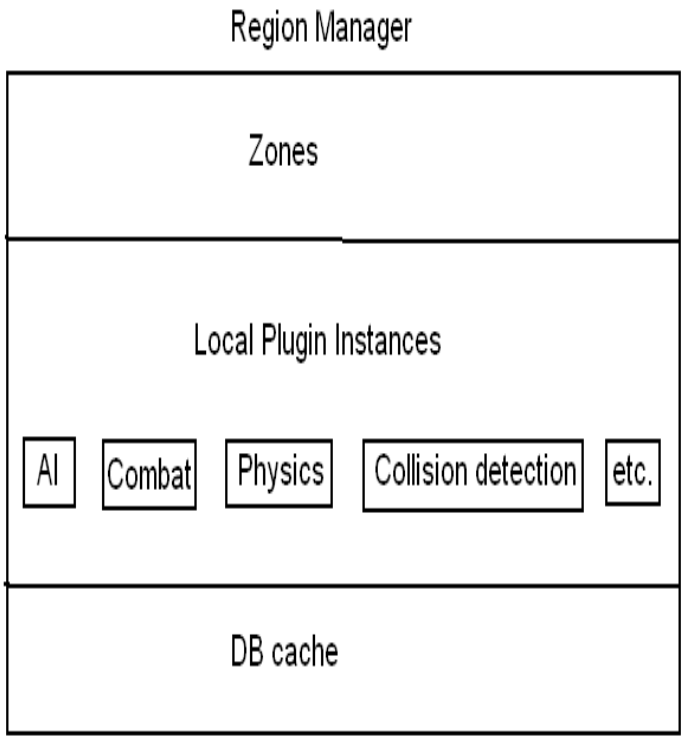

Fig. 4 region manager with local plugin instances

V. Vertical scaling using GPGPU

Previous section has described the dynamical regional decomposition and middleware components required for horizontal scalability

In this section we describe a brand new idea that builds on the decomposition idea to make use of the huge parallel computing power of the new generations of graphics processing units.

A. GPUs evolution

A graphics processing unit (GPU) is a specialized hardware module, used as component of a PC, graphics station or gaming console, specialized in performing high speed graphics calculations.

GPUs spectacular evolution can be briefly resumed:

- First GPUs only supported basic graphics functions; as hardware they were adapted general purpose co-processors or signal processors, use to take some of the graphics processing workload from the main computer
- PC market boom turned the GPU into a standard component, taking over most standard 2D and 3D graphics calculation from the CPU, being manufactured by many companies and rapidly evolving.
- Programmable shaders were introduced, being the first form of GPU programmability, with the aim of giving the graphics applications developer more control over the operations performed in the graphics pipeline
- General-Purpose Computation on GPUs (GPGPU)

came into picture and keeps getting more and more popularity.

GPGPU is basically about using the GPU to perform computation in many applications traditionally handled by the CPU – made possible by the addition of programmable stages to the rendering pipelines and various libraries and development toolkits

Struggling with the peculiarities of the GPUs programming, GPGPU developed a set of specific techniques to convert general parallel computations to the forms accepted by GPUs [7].

– In the near future the trend of turning dedicated GPUs architectures into more general parallel ones will certainly continue, together with some hybrid solutions, like the announced Intel's Larrabee

All these trends and influences will certainly change the traditional role of the GPUs and CPUs, as hardware and programming models and development toolkits.

Fig. 5 shows a comparative evolution of GPU (illustrated by NVIDIA graphics cards) and desktop CPU (illustrated by Intel processors) computational power.

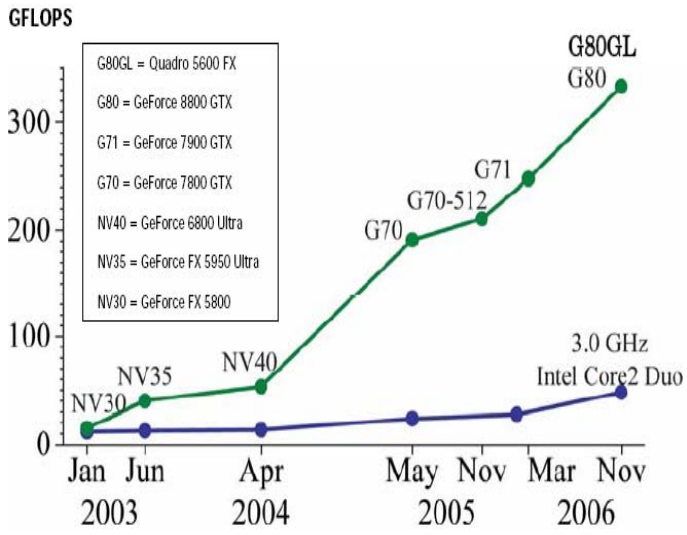

Fig. 5 comparative evolution of GPU and desktop CPU (from: Case studies on GPU usage and data structure design - Jens Breitbart)

The growth tendency much higher for GPUs is backed by the fact that they tend to increase computing power by increasing the number of processors, rather than the power of each individual processor, as CPUs do.

Other explanation for the huge difference is that GPUs tend to use most of their hardware for highly parallel computational operations, rather than control operations and caches – as CPUs, as shown symbolically in Fig. 6.

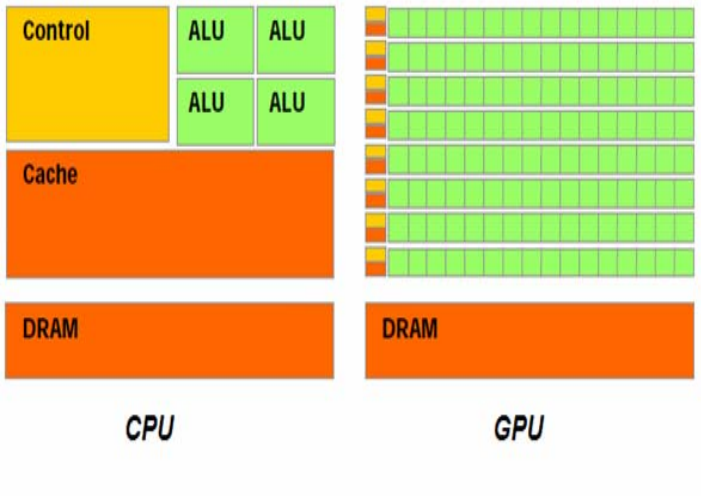

Fig. 6 symbolic usage of transistors in CPUs and GPUs

B. Current GPGPU concepts and limitations

1) Streams

The closest concept to the hardware and programming model of a modern GPU is that of a SIMD machine, or stream processing.

Streams are sets of records with identical format that require similar processing. In the case of GPGPU, the most natural format for a stream is that of a 2D grid, which fits with the rendering GPU model.

2) Kernels

The processing or functions that work on stream are usually named “kernels”.

For example, vertex or fragment shaders are particular cases of kernels.

In the particular case of GPUs, the kernels can be seen as the body of loops iterated over 2D matrixes.

Depending on actual GPGPU model, there is some flexibility in organizing the kernels in groups, groups having some things in common, for instance fast registries or some shared memory with faster access then the global memory used by all threads.

3) Flow control and limitations

The common flow control mechanisms that all programmers are used to (like if-then-else or loops) have been only recently added to GPU. Many limitations still exist, for example:

– runtime ramifications come with a big performance cost
– recursivity is not supported and can just be partially emulated
– data transfer from/to memory is also costly in terms of performance

We must highlight that a runtime branching followed by barrier synchronization for a group of threads running same kernel might come, depending on actual GPU model, at a cost of hundreds of normal operations. Same for data transfer to/from memory.

This makes the GPGPU only suitable for some kinds of operation that require few data transfers and have a high degree of parallelism.

However, GPU designers are taking into consideration the demands of the on growing GPGPU market and trying to improve the programming models supported by GPUs, either by more flexible end efficient architecture or by specialized GPGPU libraries and toolkits.

VI. Regional decomposition with GPGPU

The regional decomposition method is general enough to be used for both horizontal (multiple computers) and vertical (multi-processor computers) scalability.

We describe here our idea to map the method to the huge as computing power but limited as programming flexibility form of parallelism supported by most modern GPUs.

The challenge is to distribute the workload between CPU and GPU in an efficient way.

The following aspects must be considered:

- granularity of kernels must be small and their nature highly parallel (runtime ramifications should be minimalistic)
- coordination should be performed by the CPU, to integrate easily in the rest of the server architecture
- data transfers CPU-GPU must be minimized
- Workload distribution over streaming processors must be adaptable in real time

We have divided the tasks performed on a computer that relate to region management, as described in previous paragraphs, into two classes, considering the particularities of the GPU and CPU programming models and the specific of each task.

- GPU tasks
- CPU tasks

A. GPU tasks

GPU should only handle computational intensive tasks, organized in low granularity and highly parallel (static and at runtime also) kernels:

- collision detection
- advanced physics calculations
- basic decisional AI

The code executed by GPU will have two components:

- a component that creates the kernels, assign them to processor groups and launches them
- the kernels

B. CPU tasks

Will be responsible for the high level control:

- regions decomposition
- input/output
- distributed control, messaging, persistence
- synchronization with GPU

Of course, CPU will also handle all other regional level tasks that are not suitable for GPGPU.

C. Frames, Execution and Synchronization

The server workflow consists of frames. A frame is a very short time interval, which, as a design choice, can be fixed or variable.

During a frame the following things will happen:

- user input is taken from input queues, pre-processed and passed to the GPU code of control.
- GPU will interpret input and create kernels to handle it.
  For example, a collision detection kernel will be created for each pair of moving objects from the virtual space
- kernels are launched over streaming processors
- the streaming processors execute the kernels
- depending on frame type:
  - when all kernels are completed (in case of variable frame design)
    or
  - when the frame duration has elapsed (in case of design with fixed frame)

  the available results from kernels execution are placed in output queues

D. Load balancing

A computer can handle some number of regions. Each region is assigned a number of GPU streaming processors, according to its computational needs.

If this number can be changed dynamically, the load balancing comes naturally at no cost.

Most current GPUs do not offer possibilities to explicitly specify the number of processors assigned to a group of kernels, but most of them offer facilities that allow emulating this behavior (for example, in NVIDIA's CUDA, the kernels in a group share a region of faster shared memory).

It is also expected that incoming generations of GPU will include more facilities for controlling processors allocation to groups of threads, maybe even explicitly.

E. Tuning

Optimizing the GPU tasks require perfect knowledge of the actual GPU hardware and software particularities.

In most cases, data transfer is quite slow; hence GPU tasks and data structures must be designed in a way to minimize these.

Fig. 4 shows the costs (processor cycles) related to each type of operation on some NVIDIA GPU. We can make draw the following conclusions:

- major penalties for memory access
- high costs also for the operation of synchronizing groups of kernels.

It is desirable that GPU tasks have small memory transfer footprints and also minimal runtime ramifications, but this is not always possible.

F. Using the solution in frameworks for RAD

An actual implementation of the proposed solution will be heavily dependent upon hardware/software particularities of

GPU models.

If such solution is to be integrated in a framework for RAD of massive multiplayer 3D virtual spaces, it should be made as easier to use as possible.

This apparently difficult issue can be solved by rigorous design of the framework, by pre-integrating the solution for basic cases and also making it easily adaptable or extensible for special cases.

Basically, at least the following aspects need to be carefully considered:

- the solution should be pre-integrated and configured to handle most common highly computational tasks
- the developer should be able to explicitly specify the tasks to be handled by GPU
- both the GPU control code and the kernels should be programmable in a simple scripting language (e.g. Python), many developers being used to this approach
- the documentation should warn the developers about the GPU programming particularities and prevent them from writing inefficient code
- the system should include profiling facilities, allowing measurement of the execution time of tasks, so that the developer can tune his application

G. Prototype and results

The solution described above was tested by creating a server prototype, during the "Graphics and Virtual Reality Workshop 2008", which took place in the University "POLITEHNICA" from Bucharest.

The architectural concept was implemented using NVIDIA graphics cards as hardware and CUDA as development toolkit.

The workshop focused on GPGPU and its main project was to validate the architectural concept described in this paper.

It was created a prototype for a massive multiplayer 3D virtual space as server the decomposition method implemented with GPGPU, a client and a testing environment.

The server prototype only included basic elements for testing:

- basic TCP/IP multi-player communication
- CPU and GPU implementation of the regions decomposition and management concept
- GPGPU collision-detection
- support for profiling the CPU and GPU tasks execution time

There was also created a basic client to run the tests, and a testing environment.

The actual tests performed were about creating collisions between huge numbers of objects in the 3D virtual space and measuring the difference from performing collision detection with CPU implementation and with GPU implementation, as described in previous paragraphs.

The results were really encouraging, showing speeding up of more than one order of magnitude for some types of tasks when adapted to GPU.

We are determined to further explore the possibilities of the concept.

Instruction	Cost (clock cycles per warp)
FADD, FMUL, FMAD, IADD	4
bitwise operations, compare, min, max	4
reciprocal, reciprocal square root	16
accessing registers	0
accessing shared memory	≥ 4
reading from device memory	400 - 600
reading from constant memory	≥ 0 (cached) 400 - 600 (else)
synchronizing all threads within a block	4 + possible waiting time

Fig. 4 overview of instructions cost on the G80 architecture (from: Case studies on GPU usage and data structure design - Jens Breitbart)

VII. Conclusion

The decomposition idea and its mapping to GPGPU have a huge significance for the 3D massive multiplayer virtual spaces server world.

If the proposed architectural solution will be successfully implemented, we can see, in the near future the hundreds of computers server farms in use by successful MMORPGs being replaced with only a few PCs equipped with multiprocessors GPUs.

As the GPUs and CPUs programming models are evolving very fast, merging more or less, we can expect that all the limitations (that make an actual implementation of the of the solution describe in this paper to be quite hard) to become history, and variants of the solution to be easily implementable in the near future on next generations of GPUs/CPUs.

Full scalability, full spatial continuity for large 3D virtual spaces, rapid prototyping, cheap and flexible hosting and maintenance and many other benefits may emerge as result of these.

Such possibilities will contribute to the acceptance of 3D massive multiplayer virtual spaces as the next paradigm for human-computer-human interaction, used throughout almost all human activities.

References

[1] J. Waldo, "Scaling in Games and Virtual Worlds," *Communications of the ACM*, Vol 51, No 08, 2008, pp. 38–44.

[2] Ta Nguyen, Binh Duong, Suiping Zhou, "A dynamic load sharing algorithm for massively multiplayer online games," *The 11th IEEE International Conference on Computational Science and Engineering*, 2003, pp. 131–136.

[3] P. Morillo, J. Orduña, M. Fernández, "Workload Characterization in Multiplayer Online Games," *ICCSA* (1), 2006, pp. 490–499.

[4] S. Ferretti, "Interactivity Maintenance for Event Synchronization in Massive Multiplayer Online Games," *Technical Report UBLCS-2005-05*, March 2005.

[5] S. Bogojevic, M. Kazemzadeh, "The architecture of massive multiplayer online games," M.S. thesis, Lund Institute of Technology, Lund University, Lund, Sweden, 2003.
[6] Multiverse Platform Architecture [Online]. Available: http://update.multiverse.net/wiki/index.php
[7] John R Humphrey, Daniel K. Price, James P. Durbano, Eric J. Kelmelis, Richard D. Martin, "High Performance 2D and 3D FDTD Solvers on GPUs," *Proceedings of the 10th WSEAS International Confenrence on Applied Mathematics*, Dallas, Texas, USA, November 1-3, 2006, pp. 547–550.
[8] A.S.Drigas, L.G.Koukianakis, G. Glentzes, "A virtual Lab for Hellenic cultural Heritage," *Proceedings of the 5th WSEAS Int. Conf. on Artificial Intelligence, Knowledge Engineering and Data Bases*, Madrid, Spain, February 15-17, 2006, pp291–296.
[9] N. Ibrahim, M.A.M. Balbed, A.M. Yusof, F. Hani, M. Salleh, J. Singh, M.S. Shahidan "Virtual Reality Approach in Acrophobia Treatment," *7th WSEAS Int. Conf. on Applied Computer & Applied Computational Science (ACACOS '08)*, Hangzhou, China, April 6-8, 2008, pp. 194–197.
[10] N. Sala, "Multimedia and Virtual Reality in Architecture and in Engineering Education," Proceedings of the 2nd WSEAS/IASME International Conference on Educational Technologies, Bucharest, Romania, October 16-17, 2006, pp. 18–23.