Systemic Computation using Graphics Processors

Marjan Rouhipour[1], Peter J Bentley [2], Hooman Shayani[2]

[1] BIHE University (The Bahá'í Institute for Higher Education), Iran.
marjan.rouhipour@bihe.org
[2] Department of Computer Science, University College London, Malet Place, London.
{p.bentley, h.shayani}@cs.ucl.ac.uk

Abstract. Previous work created the systemic computer – a model of computation designed to exploit many natural properties observed in biological systems, including parallelism. The approach has been proven through two existing implementations and many biological models and visualizations. However to date the systemic computer implementations have all been sequential simulations that do not exploit the true potential of the model. In this paper the first parallel implementation of systemic computation is introduced. The GPU Systemic Computation Architecture is the first implementation that enables parallel systemic computation by exploiting multiple cores available in graphics processors. Comparisons with the serial implementation when running a genetic algorithm at different scales show that as the number of systems increases, the parallel architecture is several hundred times faster than the existing implementations, making it feasible to investigate systemic models of more complex biological systems.

Keywords: Bio-inspired computation; systemic computation; GPU; parallel architectures; genetic algorithm

1 Introduction

In biological modeling and bio-inspired computation, the demand for fast parallel computation has never been greater. In computer science, entire fields now exist that are based purely on the tenets of simulation and modelling of biological processes. In the developing fields of synthetic biology, DNA computing, and living technology, computer modelling plays a vital role in the design, testing and evaluation of almost every stage of the research[1]. There are also many fields of computer science that focuses on bio-inspired algorithms such as genetic algorithms, artificial immune systems, developmental algorithms, neural networks, and swarm intelligence.

Almost without exception these computer models and algorithms involve parallelism, although they are usually implemented as serial simulations of parallel processes. While multi-core processors, clusters or networked computers provide one

[1] Evident from the many publications of the European Center for Living Technology: http://www.ecltech.org/publications.html

way to parallelise the computation, the underlying computer architectures remain serial, and so can significantly limit our ability to scale up our models and bio-inspired algorithms and make them practical for real-world problems [1].

Several research groups have focused on this area for many years, resulting in novel bio-inspired architectures such as the POEtic tissue [2] and the Ubichip of Perplexus project [3], as well as formalisms such as PI-Calculus, Bi-Graphs [4] and Brane Computing [5]. *Systemic computation* is a similar attempt to exploit desirable natural properties such as parallelism within a computer, developed in 2005 [1].

Although not the first such model, SC is the result of considerable research into bio-inspired computation and biological modelling, and has been developed into a working computer architecture [1,6]. To date, two simulations of this architecture have been developed, with corresponding machine and programming languages, compilers and graphical visualiser [1,6]. Extensive work has shown how this form of computer enables useful biological modeling and bio-inspired algorithms to be implemented with ease [7-11] and how it enables properties such as fault-tolerance and self-repairing code [12]. Research is ongoing in the improvement of the PC-based simulator, refining the systemic computation language and visualiser [10,14]. However, the systemic computation model defines a highly parallel, distributed form of computer. While simulations on conventional computers enable improvement of the model and programming tools, the speed of simulated systemic computation is too slow to be useable for larger models. The work described in this article aims to overcome this problem by making use of Graphics Processing Units (GPUs) to parallelize some of the bottlenecks in systemic computation and thus take the first steps towards a fully parallel systemic computer, capable of high-speed modeling.

Graphic Processing Units are multi core processors designed to process graphical information at high speed. Because of their price and power, the use of GPUs for more general-purpose computation is rapidly becoming something of a revolution in affordable parallel computation [15]. More recently the design of GPUs was changed to support more general computation. Today GPGPU (General Purpose GPU) languages are used widely in scientific computation. They are used in physical based simulation, signal and image processing, global illumination, and geometric computing [15]. GPGPU is frequently used in biological modelling and visualization that requires large-scale computation and real-time processing. They have been used in molecular modeling applications [17], string matching to find similar protein and gene sequences [18], and in implementations of bio-inspired algorithms [19].

This work describes a novel GPU-based implementation of the bio-inspired computing approach known as systemic computation. The next section summarizes systemic computation. The new GPU Systemic Computation Architecture is then described, followed by an experiment to compare the GPU version with the single-processor implementation.

2. Systemic Computation

Systemic Computation is a model of computation and corresponding computer architecture based on a systemics world-view and supplemented by the incorporation of natural characteristics [1]. This approach stresses the importance of structure and interaction, supplementing traditional reductionist analysis with the recognition that circular causality, embodiment in environments and emergence of hierarchical organisations all play vital roles in natural systems. Systemic computation makes the following assertions:

- Everything is a system.
- Systems can be transformed but never destroyed or created from nothing.
- Systems may comprise or share other nested systems.
- Systems interact, and interaction between systems may cause transformation of those systems, where the nature of that transformation is determined by a contextual system.
- All systems can potentially act as context and affect the interactions of other systems, and all systems can potentially interact in some context.
- The transformation of systems is constrained by the scope of systems, and systems may have partial membership within the scope of a system.
- Computation is transformation.

In systemic computation, everything is a system, and computations arise from interactions between systems. Two systems can interact in the context of a third system. All systems can potentially act as contexts to determine the effect of interacting systems. Every system is divided into three parts: two schemata and one functional region. These three parts can be used to hold anything (data, typing, etc.) in binary. The functional region defines the transformation of two systems interacting in its context. The two schemata specify through a matching function which subject systems may interact in this context. A system can also contain or be contained by other systems. This enables the notion of scope. Interactions can only occur between systems within the same scope. An SC program therefore comprises systems that are instantiated and positioned within an embedded hierarchy (some inside each other). It defines an initial state from which the systems can then randomly interact, transforming each other through those interactions and following an emergent process rather than a deterministic algorithm. For full details see [1] and [10].

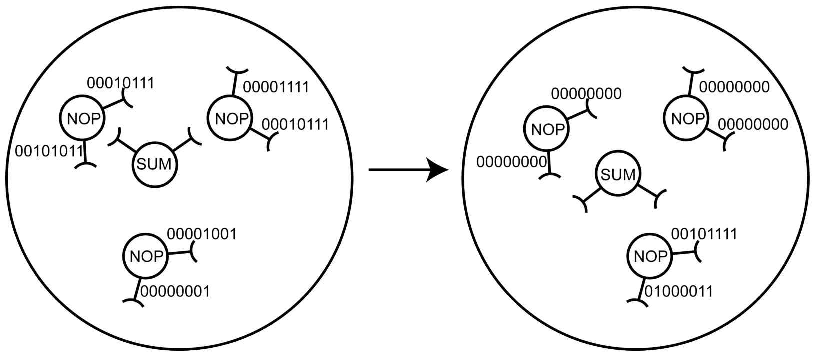

Figure 1. A basic SC program, which sums the values held in the schemata of systems, leaving one system containing the solution. Initial state is shown on the left, state after several interactions is shown on the right.

Systems can be implemented using representations similar to those used in genetic algorithms and cellular automata. In implementations to date, each system comprises three binary strings: two schemata that define sub-patterns of the two matching systems and one coded pointer to a transformation function. Two systems that match the schemata will be transformed according to the appropriate transformation function (i.e. their binary strings are modified according to the function defined in the context). A simple example of a (partially interpreted) system string (where $S1_1$ is the first schema and $S1_2$ is the second schema of system 1) might be:

"zzx00rzz [$S1_1$=SUM($S1_1$,$S2_1$); $S1_2$=SUM($S1_2$,$S2_2$); $S2_1$=0; $S2_2$=0] zzx00rzz"

meaning: for every two systems that have functional part of NOP that interact in the context of this system, add their two S_1 values, storing the result in S_1 of the first system and add the two S_2 values, storing the result in S_2 of the first system, then set S_1 and S_2 of the second system to zero. (The table of schema codes is given in [1].) Given a pool of inert data systems, able to interact but with no ability to act as context, for example (where NOP means "no operation"):

```
"00010111 NOP 01101011" and "00001111 NOP 00010111"
```

after a sufficient period of interaction, the result will be a single system with its S_1 and S_2 values equal to the sum of all S_1 and S_2 values of all data systems, with all other data systems having S_1 and S_2 values of zero. Figure 1 illustrates the program using SC graph notation. (The program performing this operation was described in [1].)

Bentley [1] describes the first implementation of the systemic computer. The initial work included the creation of a virtual architecture, instruction set, machine code and corresponding assembly language with compiler. Over 30 transformation functions were implemented (e.g., arithmetic and logical operations, and basic i/o). The simulation was implemented in ANSI C on a PowerBook Macintosh G4, enabling systemic computation programs to be simulated using conventional computer processors. Later work by Le Martelot created a second implementation on PCs with a higher-level language and visualization tools [2,6-12,14]. Other work provided a discussion on the use of sensor networks to implement a systemic computer [13]. Many systemic computation models have been written, showing that simulations of this parallel computer can perform tasks from investigations of neurogenesis to a self-adaptive genetic algorithm solving a travelling salesman problem [10]. Work on the language and refinements to systemic computation and its use for modelling are underway. However, perhaps the biggest single problem with all implementations to date has been the speed of execution. The simulation of a parallel bio-inspired computer on a conventional serial computer can be excessively slow for models using large numbers of systems. For this reason, we hypothesize that an implementation on GPUs may provide significant speedup.

3. GPU Implementation of Systemic Computation

Systemic computation is a Turing Complete parallel computer [1], but it is still a significant challenge to exploit the parallel architecture of the GPU to implement such a flexible bio-inspired approach.

The constituent elements of Systemic Computation are systems. Two systems can only interact with a context and within a shared scope, which are also systems. Akin to Bentley's implementation [1], membership of scopes can be implemented as a global scope table, a row and column for every system, with each entry defining the membership of one system within another system. Also akin to Bentley's original implementation [1], in the GPU design we can implement the concept of a system by storing it in three binary parts: two schemata and one function. If the current system is acting as a context, then its two schemata define all possible pairs of systems that could interact within the current context. The system function defines the interaction between the two systems, i.e. it provides a transformation function. In the implementation created by Bentley [1], the transformation function contains a matching threshold for schema 1 and schema 2. The length of each part of the system is 16 character codes. Each code is decoded to three binary characters (0, 1, and wildcard). If the difference between a system and the decoded schema's part is less than the schema's threshold, that system matches the context system's schema [1].

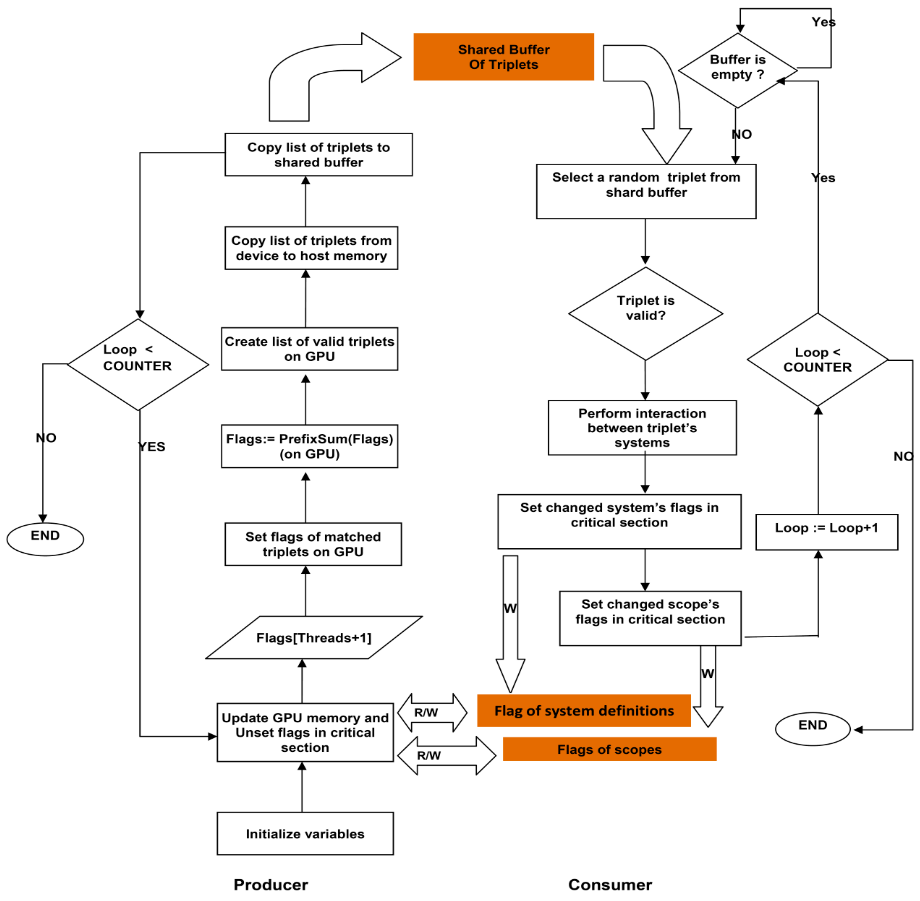
Shared Buffer Of Triplets
Buffer is empty ?
Yes
NO
Copy list of triplets to shared buffer
Select a random triplet from shard buffer
Yes
Copy list of triplets from device to host memory
Triplet is valid?
Create list of valid triplets on GPU
Loop < COUNTER
Loop < COUNTER
Flags:= PrefixSum(Flags) (on GPU)
Perform interaction between triplet's systems
NO
YES
NO
Set changed system's flags in critical section
Loop := Loop+1
END
Set flags of matched triplets on GPU
W
Set changed scope's flags in critical section
Flags[Threads+1]
W
Update GPU memory and Unset flags in critical section
R/W
Flag of system definitions
R/W
Flags of scopes
END
Initialize variables
Producer
Consumer

Figure 2. Producer and Consumer flowchart.

We call a system that has a valid transformation function, a 'context' or 'functional' system. A context system and two interacting systems together are called a triplet. If two interacting systems can match the schema parts of a functional system to form a triplet, and all of them are in the same scope, it is defined as a matched or valid triplet.

The GPU Systemic Computation architecture has two main tasks: 1) finding a valid triplet, and 2) performing a transformation to the interacting systems. The finding of valid triplets is the biggest bottleneck in systemic computation, so in our design, the *producer* finds matched triplets and puts them in a shared buffer whilst the *consumer* picks one of triplets and performs the interaction between them. The producer and consumer are two threads, running in parallel on the CPU, see Figure 2.

3.1 Consumer: Performing System Interactions

The consumer is a thread running on the CPU. It is responsible for enabling the interactions between systems. This thread selects valid triplets (each a valid context and two matching systems) randomly from the shared buffer. It then uses the transformation function defined in the context system to transform the pair of interacting systems. However, performing the transformation may change the systems' scopes and definitions. As a result, other triplets in shared memory that share the one of the current triplet's systems may no longer match (i.e., the transformation of one triplet may invalidate other triplets). In order to solve this problem we check the validity of triplets before performing interaction. After selecting a triplet, if the triplet is still a matched triplet, the transformation is performed. Then the flags of systems definitions and scopes are set. These flags are necessary to update data on GPU memory for the other thread, *producer*.

3.2 Producer: Finding Matching Systems

General Purpose Graphic Processing Unit (GPGPU) languages are based on shared memory [20][21]. In the GPU Systemic Computation Architecture, systems are shared data and the instructions that check validity of triplets in a scope are the same for all threads. So, CUDA is a good choice to find matched triplets in parallel.

Finding a list of matched triplets is both a sequential and parallel procedure. There are six main steps: initializing, updating, finding matched triplet, prefix sum, creating a list of matched triplets, and copying them to the shared buffer. These steps are run sequentially on the CPU. The third, fourth, and fifth steps are the main parts that are run on GPU. Each of these steps is a kernel, a section of code that is run on the GPU. As only one kernel can be run on the GPU in CUDA, and the output of each step is an input of the next step, these steps run sequentially on the CPU, but individually parallel on the GPU. In the summary below we focus on kernels.

STEP 1: Initialize. Memory is allocated on the GPU for different variables: system definitions, scope table and decoded systems (including the two decoded schemata and threshold function).

STEP 2: Update. The other thread that performs transformation functions changes the scope table and system definitions, and so the producer thread always updates variables on GPU before checking all possible combination of triplets. Then new values of variables are copied from the host, (a part of the hardware that is on the CPU's processing part) to the device, (part of hardware that is on the GPU's part for processing). In addition, functional systems and valid scopes (scopes with equal or greater than three systems inside and at least one functional system) are found and copied to the GPU Memory. For finding differences between the schema part and the system, the schema is decoded; therefore, a list of decoded functional systems is prepared on the device and updated after being changed by the consumer thread.

STEP 3: Finding Matched Triplets. In this step a kernel is called that searches through all possible triplets in order to find matched triplets. Before calling the kernel, memory is allocated for the list of flags. Each thread has a flag that is initialized to zero. Next, the GPU grid and block dimensions are set.

All threads in a block check one scope and context for some interacting systems. Thus, one thread in each block, usually the first one, calculates the index of context and scope systems and stores it in the shared memory of the block; meanwhile, other threads in a block wait. If the context is in the scope, the index of interacting systems is then calculated. After that if interacting systems are in the scope and three systems are matched, the triplet's flag is set to one. (Memory access is optimized by making use of the short-latency, on-chip memory as a cache.)

STEP 4: Prefix Sum. In this step we want to create a list of parallel matched triplets, whose flags have been set in the previous step. In order to do so, the index of matched triplets in the new list have to be found. The prefix sum kernel is run on the flags to find indexes of matched triplets. The prefix sum calculates number of previously matched triplets in the current list for each matched triplet. A parallel implementation of this algorithm is available in CUDA SDK 2.2[2]. The current implementation of this algorithm is only run in one grid dimension, but we have changed it to two dimensions to support a larger size array, if sufficient memory is available.

STEP 5: Creating List of Triplets. The kernel for creating lists of matched triplets is run with the same grid's and block's dimensions of the found matched triplets kernel. The first thread calculates the block and context, stores in the shared memory. Then threads read two subsequent memories of prefix-sum flags. If a thread identified two different values, the thread's dimensions indicate indexes of matched triplet's systems and scope. Here each flag memory is read twice, for optimization the first 16 threads of each block load the 17 subsequent memory of flags to shared memory. This helps to reduce the number of reads from global memory by half.

STEP 6: Copy matched triplet list to buffer. A list of matched triplets is copied to the host. It is then randomized and copied to the shared buffer. For a large number of

[2] Available online at: http://www.nvidia.com/object/cuda_get.html

systems, it is not possible to process all combination of triplets at once, because there is not enough memory for flags and list of matched triplets on the GPU. So each time step, a subset of all possible triplets is chosen randomly and processed, a list of matched triplets of the subset triplets is created. After all possible triplets checks, variables update the GPU memory and the second round starts.

4. Architecture Testing and Evaluation

In order to assess the new implementation we run a systemic program on the original architecture created by Bentley [1] and the new GPU Systemic Computation Architecture, and compare the results. In the following section we outline a problem implemented in the Systemic Computation language: the knapsack problem using a Genetic Algorithm (GA). The problem is specifically designed to challenge the systemic computer with a complex parallel computation.

4.1 Genetic Algorithm Optimization of Binary Knapsack

In the knapsack problem there are *n* objects with value $v_i > 0$ and weight $w_i > 0$. We want to find a set of objects with maximum total weight that fits into a knapsack with capacity C. Thus, we wish to maximize:

$$\sum_{i=1}^{n} v_i x_i \text{ where } \sum_{i=1}^{n} w_i x_i \leq C \text{ and } x_i \in \{0,1\}$$

Here, we use a Genetic Algorithm (GA) [10] as a bio-inspired algorithm to implement the optimization program. The binary knapsack implementation in the Systemic Computation language is derived from the genetic algorithm model developed for systemic computation in [2]. There are three different solutions, as shown in Figure 3: uninitialized solutions, initialized solutions, and final solutions. The chromosome size equals the schema size (16 in this implementation). So, this program supports a knapsack with 16 objects.

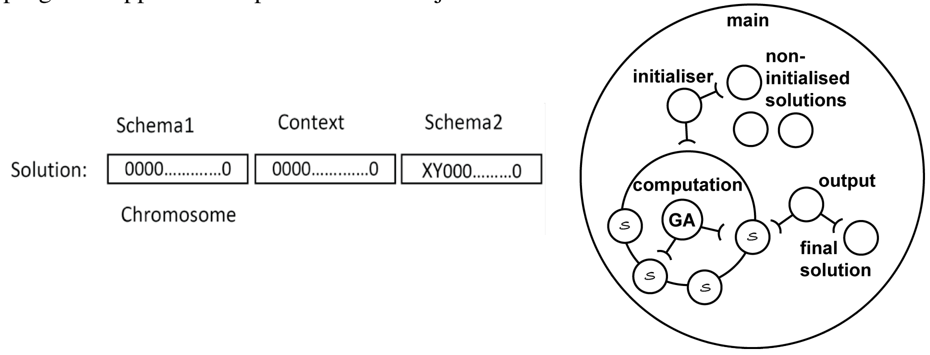

Figure 3. Left: The **Solution** system *S*. Schema1 stores the chromosome. **XY** in Schema2 specifies solution type (00: non initialized, 10: initialized, 11: final solution). Right: the systemic program (not all non-initalised, inialised solutions and GA systems shown).

Solution initialisation is random. An initialiser system selects one solution and initializes it randomly then places it in the scope of the computation system. Three GA operators are used: uniform crossover, one-point crossover and binary mutation [10]. Candidate (evolving) solutions to the problem are implemented as systems, and GA operators are implemented as context systems, defining the interaction of pairs of solution systems. Each operator performs crossover or mutation to generate a new solution, this is evaluated and then the less fit solution is replaced by the new one. The final solution system is used to store the final chromosome. The output context accepts the final solution and one initialized solution, and updates the final solution if the input solution system has better fitness than final solution system.

4.3 Experiments

To assess any performance gain of the GPU Systemic Computation Architecture, comparisons are made with Bentley's original serial implementation [1]. We study the effect of number of systems. The execution time in both experiments includes the running time for 10,000 interactions. It does not include reading of input and program files and initializing variables on CPU or storing results, but it does contain initializing and updating GPU memory, allocating and releasing memory from it. The hardware and software specification used in this work is given in Table 1. Bentley's original Systemic Computation Architecture was written with C language [1], thus we use C language based on CUDA as programming language.

Setup
The goal of the experiment is to compare the performance of the new parallel implementation and Bentley's original sequential implementation of the systemic computation architecture with increasing number of Solution systems on the GA binary knapsack program. In this experiment, increasing number of systems refers to increasing number of GA solution systems. Each systemic computation program was run for 10 times; reported execution time is the average of all programs' execution time. In this experiment, the number of knapsack objects is 16; the maximum knapsack's weight is 80.0 kg. The configuration of experiment is:

- Context systems: 3 GA systems and 1 output system
- Solution systems: 50 to 4000 systems for sequential implementation and 50 to 8000 systems for parallel implementation (each increment is double the previous increment except 800 to 1000 with an increment of 200)
- Final Solution system: 1 system
- Scope: 1 main scope and 1 computation scope
- Initial increment: 50

Table 1. Hardware and operating system specifications are used for experiments

CPU	Intel® dual core™, 2.40 GHz	
RAM	2 GB	
OS	Microsoft Windows XP professional 2002 SP1	
GPU	Name:	GeForce 9800 GT
	CUDA:	1.1
	Size of Global memory:	1 GB
	Multiprocessors	14
	Number of cores:	112
	Clock Rate:	1.62 GHz

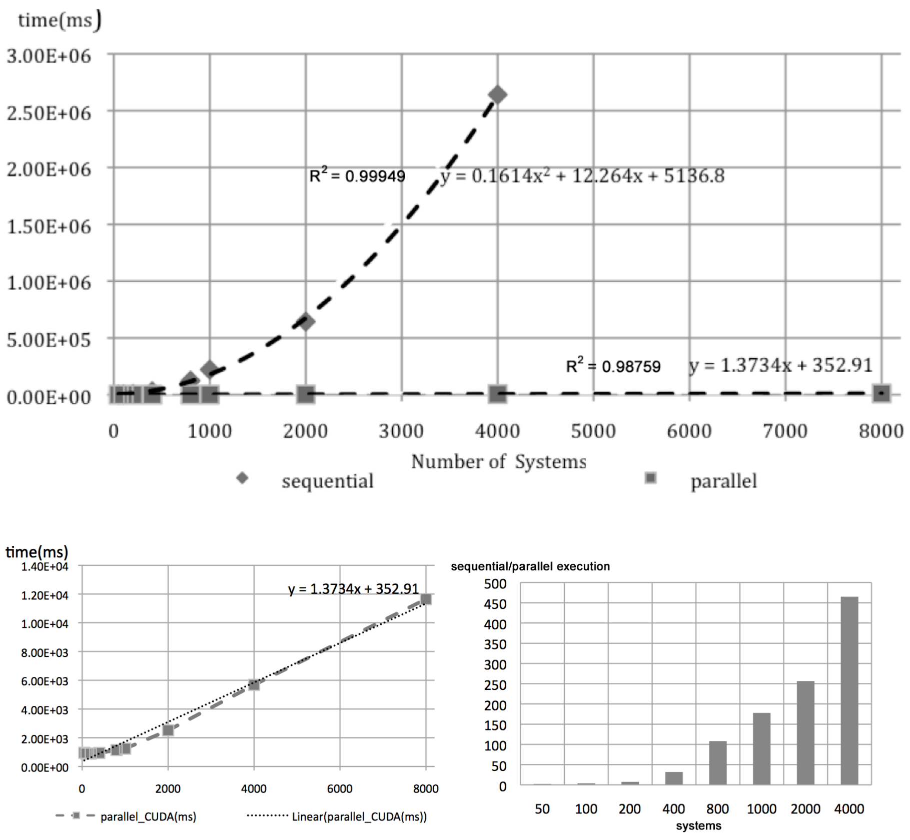

Figure 4. Top: Execution time of knapsack problem on both sequential and parallel implementation with increasing number of systems. Bottom left: execution time of parallel implementation alone. Bottom right: improvement as shown by sequential divided by parallel execution times for different numbers of systems in the program.

Results

Figure 4 shows the execution time for both parallel and sequential implementation with increasing number of systems. As can be seen in the figure, the execution time of the sequential implementation increases with a 2nd degree polynomial, characteristic of algorithms and implementations with time complexity $O(n^2)$. The parallel implementation appears to increase linearly, characteristic of algorithms and implementations with time complexity $O(n)$. The increase in performance varies according to the number of systems, e.g. it is 2.3 times as fast for 50 systems, 108 times faster for 800 systems, 256 times faster for 2000 systems, and 465 times faster for 4000 systems. The improvement derives from the efficient division of labour using the parallel resources of the GPU; it is likely that as the number of systems increases beyond this capacity, the execution time will appear as $O(n^2)$. Consistent results have since been found with other programs, including those that move systems between scopes.

5. Conclusion

The need for fast bio-inspired computation has never been greater. Systemic computation is a new bio-inspired model of computation that has shown considerable success for biological modeling and bio-inspired computation [6-14]. However until now it has only been available as a serial simulation running on conventional processors. In this work the first parallel GPU Systemic Computation Architecture was presented. Its performance was assessed by comparing the change in execution time needed when scaling up the number of systems within a genetic algorithm knapsack problem. As the number of systems increased, the parallel GPU architecture was several hundred times faster than the existing implementations.

These highly successful results will make it possible to investigate systemic models of more complex biological systems in the future. Further improvements are also planned. The GPU SC architecture is just the first step towards creating a fully parallel systemic computer. Future work will continue the development of parallel SC architectures and will investigate the use of reconfigurable hardware such as FPGAs.

References

[1] P. Bentley, "Systemic computation: A model of interacting systems with natural characteristics," J of parallel, emergent and distributed systems, vol. 22, pp. 103-121, 2007.

[2] W. Barker, D. M. Halliday, Y. Thoma, E. Sanchez, G. Tempesti, and A. M. Tyrrell. Fault tolerance using dynamic reconguration on the poetic tissue. IEEE Transactions on Evolutionary Computation, 11(5):666-684, 2007. doi: 10.1109/TEVC.2007.896690.

[3] A. Upegui, Y. Thoma, E. Sanchez, A. Perez-Uribe, J.-M. Moreno, and J. Madrenas. The perplexus bio-inspired recongurable circuit. In Proceedings of Adaptive Hardware and Systems (AHS 2007) Second NASA/ESA Conference, pages 600-605, 2007.

[4] R. Milner. Pure bigraphs: structure and dynamics. Inf. Comput., 204(1):60-122, 2006. ISSN 0890-5401.

[5] G. Paun. P systems with active membranes: Attacking np complete problems. Journal of Automata, Languages and Combinatorics, 6:75-90, 1999.

[6] Le Martelot, E., Bentley, P. J., and Lotto, R. B. (2007) A Systemic Computation Platform for the Modelling and Analysis of Processes with Natural Characteristics. In Proceedings of the 9th Genetic and Evolutionary Computation Conference (GECCO 2007) Workshop: Evolution of Natural and Artificial Systems - Metaphors and Analogies in Single and Multi-Objective Problems, pp. 2809-2816, July 7-11, 2007.

[7] Bentley, P. J. (2009) Methods for Improving Simulations of Biological Systems: Systemic Computation and Fractal Proteins. In J R Soc Interface 2009 6:S451-S466;

[8] Le Martelot, E., Bentley, P. J., and Lotto, R. B. (2007) Exploiting Natural Asynchrony and Local Knowledge within Systemic Computation to Enable Generic Neural Structures. In Proceedings of the 2nd International Workshop on Natural Computing (IWNC 2007), pp. 122-133, December 10-13, 2007, Nagoya University, Nagoya, Japan.

[9] Le Martelot, E. and Bentley, P. J. (2008) Metabolic Systemic Computing: Exploiting Innate Immunity within an Artificial Organism for On-line Self-Organisation and Anomaly Detection. In J of Mathematical Modelling and Algorithms (JMMA) vol. 8(2), pp. 203-225.

[10] Le Martelot, E. and Bentley, P. J. (2008) Modelling Biological Processes Naturally using Systemic Computation: Genetic Algorithms, Neural Networks, and Artificial Immune Systems. Book Chapter in Choing, R. (Ed) Nature-Inspired Informatics for Intelligent Applications and Knowledge Discovery, pp 204-241, IGI Global.

[11] Le Martelot, E., Bentley, P. J., and Lotto, R. B. (2008) Eating Data is Good for Your Immune System: An Artificial Metabolism for Data Clustering using Systemic Computation. In Proceedings of the 7th International Conference on Artificial Immune Systems (ICARIS 2008), pp. 412-423, August 10-13, 2008, Phuket, Thailand.

[12] Le Martelot, E., Bentley, P. J., and Lotto, R. B. (2008) Crash-Proof Systemic Computing: A Demonstration of Native Fault-Tolerance and Self-Maintenance. In Proceedings of the 4th IASTED International Conference on Advances in Computer Science and Technology (ACST 2008), pp. 49-55, April 2-4, 2008, Langkawi, Malaysia.

[13] Bentley, P. J. (2007) Designing Biological Computers: Systemic Computation and Sensor Networks. Submitted chapter for Bio-Inspired Computing and Communication, notes arising from BIOWIRE 2007: A workshop on Bioinspired design of networks, in particular wireless networks and self-organizing properties of biological networks.

[14] Le Martelot, E. and Bentley, P. J. (2009) On-Line Systemic Computation Visualisation of Dynamic Complex Systems. In Proceedings of the 2009 International Conference on Modeling, Simulation and Visualization Methods (MSV'09), pp. 10-16, July 13-16, 2009.

[15] J. Owens, et al., "A Survey of General-Purpose Computation on Graphics Hardware " Computer Graphics Forum, vol. 26, pp. 80--113, 2007

[16] I. Buck, et al., "Brook for GPUs: stream computing on graphics hardware," ACM Transactions on Graphics, vol. 23, pp. 777-786, 2004.

[17] C. Rodrigues, et al., "GPU acceleration of cutoff pair potentials for molecular modeling applications," in Conference On Computing Frontiers, 2008, pp. 273-282.

[18] M. Schatz and C. Trapnell, "Fast exact string matching on the gpu," Center for Bioinformatics and Computational Biology, 2007.

[19] T. Clayton, L. Patel, G. Leng, A. Murray, I. Lindsay, "Rapid evaluation and evolution of neural models using graphics card hardware," in Genetic and Evolutionary Computation Conference, 2008, pp. 299-306.

[20] Programming Guide, NVIDIA Corporation, 2009, version 2.2. NVIDIA Corporation 2701 San Tomas Expressway Santa Clara, CA 95050. Available online at: http://developer.download.nvidia.com/compute/cuda/2_2/toolkit/docs/NVIDIA_CUDA_Programming_Guide_2.2.pdf

[21] NVIDIA GeForce 8800 GPU Architecture Technical Brief TB-02787-001_v01, 2006.