12th AIAA/ISSMO Multidisciplinary Analysis and Optimization Conference
10 - 12 September 2008, Victoria, British Columbia Canada

AIAA 2008-6021

Implementation of Digital Pheromones in PSO Accelerated by Commodity Graphics Hardware

Vijay Kalivarapu* and Eliot Winer†
Virtual Reality Applications Center, Iowa State University, Ames, IA, 50011, USA

In this paper, a model for Graphics Processing Unit (GPU) implementation of Particle Swarm Optimization (PSO) using digital pheromones to coordinate swarms within n-dimensional design spaces is presented. Previous work by the authors demonstrated the capability of digital pheromones within PSO for searching n-dimensional design spaces with improved accuracy, efficiency and reliability in both serial and parallel computing environments using traditional CPUs. Modern GPUs have proven to outperform the number of floating point operations when compared to CPUs through inherent data parallel architecture and higher bandwidth capabilities. The advent of programmable graphics hardware in the recent times further provided a suitable platform for scientific computing particularly in the field of design optimization. However, the data parallel architecture of GPUs requires a specialized formulation for leveraging its computational capabilities. When the objective function computations are appropriately formulated for GPUs, it is theorized that the solution efficiency (speed) can be significantly increased while maintaining solution accuracy. The development of this method together with a number of multi-modal unconstrained test problems are tested and presented in this paper.

I. Introduction

Particle Swarm Optimization (PSO) [1,2] is a population based heuristic method retaining many characteristics of evolutionary search algorithms such as GA and SA. It is a recent addition to global search methods [3] and one of its key features is its simplicity in implementation due to a small number of parameters to adjust [4, 5]. In a regular PSO, an initial randomly generated population swarm (a collection of particles) propagates towards the global optimum over a series of iterations. Each particle in the swarm explores the design space based on the information provided by two members – the best position of a swarm member in its history trail (*pBest*), and the best position attained by all particles (*gBest*) until that iteration. This information is used to generate a velocity vector indicating a search direction towards a promising design point, and the location of each swarm member is updated.

The drawback of this approach is that information from these two members alone is not sufficient for the swarm to propagate toward the global optimum efficiently. This either could cause the swarm to lock into a local minimum or take a long time to approach the global optimum. Previous work by the authors demonstrated promising performance improvement of PSO in terms of increased solution efficiency, accuracy, and reliability through implementing digital pheromones in PSO [6, 7] in both single and parallel computing environments using a traditional CPU. A quantitative assessment has also been made through statistical hypothesis testing [8].

Commodity GPUs were fixed functional and traditionally used for visualization purposes. However, the advent of programmable graphics hardware has unleashed a promising potential for scientific computing. Researchers and developers have begun to harness GPUs for general purpose computation under a collective effort known as the GPGPU (General-Purpose Computation using Graphics Hardware) [9]. A tremendous amount of success has already been achieved in areas such as: a) computational geometry [10-13], b) geographic information systems [14], c) medical

* Postdoctoral Research Associate and author of correspondence, Department of Mechanical Engineering, Human Computer Interaction, Virtual Reality Applications Center, 2274 Howe Hall, Iowa State University, Ames, IA, 50011, USA, Student Member. Email: vkk2@iastate.edu, vijaykiran@gmail.com, Phone: 515-294-5318, Fax: 515-294-5530.
† Assistant Professor, Department of Mechanical Engineering, Human Computer Interaction, Virtual Reality Applications Center, 2274 Howe Hall, Iowa State University, Ames, IA, 50011, USA, Member.

and bio-medical applications [15], and e) solving dense linear systems [16]. For their low cost and ubiquitous availability, GPUs have a superior processing architecture when compared to modern CPUs, and thus presents a tremendous opportunity for developing optimization algorithms appropriate for GPUs.

GPUs are data parallel in nature, meaning that they can be utilized best when a single operation can be performed on multiple data. Additionally, computations on GPUs are most efficient when access to system memory is minimal thereby reducing bandwidth latencies. These requirements entail an algorithm to be appropriately formulated for GPU operations. One such model for implementing digital pheromones in PSO has been developed on commodity GPUs and presented in this paper. The developed method has been tested on various multi-modal n-dimensional test problems and compared against results from single and parallel CPU implementation of the method.

II. Background

A. Particle Swarm Optimization

PSO shares many characteristics of evolutionary search algorithms such as Genetic Algorithms (GA) and Simulated Annealing (SA) – a) Initialization with a population of random solutions, b) Design space search for optimum through updating generations and c) Update based on previous generations [17]. The success of the algorithm has brought substantial attention among the research community in the recent past [18, 19]. The working of the algorithm is based on a simplified social model similar to the swarming behavior exhibited by insects and birds. In this analogy, a swarm member uses its own memory and the behavior of the rest of the swarm to determine the suitable location of food (global optimum). The algorithm iteratively updates the direction of the swarm movement toward the global optimum. The mathematical formulation of the method is given in Equations (1) and (2).

$$V_{i+1} = w_i * V_i + c_1 * rand_p() * (pBest_i[] - X_i[]) + c_2 * rand_g() * (gBest[] - X_i[]) \quad (1)$$

$$X_{i+1} = X_i + V_{i+1} \quad (2)$$

$$w_{i+1} = w_i * \lambda_w \quad (3)$$

'pBest' represents the best position attained by a swarm member in its history trail, and *'gBest'* represents the best position attained by the swarm in the entire iteration history. Equation (1), represents the velocity vector update of a traditional PSO method where $rand_p()$ and $rand_g()$ are random numbers generated between 0 and 1 each for *pBest* and *gBest*. c_1 and c_2 are confidence parameters. w_i is called as the inertia weight [20, 21] and decreases in every iteration by a factor of λ_w, as represented in Equation (3). Equation (2) denotes the updated swarm location in the design space.

In addition to the originally developed PSO algorithm, significant enhancements have been proposed such as: a) mutation factors for better design space exploration [22, 23], b) methods for constraint handling [24, 25], c) parallel implementation [26, 27], d) methods for solving multi-objective optimization problems [28], e) methods for solving mixed discrete, integer and continuous variables [29].

B. Digital Pheromones

Pheromones are chemical scents produced by insects to communicate with each other to find a suitable food source, nesting location, etc. The stronger the pheromone, the more the insects are attracted to the path. A digital pheromone is analogous to an insect generated pheromone in that they are the markers to determine whether or not an area is promising for further investigation. One of the well-known applications of digital pheromones is its use in the automatic adaptive swarm management of Unmanned Aerial Vehicles (UAVs) [30, 31]. In this research, the UAVs are automatically guided towards a specific zone or target through releasing digital pheromones in a virtual environment, thereby reducing the requirement of humans physically controlling from ground stations. Other applications of digital pheromones include ant colony optimization for solving minimum cost paths in graphs [32, 33] solving network communication problems [34]. The concept of digital pheromones is considerably new [35] and has not yet been explored to its full potential for investigating n-dimensional design spaces for locating an optimum.

In a regular PSO algorithm, the swarm movement obtains design space information from only two components – *pBest* and *gBest*. When coupled with an additional pheromone component, the swarm is essentially presented with more information for design space exploration and has a potential to reach the global optimum faster.

C. Overview of Digital Pheromones in PSO

In a basic PSO algorithm, the swarm movement is governed by the velocity vector computed in Eq (1). Each swarm member uses information from its previous best and the best member in the entire swarm at any iteration. However, multiple pheromones released by the swarm members could provide more information on promising locations within the design space when the information obtained from pBest and gBest are insufficient or inefficient.

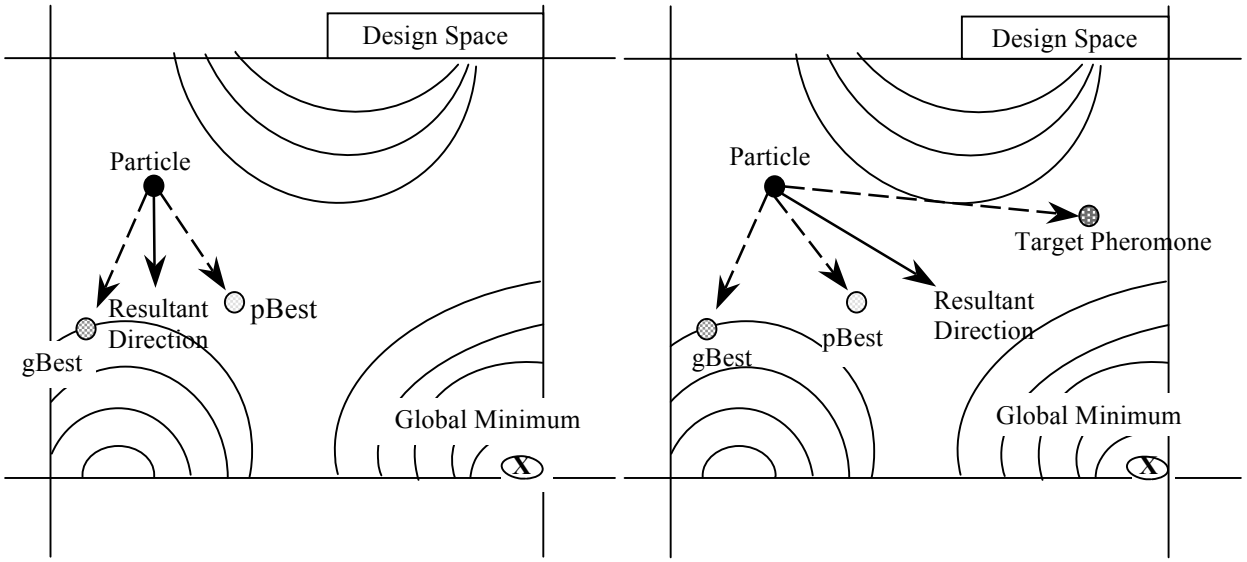


Figure 1 (a) Particle movement in a basic PSO, **(b)** Particle movement with digital pheromones

Figure 1a displays a scenario of a swarm member's movement whose direction is guided by pBest and gBest alone. If $c_1 >> c_2$, the particle is attracted primarily towards its personal best position. On the other hand, if $c_2 >> c_1$, the particle is strongly attracted to the gBest position. In the scenario dominated by c_2, as presented in figure 1a, neither pBest nor gBest leads the swarm member to the global optimum, at the very least, not in this iteration adding additional computation to find the optimum. Figure 1b shows the effect of implementing digital pheromones into the velocity vector. An additional target pheromone component potentially causes the swarm member to result in a direction different from the combined influence of pBest and gBest thereby increasing the probability of finding the global optimum.

Figure 2 summarizes the general procedure for PSO, with steps involving digital pheromones highlighted. The method initialization is similar to a basic PSO except that 50% percent of the swarm within the design space is randomly selected to release pheromones in the first iteration. This parameter is user-defined, but experimentation has shown 50% to be a good default value. For subsequent iterations, each swarm member that realizes any improvement in the actual objective function value is allowed to release a pheromone. Pheromones from the current as well as the past iterations that are close to each other in terms of the design variable value are merged into a new pheromone location. Therefore, a pheromone pattern across the design space is created, while keeping the number of pheromones manageable. In addition, the digital pheromones are decayed every iteration just as natural pheromones. Based on the current pheromone level and its position relative to a particle, a ranking process is used to select a target pheromone for each particle in the swarm. This target position towards which a particle will be attracted is called the target pheromone and added as an additional velocity vector component to pBest and gBest. This procedure is continued until a prescribed convergence criterion is satisfied. A detailed account of this procedure is fully explained in the previous work by authors [36], and is not described in this paper to maintain conciseness. The new velocity vector update equation is shown in eq. (4).

$$V_{i+1} = w_i \times V_i + c_1 \times rand_p() \times (pBest_i[] - X_i[]) + c_2 \times rand_g() \times (gBest[] - X_i[]) + c_3 \times rand_T \times (T\arg etPheromone_i[] - X_i[]) \quad (4)$$

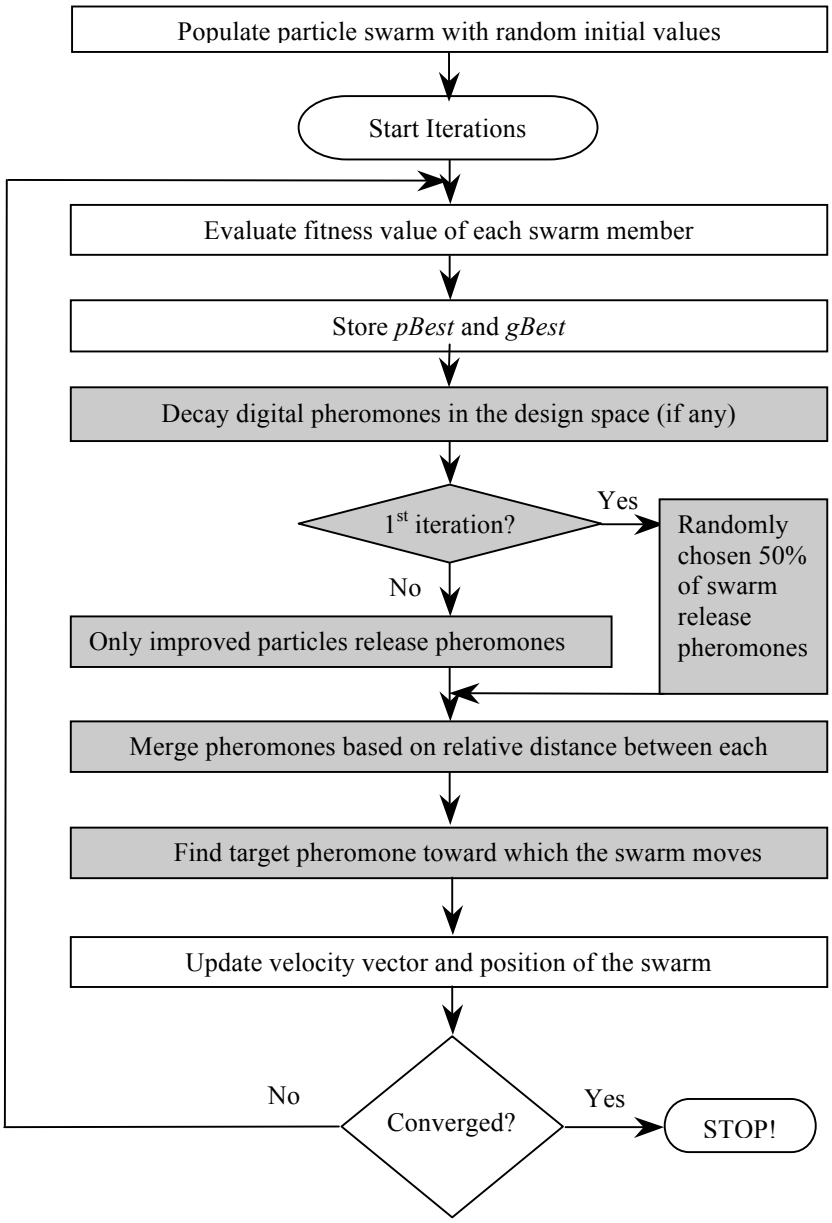


Figure 2 Overview of PSO with Digital Pheromones

D. Feasibility of GPUs

Recently, technologies such as hyper threading and multi-core processing [37] have been the main drivers increasing CPU performance as opposed to the addition of more transistors onto a CPU chip. While hyper threading requires an additional burden on the programmer to develop thread-enabled code to realize performance improvements, multi-core processor improvement is only linearly related to the number of cores used on the processor chip. For example, a dual core processor can only increase the CPU performance by approximately a factor of two. However, commodity Graphics Processing Units (GPUs) or more commonly graphics cards, another

proven and developing technology, is capable of improving computational performance more than ten times that of a modern CPU [38]. For their price and ubiquitous availability, GPUs have a superior processing architecture when compared to modern CPUs. For example, a dual core processor has essentially two CPUs on one chip, but depending upon the type, GPUs can have greater than 24 processors (24 fragment shading pipelines). In addition, GPUs are capable of supporting hundreds of hardware threads as opposed to one or two on a CPU. Early GPUs had fixed functionality that made them ideal for supporting visualization and gaming. Modern GPUs include improved programmable processing units and support vectorized floating point operations. The advent of programmable graphics hardware in recent years has unlocked the use of GPUs for purposes other than visualization to enable CPU type operations to be performed. GPUs offer distinct advantages to any process involving large amounts of computation as they are now: 1) programmable, 2) priced significantly less than a high performance CPU, 3) data parallel in architecture, 4) highly threaded, and 5) good at reducing main memory access costs.

The programming component of GPUs primarily consists of vertex shaders and fragment shaders (also called pixel shaders). In graphics programming, vertex shaders handle transformation of vertices of an object and fragment shaders handle computing the pixel color values that fill the screen. Initially, graphics programmers created low-level (fine control) vertex and fragment shaders to achieve these tasks. However, due to the tediousness involved in programming with these shaders and limited flexibility in terms of debugging and code re-use, low-level shader programming is not a preferred method for graphics programming. High-level shading languages, which incorporate several low-level function calls into easier to use functions, are now available, which solve the rigid low-level programming issues. The function of a shading language is to compile a shader program into individual vertex and/or fragment components and perform required computations before rendering images on the screen. Even though these operations were designed to create realistic computer graphics, they are still mathematical. If it is understood what mathematics are being performed, the data placed in a texture can be multiplied, divided, or subjected to other complex mathematical operations.

While CPU programming has a large number of well-established programming languages to choose from, there are only few GPU programming languages such as Cg [39], GLSL [40], HLSL [41], Sh [42], and Ashli [43]. These languages are quite graphics specific, so the terminology used in programming follow the mapping constructs to CPU programming given in Table 1.

Table 1 Terminology used for mapping CPU algorithms to the GPU

CPU	GPU
Arrays or streams	Textures
Parallel loops	Quads
Loop body	Vertex + fragment program
Output arrays	Render targets
Memory read	Texture fetch (gather)
Memory write	Framebuffer write (scatter)

These shader languages adopt a C/C++ style of programming syntax. While Cg abstracts the graphics hardware quite closely, GLSL has some data types defined outside of the scope of current day graphics cards such as integers and matrices. As graphics hardware begins to support these data types, GLSL will be a powerful language. Sh on the other hand provides stream-programming capabilities particularly suitable for general purpose GPU (GPGPU) programming. Ashli is a layer above the other shader languages that internally supports reading shaders written in GLSL and HLSL, thereby providing a higher level of flexibility in GPU programming.

Other high-level programming languages have emerged in recent years that focus more on the GPGPU functionality as opposed to graphics specific constructs. Some such languages are Brook [44], Scout [45], Microsoft Accelerator [46], CGiS [47], and the Glift template library [48]. Performance and other comparison characteristics for these languages have been studied[51] to provide a guideline for use in specific applications. CUDA [49] is one of the latest development tools from NVIDIA aimed at GPGPU computing. This promises to eliminate stream shader programming and GPUs can be programmed through multi-threaded C programming for exponential information flow.

Studies have shown that GPUs exceed the number of floating point operations per second and memory bandwidth on comparable CPUs. For example, a 3GHz Intel Pentium 4 processor peaks at 12 GFLOPS (Giga-

Floating Point Operations) with ~6 GB/sec of memory bandwidth as opposed to an ATI Radeon X1800 XT GPU that peaks at 83 GFLOPS with 42 GB/sec of memory bandwidth. This is an improvement of almost 600% in floating point operations. The number of transistors that a GPU can hold is up to 222 million compared to 50 million on an Intel Pentium 4 CPU, an increase of over 400%. Clearly, it can be seen that GPUs promise a tremendous amount of computing power than their CPU counterparts [50, 52]. The technological advancements in GPU hardware have been predicted to follow a pace equal to three-times that of Moore's law. In addition, most computers and workstations currently have a GPU. These performance gains could be instantly realized without the need to purchase additional hardware. If a computer is lacking a GPU, a robust graphics card can be purchased for as little as $100-$400 to acquire tremendous processing power. Figure 3 compares the performance curves of GPUs (NVIDIA and ATI) versus CPUs (Intel) in recent years.

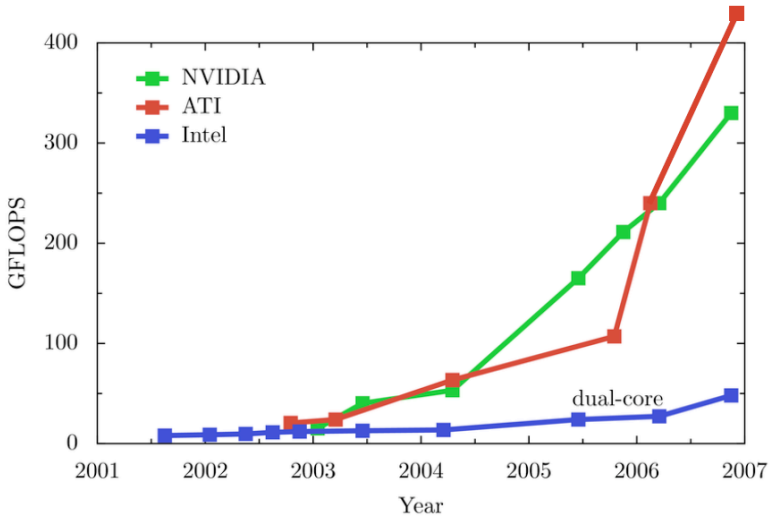


Figure 3 Floating point operation increase of GPUs and CPUs in the past 6 years
(Figure Courtesy: www.gpgpu.org)

If these performance gains could be harnessed either on a single computer, a cluster, or a network of workstations (common in many companies and academic institutions), problems currently requiring enormous computational resources could be solved on commodity hardware. As identified in the introduction, large-scale, multi-objective optimization offers tremendous benefits to companies and researchers, if they have access to immense computational resources. By taking advantage of the power of GPUs, a new source of resources, already available, can become practically usable.

III. Methodology

A. GPU Formulation of PSO Algorithm

Commodity Graphical Processing Units (GPUs), commonly known as graphics cards or video cards were traditionally used for visualization purposes until recently. A user could control various parameters in a graphics code, but the underlying functionality and sequence of operations were fixed. In recent years, this fixed functionality has been replaced with the capability to perform not only graphical operations but also general purpose computing. In 2004, the industry open standard OpenGL 2.0 API was released providing a formal channel for programmability of vertex and fragment shading operations under core OpenGL specifications [53]. Along with a hardware programmable component, hardware advancements have made GPUs capable general-purpose processors capable of very high computational speeds for a variety of scientific applications. Their speed is attributed to their highly data parallel architecture. GPUs take advantage of their hardware parallelism, meaning that computations can be performed on multiple data simultaneously based on the Single Instruction Multiple Data (SIMD) technique.

Although the programmable functions in GPU are graphical in context, the underlying operations are mathematical. Since these operations can be performed dramatically faster than on a traditional CPU, GPUs are increasingly becoming the mainstream for scientific and computation intense operations. Figure 4 is a very simplified view of a fixed function graphics pipeline containing relevant information on data traversal from within

the graphics application to the frame buffer. A frame buffer is the region of the graphics memory that is modified as a result of OpenGL rendering. In a general sense, the frame buffer corresponds to an OpenGL rendering in a window.

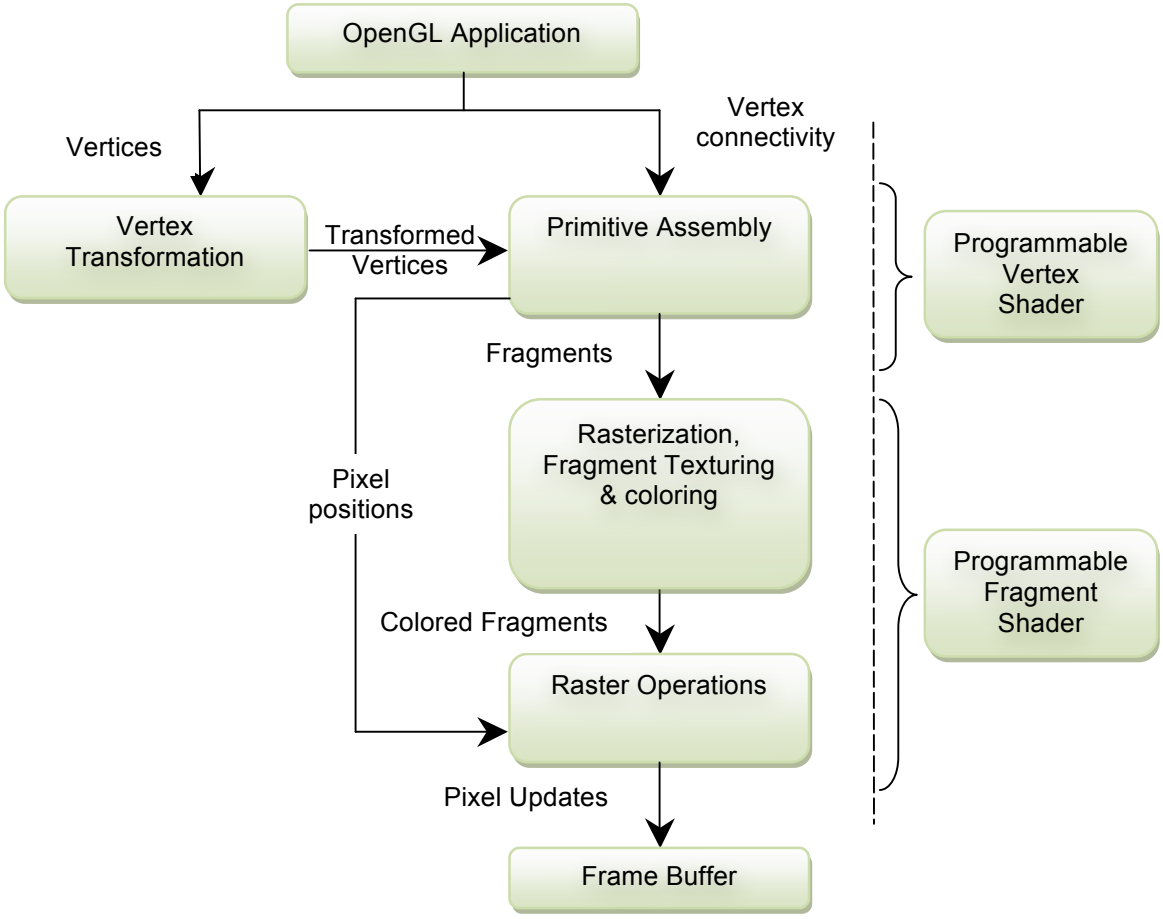


Figure 4 Simplified Graphics Pipeline (programmable components indicated)

In the vertex transformation component, the input vertices are appropriately transformed and passed to the assembly component where the vertices are assembled into a geometric primitive. Also, per vertex operations such as lighting, texture coordinates, clipping against view frustum are computed in these components. Geometric primitives that passed through the primitive assembly component in the pipeline are decomposed into smaller units corresponding to pixels in the destination frame buffer in a process termed rasterization. Each decomposed small unit is called a fragment. For example, if a line covers 10 pixels on the screen, rasterization converts the line geometry information obtained from vertex primitive assembly component into 10 fragments. Each of these fragments is then subjected to various fragment processing operations such as texture mapping, fog, and coloring. The last stage of the graphics pipeline includes performing various per-fragment operations such as pixel ownership test, scissor test, alpha test, stencil test, and the depth test. The underlying operations for vertex and fragment processing are essentially mathematical and can be replaced by programmable vertex and fragment shaders as indicated on the right side of the Figure 4. Figure 5 is a visual summary of the various stages involved in vertex and fragment processing as explained above.

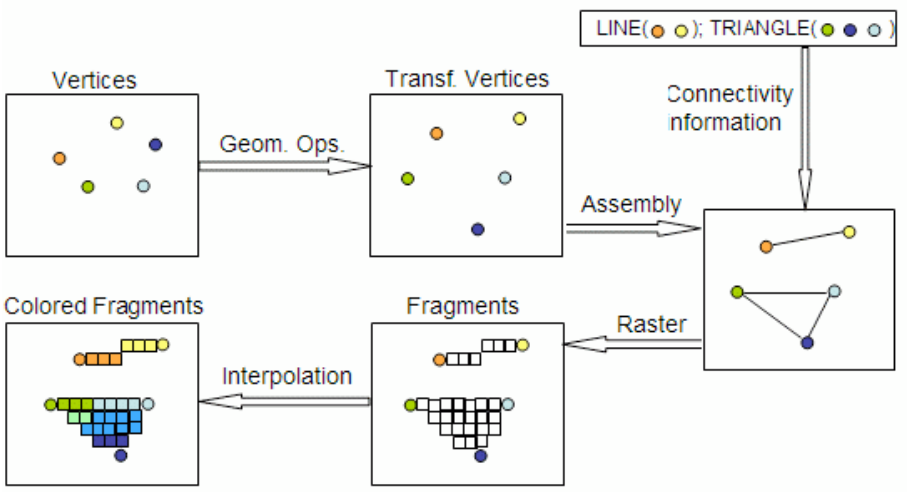


Figure 5 Visual Summary of a Fixed Functionality Graphics Pipeline
(Figure Courtesy: www.lighthouse3d.com)

A. Choice of GLSL as Shading Language

As outlined in section II D, there are a handful of shading languages available to interface with graphics hardware. From the available choice of shading languages, GLSL was chosen for this research for the following reasons:

1. It is a high-level shading language that integrates directly with the OpenGL standard.
2. It is designed with intent for expansion and increased usability in the future. For example, current day graphics cards do not support double precision real valued data types but the pace of their advancements potentially support them in the near future. GLSL specifications support for such future developments and hence adaptation can be made with minimal alterations to vertex or fragment shaders.
3. It is cross platform compatible. Therefore, the shader can be re-used on workstations running different operating systems without any change in the code.
4. It supports most GPU chip makers (e.g. NVIDIA, ATI). With minor hardware alterations, GLSL can be used on a wide variety of GPUs.
5. It closely resembles C/C++ in its programming syntax.
6. It has in-built functions and reserved data types that are graphics in context and are derived from OpenGL. This means a non-graphical developer might have a considerable learning curve before realizing the full potential of GLSL. However, when compared to operating system specific (e.g., Microsoft Accelerator, HLSL, etc) or GPU hardware specific (e.g., CUDA) shaders, GLSL provides the flexibility of working with various operating systems and graphics hardware.

B. Vertex and Fragment Shaders

Both vertex and fragment shaders can provide hardware acceleration for execution of specific portions of a PSO code. However, marked differences between the two necessitate careful consideration of how to proceed. Output from a vertex shader is sent as input to the fragment shader (as seen in the graphics pipeline, Figure 4 and Figure 5), which in turn produces usable output to the main application. In other words, using a vertex-shader is a two-step process. Output from the fragment shader can directly be passed into the main application. Additionally, the fragment shader computes interpolated pixel values for the data provided from the vertex shader causing a possible loss of data or precision. Therefore, a logical choice is to use a fragment shader for this research.

C. Formulation for GPU Computations

Shaders typically work very well with two dimensional textures (analogous to 2D arrays on CPUs). Although 1D and 3D arrays are supported by GPUs, it is typically faster to compute and operate on 2D textures. Since the primary data holders in PSO are swarm members and their locations in the design space, it is a logical first step to create a 2D texture that can hold the design variable values for all swarm members. Older OpenGL releases (pre 2.0) are compatible only with square textures (i.e. of size 2^n – 32, 64, 128, etc). Therefore, a 2D texture of size 40 x 55 previously required creation of a texture of size 64 x 64 where unused texture coordinates would be filled with

zeroes. Although this approach is not a very efficient procedure, it previously served as a good work around to deal with operations on non-square textures. The latest release of OpenGL however addresses this issue and can handle arbitrary rectangular textures, where texture memory can be fully utilized, and hence used for implementation in this research.

The first step in transferring data to the GPU is to prepare OpenGL for off-screen rendering through a Frame Buffer Object (FBO). Graphical objects typically are represented by 8-bit precision each for red, green, blue and alpha channels on a graphics window (computer screen). The purpose of a frame buffer object is to set up off-screen computations in a 32-bit floating-point precision manner and eliminate 8-bit precision for the red, green, blue and alpha channels. The next step is to define appropriate arrays and textures for facilitating inputs and outputs between CPUs and GPUs. The format of the textures created is GPU hardware specific. For example, the texture format on an NVIDIA GPU is denoted by '`GL_FLOAT_R32_NV`' and a texture format on ATI GPU is denoted by '`GL_RGBA_FLOAT32_ATI`'. Additionally, an orthogonal projection and a viewport are needed to provide a one-to-one correspondence between geometry coordinates (used in rendering) and texture coordinates (data input) and pixel coordinates (data output). All these parameters can be set while initializing the FBO.

Design variables for each swarm member are stored in an array and uploaded into the GPU memory as a rectangular texture. The design variable values for each swarm member are filled into each column of the rectangular texture. Figure 6 shows an example 'design variable texture' of size *n*x*m* with the data entry and storage sequence indicated by dashed arrows within the cells.

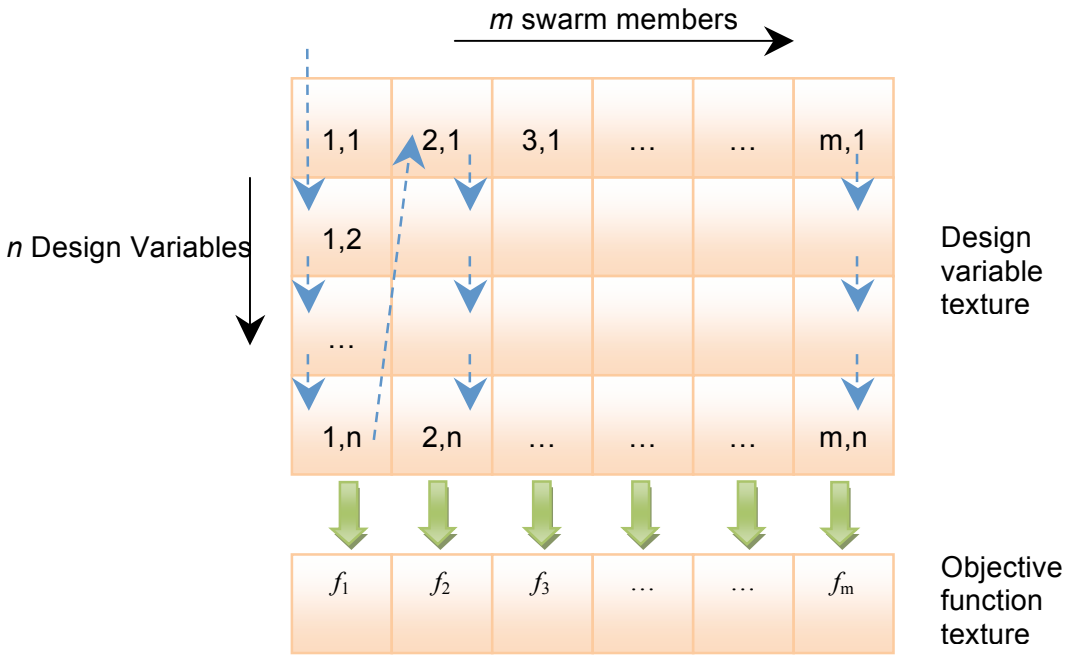


Figure 6 Data Entry Sequence in a Texture and its Use for Objective Function Evaluation

In the design variable texture, '*m*' is the number of swarm members and '*n*' is the number of design variables. The lower rectangular 'objective function texture' of size *1*x*m* holds the objective function values computed from each column of swarm members *1* through *m* from the design variable texture (Multiple Data). Each objective function texture entry requires a column of information (*1* through *n*) from the design variable texture.

D. GPU Implementation

In a PSO optimization routine, the bulk of the computational work comes from objective function evaluations. Thus, it was theorized that if objective function evaluations were delegated to the GPU, the efficiency of PSO would increase due to its data parallel architecture. Although the costs of accessing the main memory on a CPU for

input/output of data into the GPU are high, the benefits of data parallelization would outweigh these CPU-GPU network latencies. An overview of the GPU implementation of PSO with digital pheromones is outlined in Figure 7.

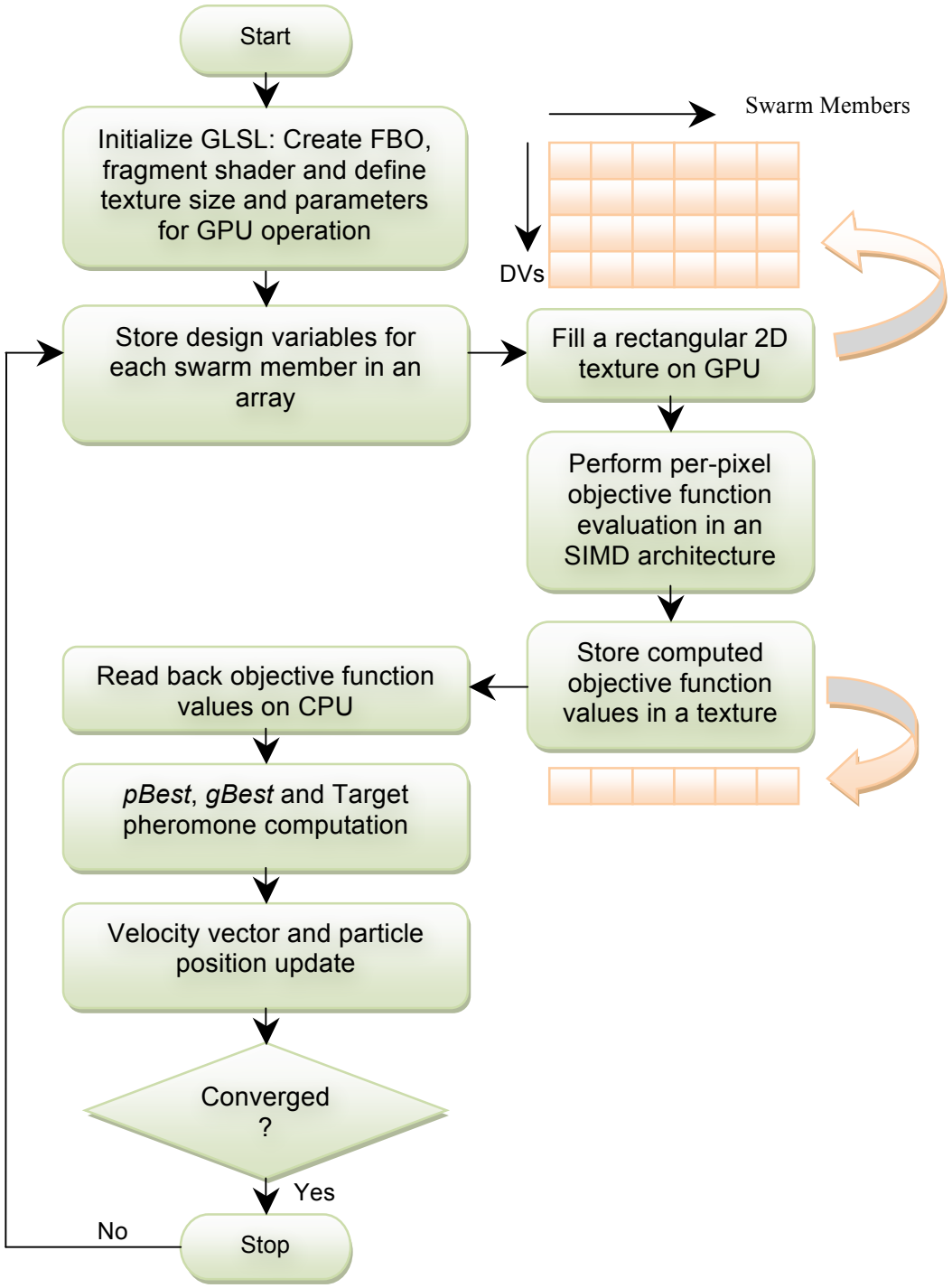


Figure 7 Flowchart for GPU Hardware Acceleration of PSO with Digital Pheromones

The GLSL initialization phase includes preparing the GPU for computations within the framework explained in section III.C. Therefore, this stage involves defining and creating textures for off-screen computations. Design

variables for each swarm member are stored into an array that automatically fills the design variable 2D texture as explained through Figure 6. The fragment shader is then invoked to perform per-pixel objective function evaluations. The fragment shader program consists of instructions to compute the objective function and is executed via rendering a quadrilateral to an off-screen buffer initialized in FBO. Therefore, with a single instruction, computations are performed on multiple data (swarm members) at once to compute the objective function.

IV. Results

In this section, results from implementing PSO with digital pheromones on a GPU are presented. Problems 1 – 10 (shown in Table 2) were used as test cases. Full mathematical descriptions for these test problems can be found in [54-56].

Table 2 Test problem matrix for GPU parallelization

Problem	Test Problem	Dimensions
7.2.1	Camelback function	2
7.2.2	Himmelblau function	2
7.2.3	Rosenbrock function	5
7.2.4	Ackley's path function	10
7.2.5	Dixon and Price function	15
7.2.6	Ackley's path function	20
7.2.7	Levy function	25
7.2.8	Sum of Squares function	30
7.2.9	Spherical function	40
7.2.10	Griewank function	50

A. Test Problem Settings

The pheromone parameters used for the GPU implementation follows the values as established by the serial implementation of PSO with digital pheromones. Therefore, the value of c_3 for lower dimensional problems (2D through 5D) is different from that of higher dimensional problems (above 5D). The values are:

- $c_3 = \begin{cases} 2.0 \text{ for problems } 7.2.1 - 7.2.3, \text{ no decay} \\ 5.0 \text{ for problems } 7.2.4 - 7.2.10, \text{ no decay} \end{cases}$

- Pheromone decay, $\lambda_p = 0.95$, and
- Move limit decay, $\lambda_{ML} = 0.95$

Though customization of parameters for each problem would further improve solution characteristics, the default parameter values catered well for most problems. A total of 35 trial runs were performed for each test case using the GPU method, and were benchmarked against test runs from CPU. Since GPUs, as of the time the research was done, did not support double precision computations, test runs were executed using single precision. Therfore, test problems listed in Table 2 were executed both on CPU and GPU on a single workstation with single precision for a fair comparison. Also to emphasize the difference in performance between CPU and GPU, the test runs were performed only on the digital pheromone implementation of PSO. Basic PSO without digital pheromones was not implemented.

The CPU used was an Intel Xeon processor (3.2 GHz) of a RedHat Enterprise Linux workstation with 2MB of cache memory. The system memory was 2GB DDR. The GPU used was an NVIDIA Quadro FX 4400 with 512MB of DDR memory. The NVIDIA driver version at the time of the code execution was 169.07. The algorithm was implemented using the C++ programming language, and the GPU implementation was made in GLSL, as described in section III.A. As a general rule of thumb, the swarm size was defined as 10 times the number of design variables, and was capped at 500 per processor as the dimensionality increased.

B. Results and Discussion

Table 3 provides a summary of results obtained from solving problems 7.2.1 – 7.2.10. Values obtained from the CPU and GPU are indicated against each problem number in the table. The average, smallest and standard deviation

of the objective function values were noted along with averages of solution duration and number of iterations as well.

Table 3 Results obtained from GPU implementation

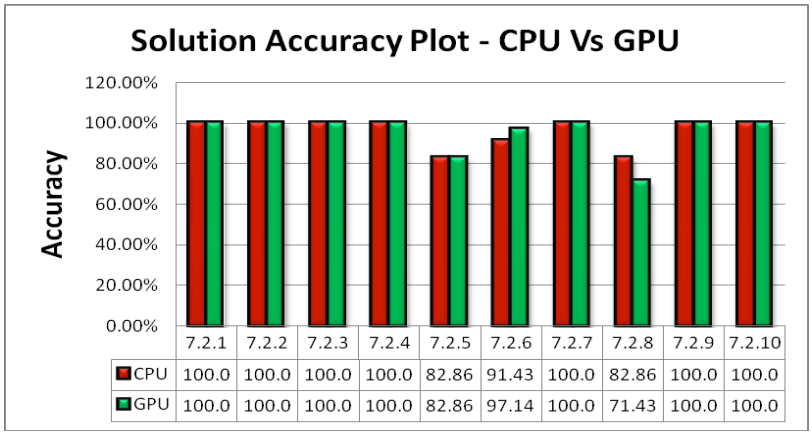
CPU/GPU	Solution Accuracy (%)	Objective Function		
		Average	Smallest	Std Dev
7.2.1 (CPU)	100.00%	-1.032	-1.032	0.000
7.2.1 (GPU)	100.00%	-1.032	-1.032	0.000
7.2.2 (CPU)	100.00%	0.000	0.000	0.000
7.2.2 (GPU)	100.00%	0.000	0.000	0.000
7.2.3 (CPU)	100.00%	0.000	0.000	0.000
7.2.3 (GPU)	100.00%	0.000	0.000	0.000
7.2.4 (CPU)	100.00%	0.000	0.000	0.000
7.2.4 (GPU)	100.00%	0.000	0.000	0.000
7.2.5 (CPU)	82.86%	0.261	0.000	0.592
7.2.5 (GPU)	82.86%	0.382	0.000	0.977
7.2.6 (CPU)	91.43%	0.077	0.000	0.262
7.2.6 (GPU)	97.14%	0.053	0.000	0.163
7.2.7 (CPU)	100.00%	0.129	0.129	0.000
7.2.7 (GPU)	100.00%	0.004	0.004	0.000
7.2.8 (CPU)	82.86%	0.286	0.001	0.329
7.2.8 (GPU)	71.43%	0.298	0.003	0.321
7.2.9 (CPU)	100.00%	0.000	0.000	0.000
7.2.9 (GPU)	100.00%	0.000	0.000	0.000
7.2.10 (CPU)	100.00%	0.005	0.000	0.006
7.2.10 (GPU)	100.00%	0.008	0.000	0.008

Legend: 7.2.1 – Camelback 2D, 7.2.2 – Himmelblau 2D, 7.2.3 – Rosenbrock 5D, 7.2.4 – Ackley 10D, 7.2.5 – Dixon and Price function15D, 7.2.6 – Ackley's path function 20D, 7.2.7 – Levy function 25D, 7.2.8 – Sum of squares function 30D, 7.2.9 – Spherical function 40D, 7.2.10 – Griewank function 50D. (CPU) – Results from CPU implementation, (GPU) – Results from GPU implementation.

It can be seen from the table that the objective function values returned by the GPU were extremely close to the values returned by the CPU, in almost all test cases. In seven out of 10 test cases, the average objective function values returned by GPU were equal to or improved when compared to CPU. For example, on the 25 dimensional Levy Function (problem 7.2.7), there is a ~97% improvement in the solution value for GPU (0.004) when compared

to the CPU result (0.129). Also, the improvement in the solution value was very consistent on the GPU implementation as apparent from the standard deviation (0.000).

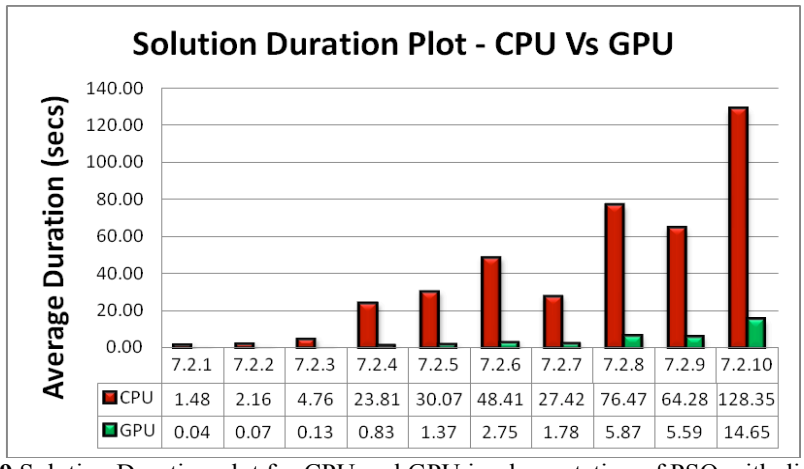
Test problems such as Ackley's path function (7.2.4, 7.2.6) and Levy function (problem 7.2.7) are prone to errors in accuracy due to having trigonometric relations in the objective functions. However, the solution accuracies were not compromised because of this reason. Figure 8 shows a visual comparison of solution accuracies between the CPU and GPU results. With the exception of the 30D Sum of squares function (problem 7.2.8), the solution accuracies on all other problems for the GPU implementation were either equal to or better than that of the CPU implementation. This suggests that GPUs can be capable co-processors for computations and not have a major effect in the outcome of the solution qualities.



	7.2.1	7.2.2	7.2.3	7.2.4	7.2.5	7.2.6	7.2.7	7.2.8	7.2.9	7.2.10
CPU	100.0	100.0	100.0	100.0	82.86	91.43	100.0	82.86	100.0	100.0
GPU	100.0	100.0	100.0	100.0	82.86	97.14	100.0	71.43	100.0	100.0

Figure 8 Solution accuracy plot for CPU and GPU implementation of PSO with digital pheromones

Figure 9 shows the solution duration charts for the GPU implementation as compared with the CPU implementation.



	7.2.1	7.2.2	7.2.3	7.2.4	7.2.5	7.2.6	7.2.7	7.2.8	7.2.9	7.2.10
CPU	1.48	2.16	4.76	23.81	30.07	48.41	27.42	76.47	64.28	128.35
GPU	0.04	0.07	0.13	0.83	1.37	2.75	1.78	5.87	5.59	14.65

Figure 9 Solution Duration plot for CPU and GPU implementation of PSO with digital pheromones

It can be seen that the solution duration for all test problems dramatically reduced on the GPU implementation as opposed to the CPU counterpart. The reduction in the solution duration is a clear indication that GPUs are not just comparable in performance to traditional CPUs, but they could exceed the throughput by a factor of about 10 or more in terms of solution times. For example, the average solution time for the 10 dimensional Ackley's path function (problem 7.2.4) on a CPU was 23.81 seconds where as the GPU took 0.83 seconds. This is approximately a 96.5% decrease. Similarly, the 50 dimensional Griewank function (problem 7.2.10) resulted in an 88% decrease in solution time on a GPU (14.65 seconds) when compared to the solution time on a CPU (128.35 seconds). The same trend was seen in all the test problems, with reduction in solution durations ranging from 88% - 97% were observed.

Table 4 Comparison of solution duration and number of iterations on CPU Vs GPU

CPU/GPU	Average Duration (secs per run)	Average # iterations	% Decrease in duration
7.2.1 (CPU)	1.48	73.97	97.03%
7.2.1 (GPU)	0.04	73.86	
7.2.2 (CPU)	2.16	107.71	96.63%
7.2.2 (GPU)	0.07	115.49	
7.2.3 (CPU)	4.76	93.54	97.19%
7.2.3 (GPU)	0.13	94.89	
7.2.4 (CPU)	23.81	231.11	96.50%
7.2.4 (GPU)	0.83	234.11	
7.2.5 (CPU)	30.07	192.11	95.46%
7.2.5 (GPU)	1.37	189.40	
7.2.6 (CPU)	48.41	229.69	94.31%
7.2.6 (GPU)	2.75	235.77	
7.2.7 (CPU)	27.42	103.03	93.49%
7.2.7 (GPU)	1.78	103.80	
7.2.8 (CPU)	76.47	235.89	92.33%
7.2.8 (GPU)	5.87	235.00	
7.2.9 (CPU)	64.28	146.57	91.30%
7.2.9 (GPU)	5.59	140.29	
7.2.10 (CPU)	128.35	228.94	88.59%
7.2.10 (GPU)	14.65	231.46	

Legend: 7.2.1 – Camelback 2D, 7.2.2 – Himmelblau 2D, 7.2.3 – Rosenbrock 5D, 7.2.4 – Ackley 10D, 7.2.5 – Dixon and Price function15D, 7.2.6 – Ackley's path function 20D, 7.2.7 – Levy function 25D, 7.2.8 – Sum of squares function 30D, 7.2.9 – Spherical function 40D, 7.2.10 – Griewank function 50D. (CPU) – Results from CPU implementation, (GPU) – Results from GPU implementation.

Table 4 tabulates the average duration, average number of iterations and the percentage decrease in solution duration through using GPUs when compared to CPU usage alone. The data parallel architecture of a GPU is attributed to this dramatic reduction in solution times. Since GPUs are inherently hardware parallel in architecture, a single instruction can be performed on multiple data simultaneously resulting in enormous time savings as evident from figures Figure 8 and Figure 9 and tables Table 3 and Table 4. Although the amount of time savings can be hardware and problem specific, the results show that GPUs has great potential to outperform CPUs with no marked difference in solution quality for optimization computations. There was also no notable difference in the number of

iterations for each test problem when executed on a CPU or a GPU. This indicates that the data traversal between CPU and GPU did not significantly affect the overall algorithm's performance. This suggests that commodity graphics cards can potentially be a very viable option in optimization computations when time and cost are important factors.

V. Conclusion

This paper presents a novel method for parallelization of PSO with digital pheromones coupled with hardware acceleration using commodity graphics cards. GPU implementation of PSO with digital pheromones has demonstrated dramatic improvement in solution efficiency (of the order of 90% in solution times) without any significant improvement in solution accuracy characteristics. The results show that GPUs not only are capable co-processors but can exceed CPU computations by a factor more than 10 times. The time savings noted in this research was through delegating objective function computations to the GPU alone. More savings are possible with further offloading of computations from CPU to GPU (e.g., velocity vector, pheromone computations). Since most computational workstations these days come with decent video cards, it will not be long before GPUs are accepted as mainstream computational cores for evolutionary optimization methods such as PSO.

Refining the performance of digital pheromones to solve a wide range of optimization problems is an ongoing venture; some of the near future goals for this research include GPU parallelization of velocity vector, digital pheromones and distributed parallel computing with CPUs and GPUs to rake further gains in computational efficiency in PSO.

References

[1] Kennedy, J., and Eberhart, R. C., "Particle Swarm Optimization", *Proceedings of the 1995 IEEE International Conference on Neural Networks*, Vol. 4, Inst. of Electrical and Electronics Engineers, Piscataway, NJ, 1995, pp. 1942-1948.

[2] Eberhart, R. C., and Kennedy, J., "A New Optimizer Using Particle Swarm Theory", *Proceedings of the Sixth International Symposium on Micro Machine and Human Science*, Inst. of Electrical and Electronics Engineers, Piscataway, NJ, 1995, pp. 39-43.

[3] Russell C. Eberhart and Yuhui Shi, "Particle swarm optimization: Developments, applications, and resources", *In Proceedings of the 2001 Congress on Evolutionary Computation 2001*, 81–86.

[4] J.F. Schutte. Particle swarms in sizing and global optimization. Master's thesis, University of Pretoria, Department of Mechanical Engineering, 2001.

[5] A. Carlisle and G. Dozier. An off-the-shelf pso. In Proceedings of the Workshop on Particle Swarm Optimization, 2001, Indianapolis.

[6] Kalivarapu, V., Foo, J. L., Winer, E. H., "Implementation of Digital Pheromones for Use in Particle Swarm Optimization", 47th AIAA/ASME/ASCE/AHS/ASC Structures, Structural Dynamics, and Materials Conference, 2nd AIAA Multidisciplinary Design Optimization Specialist Conference, Newport, RI, 1-4 May 2006.

[7] Kalivarapu, V., Foo, J., Winer, E., "A Parallel Implementation of Particle Swarm Optimization Using Digital Pheromones", 11th AIAA/ISSMO Multidisciplinary Analysis and Optimization Conference, AIAA-2006-6908-694, Portsmouth, VA, September 2006.

[8] Kalivarapu, V., Winer, E., "A Statistical Analysis of Particle Swarm Optimization With and Without Digital Pheromones", 48th AIAA/ASME/ASCE/AHS/ASC Structures, Structural Dynamics, and Materials Conference, April 23-26, 2007, Honolulu, HI, AIAA 2007-1882.

[9] "General Purpose Computation Using Graphics Hardware (GPGPU)", http://www.gpgpu.org, accessed February 2007.

[10] Stewart, N., Leach, G., John, S., "Improved CSG Rendering using Overlap Graph Subtraction Sequences", International Conference on Computer Graphics and Interactive Techniques in Australasia and South East Asia - GRAPHITE 2003, pp. 47-53

[11] Agarwal, P., Krishnan, S., Mustafa, N., and Venkatasubramanian, S., "Streaming Geometric Optimization Using Graphics Hardware". Proceedings of 11th European Symposium on Algorithms, Sep 2003.

[12] Pascucci, V., "Isosurface Computation Made Simple: Hardware Acceleration, Adaptive Refinement and Tetrahedral Stripping", Proceedings of VisSym 2004.

[13] Boubekeur, T., Schlick, C., "Generic Mesh Refinement on GPU", Proceedings of Graphics Hardware 2005.
[14] Mustafa, N., Koutsofios, E., Krishnan, S., and Venkatasubramanian, S., "Hardware Assisted View Dependent Map Simplification", 17th Annual ACM Symposium on Computational Geometry, June 2001.
[15] Charalambous, M., Trancoso, P., and Stamatakis, A., "Initial Experiences Porting a Bioinformatics Application to a Graphics Processor", Proceedings of the 10th Panhellenic Conference in Informatics (PCI 2005).
[16] Graca, G., Defour, D., "Implementation of float-float operators on graphics hardware", 7th conference on Real Numbers and Computers, RNC7, Nancy, France, July 2006.
[17] Hu X H, Eberhart R C, Shi Y H., "Engineering Optimization with Particle Swarm", *IEEE Swarm Intelligence Symposium*, 2003: 53-57.
[18] G. Venter and J. Sobieszczanski-Sobieski, "Multidisciplinary optimization of a transport aircraft wing using particle swarm Optimization", *In 9th AIAA/ISSMO Symposium on Multidisciplinary Analysis and Optimization 2002*, Atlanta, GA.
[19] P.C. Fourie and A.A. Groenwold, "The particle swarm algorithm in topology optimization", *In Proceedings of the Fourth World Congress of Structural and Multidisciplinary Optimization 2001*, Dalian, China.
[20] Shi, Y., Eberhart, R., "Parameter Selection in Particle Swarm Optimization", Proceedings of the 1998 Annual Conference on Evolutionary Computation, March 1998.
[21] Shi, Y., Eberhart, R., "A Modified Particle Swarm Optimizer", *Proceedings of the 1998 IEEE International Conference on Evolutionary Computation*, pp 69-73, Piscataway, NJ, IEEE Press May 1998.
[22] Natsuki H, Hitoshi I., "Particle Swarm Optimization with Gaussian Mutation", *Proceedings of IEEE Swarm Intelligence Symposium*, Indianapolis, 2003:72-79.
[23] Hu, X., Eberhart, R., Shi, Y., "Swarm Intelligence for Permutation Optimization: A Case Study of n-Queens Problem", *IEEE Swarm Intelligence Symposium 2003, Indianapolis, IN, USA*.
[24] Venter, G., Sobieszczanski-Sobieski, J., "Particle Swarm Optimization", *AIAA Journal*, Vol.41, No.8, 2003, pp 1583-1589.
[25] Hu, X., Eberhart, R., "Solving Constrained Nonlinear Optimization Problems with Particle Swarm Optimization", 6th World Multiconference on Systemics, Cybernetics and Informatics (SCI 2002), Orlando, USA.
[26] Schutte, J., Reinbolt, J., Fregly, B., Haftka, R., George, A., "Parallel Global Optimization with the Particle Swarm Algorithm", *Int. J. Numer. Meth. Engng*, 2003.
[27] Koh, B, George A. D., Haftka, R. T., Fregly, B., "Parallel Asynchronous Particle Swarm Optimization", International Journal For Numerical Methods in Engineering", *International Journal of Numerical Methods in Engineering*, 67:578-595, 2006, Published online 31 January 2006 in Wiley InterScience, DOI: 10.1002/nme.1646
[28] Hu, X., Eberhart, R., Shi, Y., "Particle Swarm with Extended Memory for Multiobjective Optimization", *Proceedings of 2003 IEEE Swarm Intelligence Symposium,* pp 193-197, Indianapolis, IN, USA, April 2003, IEEE Service Center.
[29] Tayal, M., Wang, B., "Particle Swarm Optimization for Mixed Discrete, Integer and Continuous Variables", 10th AIAA/ISSMO Multidisciplinary Analysis and Optimization Conference, Albany, New York, Aug 30-1, 2004.
[30] Walter, B., Sannier, A., Reiners, D., Oliver, J., "UAV Swarm Control: Calculating Digital Pheromone Fields with the GPU", *The Interservice/Industry Training, Simulation & Education Conference (I/ITSEC),*Volume 2005 (Conference Theme: One Team. One Fight. One Training Future).
[31] Gaudiano, P, Shargel, B., Bonabeau, E., Clough, B., "Swarm Intelligence: a New C2 Paradigm with an Application to Control of Swarms of UAVs", In Proceedings of the 8th International Command and Control Research and Technology Symposium, 2003.
[32] Colorni, A., Dorigo, M., Maniezzo, V., "Distributed Optimization by Ant Colonies", *In Proc. Europ. Conf. Artificial Life,* Editors: F. Varela and P. Bourgine, Elsevier, Amsterdam, 1991.
[33] Dorigo, M., Maniezzo, Colorni, A., "Ant System: Optimization by a Colony of Cooperating Agents", *In IEEE Trans. Systems, Man and Cybernetics*, Part B, Vol. 26, Issue 1, pp 29-41, 1996.
[34] White, T., Pagurek, B., "Towards Multi-Swarm Problem Solving in Networks", *icmas*, p. 333, Third International Conference on Multi Agent Systems (ICMAS'98), 1998.
[35] Parunak, H., Purcell M., O'Conell, R., "Digital Pheromones for Autonomous Coordination of Swarming UAV's". *In Proceedings of First AIAA Unmanned Aerospace Vehicles, Systems, Technologies, and Operations Conference*, Norfolk, VA, AIAA, 2002.
[36] Kalivarapu, V., Foo, J-L, Winer, E., "Improving Solution Characteristics of Particle Swarm Optimization using Digital Pheromones", Journal of Structural and Multidisciplinary Optimization, Accepted for publication, January 2008

[37] Sutter, H., "The Free Lunch is Over: A Fundamental Turn Toward Concurrency in Software", Dr. Dob's Journal, 30(3), March 2005, website: http://www.gotw.ca/publications/concurrency-ddj.htm, accessed February 2007
[38] Owens, J., Luebke, D., Govindaraju, N., Harris, M., Krüger, J., Lefohn, A., and Purcell. T., "A Survey of General-Purpose Computation on Graphics Hardware" In *Eurographics 2005, State of the Art Reports*, August 2005, pp. 21-51
[39] Fernando, R., Kilgard, M., *The Cg Tutorial: The Definitive Guide to Programmable Real-Time Graphics,* Addison-Wesley Publications, 2003, ISBN: 0321194969
[40] Rost, R., *OpenGL(R) Shading Language (2nd Edition) (OpenGL),* Addison-Wesley Publications, 2006, ISBN: 0321334892
[41] DirectX 9 High Level Shading Language (Microsoft DirectX 9 HLSL), http://msdn2.microsoft.com/en-us/library/ms810449.aspx, accessed February 2007
[42] McCool, M., D Toit, S., Popa T., Chan, B., Moule K., "Shader Algebra", ACM Transactions on Graphics 23,3 August 2004, pp. 787-795
[43] Bleiweiss, A., Preetham, A., "Ashli-Advanced Shading Language Interface", ACM Siggraph Course Notes, July 2003, http://ati.amd.com/developer/SIGGRAPH03/AshliNotes.pdf, accessed February 2007
[44] Buck, I., Foley, T., Horn, D., Sugerman J., Fatahalian K., Houston, M., Hanrahan, P., "Brook for GPUs: Stream Computing on Graphics Hardware", ACM Transactions on Graphics 23, 3, August 2004, pp. 777-786
[45] McCormick, P., Inman J., Ahrens, J., Hansen, C., Roth, G., "Scout: A Hardware-Accelerated System for Quantitatively Driven Visualization and Analysis", In IEEE Visualization 2004, October 2004, pp. 171-178
[46] Tarditi D., Puri, S., Oglesby, J., "Accelerator: Using Data-Parallelism to Program GPUs for General Purpose Uses", In Proceedings of the Twelfth International Conference on Architectural Support for Programming Languages and Operating Systems, October 2006
[47] Lucas P., Fritz, N., Wilhelm, R., "The CGiS Compiler", In proceedings of the 15th International Conference on Compiler Construction", vol. 3923 of Lecture Notes in Computer Science, Springer, March 2006, pp. 105-108
[48] Lefohn, A., Kniss, J., Strzodka, R., Sengupta, S., Owens, J., "Glift: An Abstraction for Generic, Efficient GPU Data Structures", ACM Transactions on Graphics 26, 1, January 2006, pp. 60-99
[49] "NVIDIA CUDA Homepage", http://developer.nvidia.com/object/cuda.html, accessed February 2007
[50] Wasson, S., "ATI Stakes Claims on Physics, GPGPU Ground", The Tech Report – Personal Computing Explored", Oct 11, 2005 - http://techreport.com/onearticle.x/8887, accessed February 2007
[51] Owens, J., Luebke, D., Govindaraju, N., Harris, M., Krüger, J., Lefohn, A., and Purcell. T.,. "A Survey of General-Purpose Computation on Graphics Hardware", Volume 26 (2007). Computer Graphics Forum, Accepted for publication in March 2007 or June 2007
[52] Kruger J., Schiwietz, T., Kipfer, P., Westermann, R., "Numerical Simulations on PC Graphics Hardware", EuroPVM/MPI 2004, LNCS 3241, pp. 442-449, Springer-Verlag Berlin Heidelberg, 2004
[53] Rost, R., "OpenGL Shading Language", Second Edition, Addison Wesley Publications, ISBN: 032-133-4892, 2006
[54] Engelbrecht, A., "Fundamentals of Computational Swarm Intelligence, Wiley Publications, NY, ISBN: 047-009-1916, 2006
[55] "Test Problems in Global Optimization", Web Reference: http://www-optima.amp.i.kyoto-u.ac.jp/member/student/hedar/Hedar_files/TestGO_files/Page364.htm, cited May 23, 2008
[56] "GEATbx: Example Functions (Single and Multi-objective Functions) 2 Parametric Optimization", Web Reference: http://www.geatbx.com/docu/fcnindex-01.html, Cited May 23, 2008