Digital Pheromone Implementation of PSO with Velocity Vector 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. Particularly, the velocity vector computations are carried out on graphics hardware. 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 serial, parallel and GPU computing environments. The GPU implementation was limited to computing the objective function values alone. 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. This paper presents a method to implement velocity vector computations on a GPU along with objective function evaluations. Three different modes of implementation are studied and presented - First, CPU-CPU where objective function and velocity vector are calculated on CPU alone. Second, GPU-CPU where objective function is computed on the GPU and velocity vector is computed on GPU. Third, GPU-GPU where objective function and velocity vector are both evaluated on the GPU. The results from these three implementations are presented followed by conclusions and recommendations on the best approach for utilizing the full potential of GPUs for PSO.

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 initially randomized 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 (gBest), and the best position attained by all particles (pBest) 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, 11, b) geographic information systems 12, 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\textsuperscript{15}, and e) solving dense linear systems\textsuperscript{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. Previously, the authors have attempted to use GPUs for performing objective function evaluations and have realized substantial gains in solution efficiencies\textsuperscript{17}. The previous work has provided motivation to investigate further the possibility of off-loading CPU computations to the GPU. Specifically, velocity vector calculations are carried out on GPUs and their performance characteristics are studied. Results from CPU objective function – CPU velocity vector, GPU objective function – CPU velocity vector, and GPU objective function – GPU velocity vector are presented and inferences are presented. The developed method has been tested on various multi-modal n-dimensional test problems.

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\textsuperscript{18}. The success of the algorithm has brought substantial attention among the research community in the recent past\textsuperscript{19, 20}. 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).

\[
V_{iter+1,i} = w_{iter} \times V_{iter,i} \\
+ c_1 \times \text{rand}_{p,i}^{iter+1} (pBest_i - X_{iter,i}) \\
+ c_2 \times \text{rand}_{g,i}^{iter+1} (gBest_i - X_{iter,i}) \\
X_{iter+1,i} = X_{iter,i} + V_{iter+1,i} \\
w_{iter+1} = w_{iter} \times \lambda^w
\]

Equation (1) represents the velocity vector update, representing the direction and magnitude of ith swarm member in a basic PSO in iteration ‘iter’. Each successive iteration is represented by ‘iter+1’. The square braces in equations (1) and (2) indicate an array meaning that the corresponding value (e.g., pBest) is computed for each design variable. \text{rand}_{p,i}^{iter+1}() and \text{rand}_{g,i}^{iter+1}() are unique random numbers generated between 0 and 1 for pBest and gBest components separately for each swarm member in each iteration. The parentheses in randp() and randg() indicate that they are random number generating functions within the computer code. This ensures swarm diversity, meaning that the search is not linear in an n-dimensional space. More about swarm diversity can be obtained from\textsuperscript{21}. \text{c1} and \text{c2} are user definable confidence parameters. Typically, these are set to values of 2.0. ‘pBest’ represents the best position of the particle in its history trail, and ‘gBest’ represents the best particle location in the entire swarm. \text{witer} is termed “inertia” weight, and is used to control the impact of a particle’s previous velocity on the calculation of the current velocity vector. A large value for \text{witer} facilitates global exploration, which is particularly useful in the initial stages of an optimization. A small value allows for more localized searching, which is useful as the swarm moves toward the neighborhood of the optimum\textsuperscript{22, 23}. These characteristics are attributed to the swarm by implementing a decay factor, \lambda^w for the inertia weight, as shown 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\textsuperscript{24, 25}, b) methods for constraint handling\textsuperscript{26, 27}, c) parallel implementation\textsuperscript{28, 29}, d) methods for solving multi-objective optimization problems\textsuperscript{30}, e) methods for solving mixed discrete, integer and continuous variables\textsuperscript{31}. 
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)\textsuperscript{32, 33}. 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\textsuperscript{34, 35}, solving network communication problems\textsuperscript{36}. The concept of digital pheromones is considerably new\textsuperscript{37} 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 – \textit{pBest} and \textit{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 \textit{pBest} and \textit{gBest} are insufficient or inefficient. Figure 1a displays a scenario of a swarm member’s movement whose direction is guided by \textit{pBest} and \textit{gBest} alone. If \( c_1 \gg c_2 \), the particle is attracted primarily towards its personal best position. On the other hand, if \( c_2 \gg c_1 \), the particle is strongly attracted to the \textit{gBest} position. In the scenario dominated by \( c_2 \), as presented in figure 1a, neither \textit{pBest} nor \textit{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 \textit{pBest} and \textit{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, and is not described in this paper to maintain conciseness. The new velocity vector update equation is shown in eq. (4).

Figure 2 Overview of PSO with Digital Pheromones
\[ V_{\text{iter} + 1} = w_{\text{iter}} \times V_{\text{iter}, i} \]
\[ + c_1 \times \text{rand}_{p,i}^{\text{iter} + 1} (p\text{Best}_i[] - X_i[]) \]
\[ + c_2 \times \text{rand}_{g,i}^{\text{iter} + 1} (g\text{Best}_i[] - X_i[]) \]
\[ + c_3 \times \text{rand}_{T,i}^{\text{iter} + 1} (T\text{arg etPheromone}_i[] - X_i[]) \]

\( \text{rand}_{p,i}^{\text{iter} + 1}(), \text{rand}_{g,i}^{\text{iter} + 1}() \) and \( \text{rand}_{T,i}^{\text{iter} + 1}() \) are unique random numbers generated between 0 and 1 for pBest, gBest and target pheromone components separately for each swarm member in each iteration. The parentheses in \( \text{rand}_{\text{iter} + 1}() \) indicate that they are random number generating functions within the computer code. These values are updated in each iteration within the velocity vector equation, resulting in improved swarm diversity. The square braces in equation (6) indicate an array meaning that the corresponding value (e.g., TargetPheromone) is computed for all design variables.

\( c_3 \) is a user defined confidence parameter for the pheromone component of the velocity vector similar to \( c_1 \) and \( c_2 \) in a basic PSO. \( c_1 \) combines the knowledge from the cognitive and social components of the velocity of a particle, and complements their deficiencies. In a basic PSO, the particle swarm does not have a memory of the entire path traversed in the design space apart from the best position of an individual particle (pBest) and the best member’s position in the entire swarm (gBest). The target pheromone component addresses this issue. It is a container that functionally stores the trail path of the swarm and utilizes the best features of pBest and gBest in steering towards a promising location in the design space. The confidence parameter \( c_3 \) determines the extent of influence a target pheromone can have on the swarm when the information from pBest and gBest alone are not sufficient or efficient to determine a particle’s next move. The use of the target pheromone relies heavily on pBest and gBest. If \( c_3 = 0 \), there is no influence of pheromones and the swarm behaves as if in a regular PSO. If either of \( c_1 \) or \( c_2 \) is 0 and \( c_3 > 0 \), then the target pheromone location is essentially determined only by the non-zero component of pBest or gBest and propagated into the velocity vector. This creates a bias thereby doubling the influence of non-zero pBest or gBest components on the swarm. This means that the swarm either explores or exploits the design space with double the intensity, either of which will prevent the swarm from converging. It is therefore essential that the influence of pBest and gBest be balanced (i.e. equal) for the pheromone component to provide accurate assistance in reaching the optimum. The addition of this pheromone component to PSO increases the swarm’s diversity, resulting in an improved search in the design space. However, improvement in diversity most certainly lacks frame invariance.

This paper focuses on increasing the diversity in the swarm using digital pheromones than frame invariance. Although analytical determination of a value for \( c_3 \) is out of the scope of this research, an empirical value has been determined through experimentation. A value between 2.0 and 5.0 has shown good performance characteristics and solved a variety of problems. An inertia weight, \( w_i \) of value 1.0 is initially chosen to preserve the influence of the velocity vector from previous iterations, and gradually decreased using an inertia weight decay factor similar to the one used in a basic PSO.

D. Feasibility of GPUs

Recently, technologies such as hyper threading and multi-core processing 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. 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 42, GLSL 43, HLSL 44, Sh 45, and Ashli 46. 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](image)

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 47, Scout 48, Microsoft Accelerator 49, CGiS 50, and the Glift template library 51. Performance and other comparison characteristics for these languages have been studied 52 to provide a guideline for use in specific applications. CUDA 53 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 53,54,55. 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

American Institute of Aeronautics and Astronautics
acquire tremendous processing power. Figure 1 compares the performance curves of GPUs (NVIDIA and ATI) versus CPUs (Intel) in recent years.

![Figure 1: Floating point operation increase of GPUs and CPUs in the past 6 years (Figure Courtesy: www.gpgpu.org)](image)

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. These features are further enhanced in the most recent OpenGL 3.1 specifications released in March 2009. 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 2 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.
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 2. Figure 3 is a visual summary of the various stages involved in vertex and fragment processing as explained above.
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 2 and Figure 3), 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 generally 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), through a texture format called GL_TEXTURE_2D. Therefore, a 2D texture of size $40 \times 55$ previously required creation of a texture of size $64 \times 64$ 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 called GL_TEXTURE_RECTANGLE_ARB, 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). The purpose of a frame buffer object is to prepare OpenGL for off-screen computations instead of displaying graphical objects on the computer screen. 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 4 shows an example ‘design variable texture’ of size $mn\times n$ with the data entry and storage sequence indicated by dashed arrows within the cells.

**Figure 4** 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 $1 \times m$ 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. With a single instruction, multiple data from each column of the design variable texture can be used to compute the objective function. Since each column computes simultaneously, it will result in an enormous boost in the computation time on the GPU as opposed to a sequential computation using loops on a CPU. The internal format of the texture as described above is denoted by ‘GL_LUMINANCE’ where each texel (or a texture coordinate) is associated with one
single precision floating point number. This format is typically used when a few textures (i.e., one or two) are required for input and output.

Computations requiring more than three or four textures as an input to the GPU can be handled using another texture internal format denoted by ‘GL_RGBA’. RGBA textures can hold four single precision floating point values each for red, green, blue, and alpha channels per each texel. For example, a single GL_RGBA texel can hold values for pBest, gBest, target pheromone and design variable values per each channel as opposed to requiring four GL_LUMINANCE textures. Since multiple texture reads are not required, computations are significantly faster on GL_RGBA textures.

D. Requirement of CPU and GPU for PSO

With the promise of high-speed of GPUs over CPUs, it is logical to assume that PSO can be performed completely on GPUs instead of CPUs. However, the hardware architecture of GPUs renders it a daunting task, if not impossible to port PSO operations on GPUs alone.

Since the fragment shader returns per-pixel results, each pixel is unaware of the results computed by the adjacent pixel. For example, the pixel that computed and stored the objective function value $f_i$ does not know the objective function $f_j$ value computed and stored in the adjacent pixel. A run-through of all objective function values is required for determining pBest, gBest and target Pheromone components of the velocity vector, and each pixel value being precluded from the adjacent pixel prevents the GPU from performing such logical operations. These operations will hence necessarily require the use of a CPU rather than a GPU. Secondly, although population based methods are most suitable for GPU implementations since each swarm member operation is independent of the other, PSO requires generating numerous random numbers. Once again, the hardware architecture of GPUs that do not support bitwise logical operations prevent from generating good random numbers. Although there are ongoing research efforts in this area, current generation GPUs are not capable of generating high-quality random numbers. This impedes the swarm diversity in searching the design space, and hence good solutions. Therefore, it will require a CPU to generate these random numbers and provide them for GPU operations. Although GPU implementations come inherent with these drawbacks, the hardware parallelism and hence the speed of computations on present day GPUs can far outweigh CPU implementations.

E. GPU Implementation

Iterative computations that are independent of each other are typically the best candidates for GPU implementations. In the digital pheromone implementation of PSO, the bulk of computational work come from objective function, digital pheromone and velocity vector evaluations. Since the target pheromone evaluations are dependent upon pBest, gBest, design variable and neighboring pheromones, per-pixel calculations will require information from adjacent pixels that GPU hardware architecture does not support. Therefore, digital pheromone operations cannot be off-loaded to GPUs. Objective function evaluations on a GPU were previously implemented.

Velocity vector calculations are performed for each particle and for each swarm member. However, each computed velocity vector for a particle (and design variable) does not depend upon the values computed for another particle (and design variable). Therefore given that the pBest, gBest and target pheromone information are readily available when the velocity vector is calculated, it would be a logical step to implement them on GPUs.

An overview of velocity vector implementation along with objective function evaluations on GPUs are shown in Figure 5. 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. The textures are chosen to be rectangular of size equal to population size x number of design variables. Since the size of textures do not change in each iteration, they are created just once and re-used. A total of eight input textures are created for storing design variables, pBest, gBest, target pheromone, previous velocity vector values, and random numbers for each of pBest, gBest and target pheromone components. These values are uploaded to textures for GPU computations in each iteration. This means that eight datasets each of size population size x number of variables is transferred from CPU to GPU in each iteration.

American Institute of Aeronautics and Astronautics
With data populated on the textures for GPU operations, two fragment shaders consisting of instructions to compute the objective function and velocity vector are executed. The execution of fragment shader happens via...
rendering a quadrilateral to an off-screen buffer initialized in the FBO. Therefore, with a single instruction, computations are performed on multiple data (swarm members) at once to compute the objective function and the velocity vector.

IV. Results

In this section, results from GPU implementation of velocity vector within a PSO with digital pheromones are presented. For a better comparison metric, the following scenarios are executed and results from each are noted and discussed.

1) CPU Fitness Value + CPU Velocity Vector (CPU-CPU)
2) GPU Fitness Value + CPU Velocity Vector (GPU-CPU)
3) GPU Fitness Value + GPU Velocity Vector (GPU-GPU)

Problems 1 – 8 (shown in Table 2 Test problem matrix for GPU parallelization) were used as test cases. Full mathematical descriptions for these problems can be found in 58-60.

Table 2 Test problem matrix for GPU parallelization

<table>
<thead>
<tr>
<th>Problem</th>
<th>Test Problem</th>
<th>Published Solution</th>
<th>Dimensions</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Camelback function</td>
<td>-1.0316</td>
<td>2</td>
</tr>
<tr>
<td>2</td>
<td>Rosenbrock function</td>
<td>0.000</td>
<td>5</td>
</tr>
<tr>
<td>3</td>
<td>Dixon and Price function</td>
<td>0.000</td>
<td>15</td>
</tr>
<tr>
<td>4</td>
<td>Ackley’s path function</td>
<td>0.000</td>
<td>20</td>
</tr>
<tr>
<td>5</td>
<td>Levy function</td>
<td>0.000</td>
<td>25</td>
</tr>
<tr>
<td>6</td>
<td>Sum of Squares function</td>
<td>0.000</td>
<td>30</td>
</tr>
<tr>
<td>7</td>
<td>Spherical function</td>
<td>0.000</td>
<td>40</td>
</tr>
<tr>
<td>8</td>
<td>Griewank function</td>
<td>0.000</td>
<td>50</td>
</tr>
</tbody>
</table>

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:

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

- 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. Therefore, 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 (3.1 GHz) on a Linux workstation running Fedora 9. The system memory was 2GB DDR. The GPU used was an NVIDIA GeForce 8800 GS/PCI/SSE2 with 512MB of DDR memory. The GPU was running OpenGL 3.0 with an NVIDIA driver version of 180.37. 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 Solution values from three implementation modes – CPU-CPU, GPU-CPU, GPU-GPU

<table>
<thead>
<tr>
<th>CPU/GPU</th>
<th>Solution Accuracy (%)</th>
<th>Objective Function</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>Average</td>
</tr>
<tr>
<td>1 (CPU-CPU)</td>
<td>100.00%</td>
<td>-1.032</td>
</tr>
<tr>
<td>1 (GPU-CPU)</td>
<td>100.00%</td>
<td>-1.032</td>
</tr>
<tr>
<td>1 (GPU-GPU)</td>
<td>100.00%</td>
<td>-1.032</td>
</tr>
<tr>
<td>2 (CPU-CPU)</td>
<td>100.00%</td>
<td>0.000</td>
</tr>
<tr>
<td>2 (GPU-CPU)</td>
<td>100.00%</td>
<td>0.000</td>
</tr>
<tr>
<td>2 (GPU-GPU)</td>
<td>100.00%</td>
<td>0.000</td>
</tr>
<tr>
<td>3 (CPU-CPU)</td>
<td>94.29%</td>
<td>0.099</td>
</tr>
<tr>
<td>3 (GPU-CPU)</td>
<td>100.00%</td>
<td>0.195</td>
</tr>
<tr>
<td>3 (GPU-GPU)</td>
<td>91.43%</td>
<td>0.127</td>
</tr>
<tr>
<td>4 (CPU-CPU)</td>
<td>100.00%</td>
<td>0.000</td>
</tr>
<tr>
<td>4 (GPU-CPU)</td>
<td>100.00%</td>
<td>0.000</td>
</tr>
<tr>
<td>4 (GPU-GPU)</td>
<td>100.00%</td>
<td>0.000</td>
</tr>
<tr>
<td>5 (CPU-CPU)</td>
<td>100.00%</td>
<td>0.123</td>
</tr>
<tr>
<td>5 (GPU-CPU)</td>
<td>100.00%</td>
<td>0.004</td>
</tr>
<tr>
<td>5 (GPU-GPU)</td>
<td>100.00%</td>
<td>0.004</td>
</tr>
<tr>
<td>6 (CPU-CPU)</td>
<td>68.57%</td>
<td>0.704</td>
</tr>
<tr>
<td>6 (GPU-CPU)</td>
<td>82.86%</td>
<td>0.347</td>
</tr>
<tr>
<td>6 (GPU-GPU)</td>
<td>80.00%</td>
<td>0.274</td>
</tr>
<tr>
<td>7 (CPU-CPU)</td>
<td>100.00%</td>
<td>0.003</td>
</tr>
<tr>
<td>7 (GPU-CPU)</td>
<td>100.00%</td>
<td>0.000</td>
</tr>
<tr>
<td>7 (GPU-GPU)</td>
<td>100.00%</td>
<td>0.000</td>
</tr>
<tr>
<td>8 (CPU-CPU)</td>
<td>100.00%</td>
<td>0.004</td>
</tr>
<tr>
<td>8 (GPU-CPU)</td>
<td>100.00%</td>
<td>0.002</td>
</tr>
<tr>
<td>8 (GPU-GPU)</td>
<td>100.00%</td>
<td>0.002</td>
</tr>
</tbody>
</table>

Legend: 1 – Camelback 2D, 2 – Rosenbrock 5D, 3 – Dixon and Price function 15D, 4 – Ackley’s path function 20D, 5 – Levy function 25D, 6 – Sum of squares function 30D, 7 – Spherical function 40D, 8 – Griewank function 50D. (CPU-CPU) – Objective Function and Velocity Vector on CPU, (GPU-CPU) – Objective function on GPU and Velocity Vector on CPU, (GPU-GPU) – Objective function and Velocity Vector on GPU.

Table 3 provides a summary of results obtained from solving problems 1 – 8. 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. The table shows that the solution accuracies obtained from implementing fitness values and velocity vector on the GPU are comparable with solutions obtained from implementing the method on a CPU alone. In all the test cases, the resulting average solution values for GPU-GPU (i.e., Objective function value computed on the GPU and Velocity vector computed on the GPU) were either equal or better when compared to CPU only or the GPU-CPU only implementations. For example, on the 25 dimensional Levy Function (problem 5), the average solution value returned from GPU-GPU implementation was 0.000 as opposed to 0.123 on CPU only implementation. The standard deviation for this problem was 0.000 for GPU-GPU implementation as opposed to 0.026 for CPU only implementation. The GPU-CPU implementation resulted in a 0.004 solution value and a standard deviation of 0.000. This means that using the GPU-GPU implementation did not induce any floating-point precision errors and compromise the solution quality characteristics. A very low standard deviation suggests that GPUs can be capable co-processors for computations and do not have a major effect in the solution qualities.

Figure 6 Solution Accuracy charts for test problems 1 - 8

Figure 7 shows the solution accuracy charts for the test problems. Although the solution accuracy characteristics for the GPU-GPU implementation favor off-loading PSO computations to the GPU, the solution duration charts proved otherwise. It can be seen that all test cases regardless of whether it is two dimensional or 50 dimensional problem took substantially longer with GPU-GPU implementations when compared to CPU only or GPU-CPU implementations.

On almost all the test problems, the solution times for the GPU-CPU implementation was either equal to or less than the CPU implementation only. This means that off-loading objective function computations to the GPU have significant merit when compared to performing PSO completely on the CPU. For example, the GPU-CPU implementation for the 50 dimensional Griewank function (problem 8) resulted in a ~23% decrease in solution time when compared to CPU-CPU implementation. However, the GPU-GPU implementation resulted in a ~470% increase in solution times when compared to CPU-CPU implementation. Similarly, a 15% decrease in solution time is observed for GPU-CPU implementation for the 30 dimensional Sum of squares function (problem 6) when compared to CPU-CPU implementation. The GPU-GPU version had a ~320% increase in solution times.

Since the GPU computations take place simultaneously for all swarm members and design variables simultaneously instead of serial manner on a CPU, it is logical to assume that GPUs will result in substantially lower solution times. However, the reason for a significant impact on the results have been attributed to the CPU-GPU bandwidth latency, a bottleneck caused by limitations of communication of information between a CPU and a GPU. While the GPU-CPU implementation required only the design variable information to be transferred to the GPU (i.e., a single texture) for objective function evaluations, the computation of the velocity required eight pieces of information, each on different textures. Not only so, but this information transfer between CPU and GPU takes place every iteration. For example, a 50-dimensional 500-particle problem that takes 250 iterations to converge will
require 50 design variables x 500 particles x 8 textures x 8 bytes x 250 = 400 Megabytes of texture input to the GPU, just for velocity vector calculations. For reading the computed velocity vector values calculated by the GPU back on the CPU, a total of 50 design variables x 500 particles x 1 texture x 8 bytes x 250 iterations = 50 Megabytes of information is transferred back to the CPU in total. This means a total of 450 Megabytes of information is shuffled between CPU and GPU. Even though the hardware parallelism of the GPUs can accommodate more design variables and swarm members, it will result in a higher amount of data transfer. As such, problems ranging from two dimensions through 50 dimensions resulted in substantially higher solution times when the velocity vector computations are performed on the GPUs.

**Figure 7** Solution Duration charts for test problems 1 - 8

To ascertain whether it is the number of texture reads/writes or the amount of data that is being uploaded and downloaded from the GPU that was causing the bandwidth bottleneck, a second approach for handing textures was implemented. The results depicted in Table 3, Figure 6 and 7 used eight GPU textures of GL_LUMINANCE format. The second approach consisted of converting GL_LUMINANCE textures to GL_RGBA that can take four channels of data per texture. This means that just two textures would be sufficient to handle GPU computations instead of the regular eight. Since each GL_RGBA texture can accommodate four channels and velocity vector requires eight input values, two input textures of GL_RGBA were used. Since the objective function calculations were using GL_LUMINANCE format regardless, this required creation of two frame buffer objects for the second approach.

Figure 8 shows the results from GL_RGBA texture format implementation compared against the regular GL_LUMINANCE textures. The plot shows that there is no change in solution times in spite of using lesser number of textures. This behavior was consistent over all the test problems. These results led to deduce that even though GPUs are extremely agile in performing computations, they do come with certain hardware limitations that inhibit the utilization of the full potential of GPUs for computations such as the velocity vector in PSO. The results suggest that there is no apparent benefit of GPUs over CPUs for computing the velocity vector. However, there is a substantial merit for computing objective function values on the GPU and the remaining operations on the CPU as evident from the GPU-CPU results.
V. Conclusion and Future work

This paper presents a method for implementing the velocity vector computations for PSO with digital pheromones on GPUs. Three different modes of implementation—CPU-CPU, GPU-CPU, and GPU-GPU—were explored and the results were presented. Although the GPU implementation showed absolutely no compromise in the solution accuracy and standard deviation characteristics, the velocity vector calculations portrayed that the attempts were quite ambitious and the cost-to-benefit was quite high when compared to CPU alone implementations. Although this preliminary implementation requires further careful investigation to avoid any programmatical inconsistencies, the general trend showed that objective functions alone will provide the best benefit when implementing PSO on GPUs. Current generation GPUs are incapable of generating high-quality random numbers, a feature if available could be critical for realizing manifold improvement in the developed method in terms of solution time.

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 developing robust constrained methods suitable for PSO, and developing methods for distributed parallel computing with CPUs and GPUs to rake further gains in computational efficiency in PSO.

References


18 American Institute of Aeronautics and Astronautics


