A Dynamic Cooperative Hybrid MPSO+GA on Hybrid CPU+GPU fused Multicore

Wayne Franz, Parimala Thulasiraman
Department of Computer Science
University of Manitoba
Winnipeg, Manitoba, Canada
umfranz@cs.umanitoba.ca, {tulsi, thulasir }@cs.umanitoba.ca

***Abstract*—Todays multi-core architectures with accelerators provide tremendous compute power. Population-based meta-heuristic algorithms have proven particularly amenable to single instruction multiple data (SIMD)-style parallelization due to the fine-grained parallelism provided by these algorithms. While SIMD hardware allows one to run large scale simulations, obtaining better solution quality often requires a more thoughtful reorganization of the search technique itself. In this paper, we design a hybrid heuristic algorithm that dynamically alternates between Multi-Swarm Particle Swarm Optimization (MPSO) and Genetic Algorithm (GA) to improve solution quality. We parallelize the hybrid algorithm on a hybrid multicore computer, accelerated processing unit (APU) to improve performance. We take advantage of the close coupling the APU provides between CPU and GPU devices. Our hybrid algorithm results indicate an improvement in average solution quality over Multi-Swarm PSO across a set of standard mathematical optimization functions. We study the effect and performance of switching between CPU and GPU devices.**

I. Introduction

Meta-heuristic algorithms are often used to approximate solutions to problems which are normally very time-consuming or difficult to solve exactly. These algorithms have become popular in a number of fields, including science, engineering, and NP-hard combinatorial optimization [18], [8], [1], [27]. In this work, we study population-based meta-heuristics, which maintain a large group of candidate solutions and iteratively adjust them according to a set of predefined update equations. This allows the algorithms to perform a semi-directed search of the solution space of an optimization problem.

Meta-heuristic algorithms are susceptible to becoming "stuck" in locally optimal solutions. If these local optima do not represent the global best solution to the problem, the algorithm returns a very inaccurate result. As a result, one of the primary goals in meta-heuristic research is improving the *robustness* (average performance across a range of problem types) of algorithms. To date, work has demonstrated that single algorithms have particular difficulty with high dimensional solution spaces [22], [25]. Improved results are usually obtained by algorithms that introduce a local search technique or combine multiple different algorithmic approaches [11], [28]. In the case of the latter, increasing solution quality requires organizing and controlling multiple algorithmic techniques in such a way that the candidate solutions adhere to a controlled convergence rate as they approach the global optimum.

Meta-heuristic algorithms with the tremendous amount of parallelism are suitable for implementation on discrete GPU platforms. Chen [2] proposed a crossover strategy using particles personal best positions. This eliminates the need for a mechanism to select fitter particles for crossover. Every given number of iterations, particles are reset to their personal best positions in order to increase the exploitation of the algorithm. At the same time, crossover is performed by placing a child particle at the midpoint between the personal best position of the current particle and the personal best position of a randomly chosen particle. After tuning the algorithm, Chen observed significant increases in solution quality (as high as 40%) over standard PSO for a set of standard multi-modal benchmark functions.

Zhou and Tan [29], [30] constructed a GPU-based parallel PSO algorithm with a mutation component. Mutation was triggered when the health of the swarm declined below a predefined threshold. A swarms health was defined by the percentage of particles that attained a new personal best fitness on each iteration. The algorithm yielded an increase in solution quality over standard PSO, in addition to a significant parallel speedup. However, the authors noted that it was necessary to adjust the strength of the mutation operator to fit the characteristics of the objective function.

Shi *et al.* [20], implemented two hybrid techniques. The first, a PSO-GA-parallel-hybrid evolutionary algorithm (PGPHEA), maintains a single GA population and a single PSO swarm in parallel, pausing after a designated number of iterations to exchange particles and chromosomes between them. The second, a PSO-GA-series-hybrid evolutionary algorithm (PGSHEA), runs n PSO swarms at once. After a set number of iterations, the best particles from each swarm are selected and transformed into chromosomes for a new GA population. The GA is run for a set number of iterations, at which point the chromosomes are transferred back to their original PSO swarms, and the process repeats. The authors observed that PGPHEA and PGSHEA often produced higher solution quality than standard PSO across a set of constrained and unconstrained mathematical optimization problems. However, they also noted that the GA slowed down the hybrid algorithms, and often required more time to find an optimal solution than standard PSO.

Recently, to better utilize and take advantage of the het-

erogeneity of *both* CPU and GPU machines to maximize performance there are some work in designing hybrid parallel algorithms on these architectures [24], [12]. There are two approaches: co-operative (CPU does the pre-processing step before offloading to GPU) or concurrent (the work is subdivided between CPU and GPU and they work concurrently). To circumvent the latencies (especially memory and communication bandwidth) posed between multicores and GPU, the AMD APU [4], [5], [16] is proposed as a viable architecture.

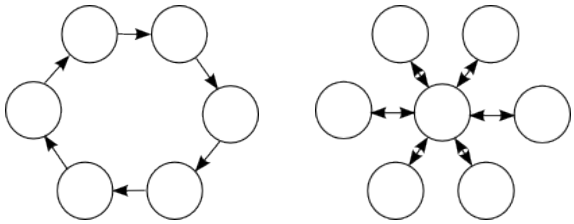
Talbi et al. [26] proposed hybrid heuristic evolutionary algorithm on GPU using co-operative approach. Compared to serial implementation, their algorithm produced 50x speedup. Vidal et al. [17], proposed a hybrid heuristic algorithm that does not use the CPU as a controller of tasks to GPU alone. While genetic algorithm with hill climbing is executed on the CPU in OpenMP, the systolic neighborhood algorithm is run on the GPU. The hybrid algorithm improved solution quality and produced 365x speedup on combinatorial optimization problems.

In Franz and Thulasiraman [13], we studied two parallel PSO-GA hybrid approaches on the accelerated processing unit (APU), fused CPU-GPU architecture. The first, multi-swarm PSO with mutation, crossover, and selection (MPSO-MCS) integrated the three GA operators into the update phase of PSO. Mutation was applied dynamically in response to changes in the quality of the solutions in a swarm, while selection and crossover were applied at regular intervals. Multiple swarms were arranged in a ring topology that allowed them to periodically exchange particles with their neighbours. On an exchange, the best particles from each swarm replaced the worst in the neighbouring swarm to the right. The second approach, MPSO with GA-initialization (MPSO-GA), used a GA to initialize the populations for a small number of iterations, then switched to MPSO-MCS for the remainder of the run. This is a *static* approach. The initial GA portion of the algorithm was executed on the CPU, while the MPSO-MCS phase ran on the GPU. Although this technique appeared advantageous early in the run, the authors discovered that the average final solution quality was not significantly different from that of MPSO-MCS.

To address the solution quality, in [14], we constructed a composite algorithm in which some swarms behaved like a GA, and others like PSO. We implemented such an algorithm and tested it in different configurations using the ring and star topologies (shown in figure 1 below). In the ring topology, each swarm is connected to a single immediate neighbour in a clockwise fashion. On a swap, each swarm sends its best k particles to its neighbour. The neighbour accepts these k best particles and uses them to overwrite its own k worst particles.

In the star topology, each of the outer swarms is connected (via a bi-directional link) to the central swarm. On a swap, the central swarm sends its k best particles to each outer swarm. The outer swarms accept these particles, and use them to overwrite the k worst particles in their own populations. At the same time, each of the outer swarms donates $\frac{k}{s-1}$ (where s is the total number of swarms) of its best particles to the central swarm. Collectively, the central swarm accepts k best particles from the outer swarms, and uses them to overwrite its k worst particles.

We showed that the topological communication structure of the populations impacted the solution quality of the hybrid algorithm.



(a) Ring (b) Star

Fig. 1: Swarm Topologies

In this paper, we improve the quality further by using a *dynamic* approach. We develop a hybrid heuristic algorithm that dynamically alternates between MPSO-MCS and GA during execution. The swarms are organized in a ring communication topology since it is better suited for parallelization than the centralized star topology. The algorithm monitors the convergence of the swarms and triggers the switch between algorithms dynamically when the average fitness ceases to improve.

In addition, in spite of the APU's low latency interconnect, passing control from CPU to GPU was found to cause a synchronization delay significant enough that it proved more efficient to run both portions of the algorithm on the GPU [13]. Therefore, we propose a co-operative approach by using the CPU to evaluate the global fitness function and implementing both parallel algorithms on the GPU in the Open Computing Language (OpenCL) [15]. This allows us to take advantage of the close coupling the APU provides between the CPU and GPU devices.

II. APU and OpenCL

The *Accelerated Processing Unit* (APU) is an AMD architecture which fuses together a CPU and GPU on a single die. This layout allows the devices to share a single memory space. On discrete GPU systems, data transfers between the host and the GPU must travel across a PCI Express bus, which can cause delays. By sharing a global memory space, transfers may be eliminated altogether because one device may simply pass a pointer to the other.

The APU we use for our experimentation is an AMD Northern Islands GPU. In this device, hardware is organized in a tree structure, with parallelism at each level [7]. The GPU consists of multiple processors called *compute units* (CUs). Our device has five. Each CU is really a composite processor (*streaming multiprocessor*) that contains a number of *processing elements* (PEs). All 16 PEs under the same CU execute the same instruction in lock-step on different data elements. Different CUs also execute the same instruction, but they may not always be completely synchronized.

The Open Computing Language (OpenCL) [15] is a C-like parallel language developed with the intention of providing a standard for heterogeneous systems [6]. OpenCL refers to threads as *work-items*. These are further bundled into *workgroups*. Only work-items within the same workgroup can synchronize or share information using local memory. It also provides *vector data types* that pack multiple primitive types into a single structure. Using these types can greatly assist the compiler's ability to pack instructions into the Very Large Instruction Word (VLIW) format necessary for execution on the PEs. OpenCL also allows us to run kernels written for the GPU on the CPU [7]. When running such kernels on the CPU, compute units map to processor cores, vector instructions use SSE/AVX, and local memory is emulated using CPU cache.

III. Dynamic Hybrid-Heuristic Algorithm

In this section, we briefly describe the two algorithms, MPSO-MCS and GA, we use to develop our hybrid heuristic algorithm.

A. MPSO-MCS

MPSO-MCS [13] as is a multi-swarm PSO that integrates the three GA operators, mutation, crossover, and selection into the update phase of PSO. The particles are arranged as multiple independent swarms in a ring topology. Every e iterations, the y best particles from each swarm replace the y worst particles of its neighbour to the right.

Each particle k maintains three pieces of information: a position vector X_k, a personal-best position vector $\acute{X}_k$, and a velocity vector V_k. Each swarm j also maintains the position of the best solution seen so far by any particle, $\hat{X}_j$. Particle position is fed into a *fitness function*, which returns particle fitness, f_k, a measure of the quality of the candidate solution. We also maintain $\acute{f}_k$ and $\hat{f}_j$, the particle-best and swarm-best fitnesses, respectively. PSO operates iteratively. Particle velocity is modified at each iteration i according to the following equation [21]:

$$V_k^{i+1} = \omega * V_k^i + c_1 * R_1 * (\acute{X}_k - X_k^i) + c_2 * R_2 * (\hat{X}_j - X_k^i) \quad (1)$$

Positions are then updated using:

$$X_k^{i+1} = X_k^i + V_k^{i+1} \quad (2)$$

Here, R_1 and R_2 are vectors of random numbers between 0 and 1 (selected from a uniform distribution), while c_1, c_2, and ω are user-defined weighting parameters used to balance the forces exerted by each of the three terms in equation 1.

Pseudo-code for a sequential version of this MPSO algorithm is shown in Algorithm 1. This code uses a simple ring topology in which exchanges happen in a clockwise-fashion around the structure.

Algorithm 1 Sequential MPSO Algorithm

```
1:  Initialize particle positions and velocities
2:  for iteration i = 0 to n − 1 do
3:    for swarm j = 0 to s − 1 do
4:      for particle k = 0 to p − 1 do
5:        Calculate f_k^i from X_k^i
6:        Set V_k^{i+1} using (1)
7:        Set X_k^{i+1} using (2)
8:        Update best values (Ẋ_k^{i+1}, X̂_j^{i+1}, ḟ_k^{i+1}, f̂_k^{i+1}) if
          necessary
9:        if i mod e = 0 and i > 0 then
10:         Overwrite y worst particles in swarm (j +
            1) mod s with y best in swarm j
11:       end if
12:     end for
13:   end for
14: end for
```

B. Genetic Algorithm

Genetic algorithm operates by iteratively applying three operators to the population of chromosomes: crossover, mutation, and selection. The evolutionary process is repeated until a stopping criteria is met. In this paper, we will refer to populations and chromosomes using the PSO terms "swarm" and "particle". Since our candidate solutions for GA and PSO use the same representation (a simple array of floating point values), both sets of terms are effectively equivalent.

We use tournament selection to randomly pick groups of chromosomes (of the same size) from the population. In each group, the individual with the best fitness is selected for crossover and each group is independently executed in parallel. We use simple *single point crossover* to recombine the parent genes. Both parent chromosomes are split at a random index. Recombination generates two new children by exchanging the second halves of the segments.

Mutation makes small random changes to the genes in chromosomes, with a predefined probability. We use a simple *uniform mutation* strategy. This strategy allows mutation to be omni-directional, as the magnitude of the change that is applied to each gene may be either negative or positive. Each gene in an individual is either mutated or not according to a user-defined probability $R \in [0, 1]$.

A high level overview of a GA is shown in Algorithm 2. The *threshold* and *max_iters* variables are user-defined parameters that are assumed to have been set at the outset of the algorithm. Line 1 initializes each chromosome i_k, $(0 \leq k < n)$ in the population P in a random fashion. These chromosomes then undergo crossover (with selection) and mutation in the while loop. The *fitness* function accepts an chromosome as an argument and returns a number indicating its quality. In Algorithm 2, these fitnesses are collectively referred to as F. The stopping criteria (line 10) assumes that we are trying to minimize the value obtained from the fitness function.

In order to dynamically switch between MPSO-MCS and GA algorithms, we use an additional heuristic to detect when stagnation (swarms becoming trapped in local optima) is occurring. When all of the swarms in the system do not attain a better global best, $gbest$, fitness position for a given number of consecutive iterations, we switch algorithms.

Algorithm 2 Sequential Genetic Algorithm

$P = \{rand(i_0), rand(i_1), ..., rand(i_{n-1})\}$
$done = False$
$best\_index = -1$
$i = 0$
while $i < max\_iters$ and not $done$ **do**
  $P' = crossover(P)$
  $P' = mutate(P')$
  $F' = \{fitness(i_0), fitness(i_1), ..., fitness(i_{n-1})\}$
  $P = P'$
  $F = F'$
  $best\_index = best\_fitness\_index(F)$
  $done = fitness(P(i_{best\_index})) \leq threshold$
  $i = i + 1$
end while
return $best\_index$

IV. Problem Set

To evaluate the hybrid algorithm, we use the benchmark functions presented in the CEC 2010 Special Session and Competition on Large-Scale Global Optimization [23], described below. These functions represent a wide range of problem characteristics and allow us to measure the robustness of the changes we introduce through hybridization.

Let $\mathbf{x} = (x_1, x_2, ..., x_d)$ be a particle position vector. Let $\mathbf{o} = (o_1, o_2, ..., o_d)$ be the globally optimal position. Define $\mathbf{z} = \mathbf{x} - \mathbf{o}$, the shifted global optimum. Then Table I shows the benchmark functions and their properties. All functions are minimization problems, with the global optimum positioned at $\mathbf{o}$.

The parameter m controls the degree of non-separability for the composite functions [23]. We set $m = 4$ for this work. This provides enough non-separability to afford a challenge for the multi-swarm algorithm, but not so much that the global best position is unattainable.

V. Parallel Dynamic Hybrid Heuristic on APU

We evaluate the heuristic on the host (CPU) side after each iteration, by mapping a pointer into the swarm-health buffer (described below) in the GPU address space. This allows us to examine the heuristic on the host side without any in-memory copying. We use synchronization to ensure memory is consistent when this pointer is mapped (we must stop the GPU, read it's memory with the CPU, and then decide whether to continue based on the output of the heuristic).

In order to describe our parallel algorithm, we define the following symbols: Let s be the number of swarms in the system, p the number of particles per swarm, and d be the number of dimensions in the solution space. We use the counter-based parallel PRNG that is well-suited for the GPU [19] that generates 2^{64} parallel streams of random values, each with a period of 2^{128}.

We now describe the various steps in the algorithm and its implementation on the GPU. We introduce two new kernels (Mutation Restoration Kernel and Crossover and Mutation Kernel) for the dynamic switching algorithm not existing in the iterative, static version [13]. Due to space limitations we do not provide the full implementation details of all the kernels here.

Function	Definition	Modality	Separability	Rotation
F_1	$F_{elliptic}(\mathbf{z})$	Uni	Separable	None
F_2	$F_{rastrigin}(\mathbf{z})$	Multi	Separable	None
F_3	$F_{ackley}(\mathbf{z})$	Multi	Separable	None
F_4	$F_{rot\_elliptic}[\mathbf{z}(P_1 : P_m)] * 10^6 + F_{elliptic}[\mathbf{z}(P_{m+1} : P_d)]$	Uni	Single-group m-nonseparable	Single-group m-rotated
F_5	$F_{rot\_rastrigin}[\mathbf{z}(P_1 : P_m)] * 10^6 + F_{rastrigin}[\mathbf{z}(P_{m+1} : P_d)]$	Multi	Single-group m-nonseparable	Single-group m-rotated
F_6	$F_{rot\_ackley}[\mathbf{z}(P_1 : P_m)] * 10^6 + F_{ackley}[\mathbf{z}(P_{m+1} : P_d)]$	Multi	Single-group m-nonseparable	Single-group m-rotated
F_7	$F_{schwefel}[\mathbf{z}(P_1 : P_m)] * 10^6 + F_{sphere}[\mathbf{z}(P_{m+1} : P_d)]$	Uni	Single-group m-nonseparable	None
F_8	$F_{rosenbrock}[\mathbf{z}(P_1 : P_m)] * 10^6 + F_{sphere}[\mathbf{z}(P_{m+1} : P_d)]$	Multi	Single-group m-nonseparable	None
F_9	$\sum_{k=1}^{\frac{d}{2m}} F_{rot\_elliptic}[\mathbf{z}(P_{(k-1)*m+1} : P_{k*m})] * 10^6 + F_{elliptic}[\mathbf{z}(P_{\frac{d}{2}+1} : P_d)]$	Uni	$\frac{d}{2m}$-group m-nonseparable	$\frac{d}{2m}$-group m-rotated
F_{10}	$\sum_{k=1}^{\frac{d}{2m}} F_{rot\_rastrigin}[\mathbf{z}(P_{(k-1)*m+1} : P_{k*m})] * 10^6 + F_{rastrigin}[\mathbf{z}(P_{\frac{d}{2}+1} : P_d)]$	Multi	$\frac{d}{2m}$-group m-nonseparable	$\frac{d}{2m}$-group m-rotated
F_{11}	$\sum_{k=1}^{\frac{d}{2m}} F_{rot\_ackley}[\mathbf{z}(P_{(k-1)*m+1} : P_{k*m})] * 10^6 + F_{ackley}[\mathbf{z}(P_{\frac{d}{2}+1} : P_d)]$	Multi	$\frac{d}{2m}$-group m-nonseparable	$\frac{d}{2m}$-group m-rotated
F_{12}	$\sum_{k=1}^{\frac{d}{2m}} F_{schwefel}[\mathbf{z}(P_{(k-1)*m+1} : P_{k*m})] * 10^6 + F_{sphere}[\mathbf{z}(P_{\frac{d}{2}+1} : P_d)]$	Uni	$\frac{d}{2m}$-group m-nonseparable	None
F_{13}	$\sum_{k=1}^{\frac{d}{2m}} F_{rosenbrock}[\mathbf{z}(P_{(k-1)*m+1} : P_{k*m})] * 10^6 + F_{sphere}[\mathbf{z}(P_{\frac{d}{2}+1} : P_d)]$	Multi	$\frac{d}{2m}$-group m-nonseparable	None
F_{14}	$\sum_{k=1}^{\frac{d}{m}} F_{rot\_elliptic}[\mathbf{z}(P_{(k-1)*m+1} : P_{k*m})]$	Uni	$\frac{d}{m}$-group m-nonseparable	$\frac{d}{m}$-group m-rotated
F_{15}	$\sum_{k=1}^{\frac{d}{m}} F_{rot\_rastrigin}[\mathbf{z}(P_{(k-1)*m+1} : P_{k*m})]$	Multi	$\frac{d}{m}$-group m-nonseparable	$\frac{d}{m}$-group m-rotated
F_{16}	$\sum_{k=1}^{\frac{d}{m}} F_{rot\_ackley}[\mathbf{z}(P_{(k-1)*m+1} : P_{k*m})]$	Multi	$\frac{d}{m}$-group m-nonseparable	$\frac{d}{m}$-group m-rotated
F_{17}	$\sum_{k=1}^{\frac{d}{m}} F_{schwefel}[\mathbf{z}(P_{(k-1)*m+1} : P_{k*m})]$	Uni	$\frac{d}{m}$-group m-nonseparable	None
F_{18}	$\sum_{k=1}^{\frac{d}{m}} F_{rosenbrock}[\mathbf{z}(P_{(k-1)*m+1} : P_{k*m})]$	Multi	$\frac{d}{m}$-group m-nonseparable	None
F_{19}	$F_{schwefel}(\mathbf{z})$	Uni	Fully nonseparable	None
F_{20}	$F_{rosenbrock}(\mathbf{z})$	Multi	Fully nonseparable	None

TABLE I: Benchmark functions and their properties [23].

A. Buffer Initialization

We maintain buffers to store the positions, velocities, fitnesses, personal-best positions, personal-best fitnesses and swarm-best positions, allocated in GPU global memory. Within the kernels, we may periodically shift portions of them to faster local memory to use as a scratch space, or to constant memory to improve single-element read access by multiple threads.

The permuted benchmark functions require a permutation P to shuffle the elements in the position vector. P is a permutation of the sequence of all possible position indices: $(0\ 1\ ...\ (d-1))$. We generate P on the host using Durstenfeld's version [9] of the Fisher-Yates shuffle algorithm and store it in a GPU-side global memory buffer. Similarly, we create an orthogonal matrix M for the rotated benchmark functions. The $m \times m$ matrix will be multiplied by the position vector to rotate the function. Our matrix generation code is adapted from a Java sample provided on the CEC website [3]. This code creates a randomly initialized matrix and applies the Gram-Schmidt process to make it orthogonal.

To improve performance, we store the matrix in column-major order to allow the elements to be accessed contiguously, to read chunks of four elements at a time and push them straight through the processing elements four ALUs. We generate the orthogonal matrix on the host in a host-resident, GPU-accessible buffer. Next, we launch a GPU kernel that reads this buffer and writes it to a GPU-resident buffer in column-major order (matrix transpose).

We maintain a swarm health buffer to record the number of consecutive iterations that each swarm has been unhealthy. This buffer is an array of size s that is stored in global memory. The dynamic switching algorithm also requires a global memory buffer of size s to track the number of consecutive iterations on which stagnation has occurred for each swarm.

B. Particle Initialization Kernel

This kernel uses one work-item per particle dimension ($s*p*d$ work-items) to initialize the X and V vectors for each particle. X is randomly initialized in the range $[-x_{max}, x_{max}]$ (see [23]). We initialize velocities to zero [10], and limit them to the range $[-x_{max}, x_{max}]$ - this clamping is done in the update position and velocity kernel. Since the work-item-to-data mapping is one-to-one, it is easy to use four-way vector data types in this kernel, reducing the number of required work-items by a factor of four.

C. Update Fitness Kernel

This kernel is our fitness function. Since our functions contain summations (most with independent terms), it is advantageous to assign multiple threads to cooperatively compute the fitness of each particle. For complex functions ($F_4 - F_{20}$), multiplying m components of the position vector by the $(m \times m)$ orthogonal matrix is done using a minimum of $m/4$ threads (functions that require a permutation will have access to more), and $m^2/2$ local memory space. We use $d/4$ threads per particle, with each thread handling four dimensions. The execution of the kernel proceeds in three phases. In phase 1, all threads read four position values from global memory and compute the values of the corresponding four terms of the summation to fully utilize the 128-bit bus and perform simultaneous arithmetic operations on all four values allowed by the VLIW configuration. In phase 2, we combine all of the partial results (for each particle) from phase 1 into a single fitness value to local memory and perform a reduction using the $d/4$ threads assigned to the same particle. Phase 3 sums the two-element chunks that resulted from the reduction and writes these final values back to the global memory fitness buffer.

Note, that we cannot split particles across workgroups because threads operating on a single particle require local memory to communicate. We launch $\left(\frac{256}{(d/4)} - (256 \bmod (d/4))\right) * d/4$ threads per workgroup, where 256 is the maximum size of a GPU workgroup for our device.

D. Update Bests Kernel

We update the particle-best and swarm-best fitnesses and positions, launching one work-item per particle. The kernel operates in two phases.

The first compares $\acute{f}_k^{i-1}$ and f_k^i for each particle k and writes to local memory. If an update is required, each work-item overwrites $\acute{f}_k^{i-1}$ and $\acute{X}_k^{i-1}$ in their global memory arrays. The second phase performs a parallel reduction to find $\hat{f}_j^i$ for each swarm j and update it, if necessary. We reuse the local buffer to perform a reduction, skipping over any unneeded values. Finally, work-items cooperate to update $\hat{X}_j^i$.

E. Update Position/Velocity

Since Equation 1 allows each dimension to be calculated independently, we launch a full $s*p*d/4$ work-items. Note that in Equation 1, the same swarm-best positions ($\hat{X}_j^i$) are read by all $p * d/4$ work-items in a swarm j. Therefore, we place them in constant memory to take advantage of broadcasting.

The mutation operation is implemented here. This kernel uses one thread per dimension. We compute the new velocity and position using equations (1) and (2). Next, with probability β, we perform mutation on both position and velocity. The resulting vectors are copied back to global memory to overwrite the previous iteration's values. Implementing the new mutation restoration mechanism also requires making several changes in this kernel. We create new buffers to store the state of the particles before mutation so that we can restore them in the event that mutation is detrimental. Before we perform the updates described above, we copy the previous iteration's fitnesses, velocities and positions to these new buffers.

F. Find Best/Worst Particles

In this kernel, we determine the indices of the particles with the y best and y worst fitnesses in each swarm. This information is stored in global memory buffers so that the exchange can be done by the swap kernel (below). We map one work-item to each particle. First, we copy the fitness data from global memory to a local buffer of size p (buf1). Next, we use two more local buffers of size p (buf2, buf3) to perform a parallel reduction to find the best and worst particles. Throughout this process, we maintain the particle index of each fitness value by keeping two auxiliary local memory arrays. Each time a value is moved in the reduction, the move is mirrored in the corresponding auxiliary buffer. After the reduction, we overwrite the best and worst values in buf1 using a negative value, and omit them from future reductions. Finally, we write the indices to global memory, reset all of the buffers (copying original fitness data from buf1 to avoid another global memory access), and restart the reduction to find the second-best and second-worst particles. This process is repeated y times. In total, five buffers of size p are used.

G. Swap Particles Kernel

This kernel performs the actual particle exchange between swarms, using the ring topology. We launch one work-item for each dimension of every particle to be exchanged ($s*y*d$ work-items for the ring topology). A portion of the best particles in each swarm overwrite the worst particles in the next swarm. Specifically, X, V, $\hat{X}$, and $\hat{f}$ are overwritten. Current fitnesses will be recalculated on the next iteration before they are needed. In order to prevent multiple particles from selecting the same partner to cross with, a random permutation of particle indices, $P = (0\ 1\ ...\ (p-1))$ is created. Each particle k crosses with $P(k)$. We implemented

an intra-swarm method that crosses particles with with $pbest$ positions from other particles in their own swarm. We use the ring topology to cross particles in swarm j with others in swarm $(j+1) \bmod s$.

H. Mutation Restoration Kernel

This is a new kernel that is introduced immediately following the update fitness phase of the algorithm. Its purpose is to restore the fitnesses, velocities and positions of unhealthy swarms if the mutation performed on the previous iteration was detrimental. Here, we map one thread to every four dimensions and launch $s * p * d/4$ threads. First, each thread checks to see if the swarm it corresponds to is unhealthy (indicating that mutation was triggered on the last iteration). If so, it compares the current fitness of the particle to the pre-mutation fitness that was saved by the update position and velocity kernel on the last iteration. If the new fitness is worse, we restore the old position and velocity from their saved buffers. Finally, one thread per swarm resets the swarm health buffer element to zero.

I. Crossover and Mutation Kernel

We use tournament selection with a group size of t (with replacement). We launch $\max(d/4, t/4)$ threads per particle. For each particle, $t/4$ threads randomly select t particles and perform a parallel reduction to find the one with the best fitness. A two-way vectorized approach executes this reduction using $t/2$ local memory.

Next, single-point crossover is performed with probability γ. After randomly choosing a crossover point (this information is shared by causing all threads operating on the same particle to use the same counter values), half of the threads for each particle read values from the left half of the first parent. The other half read values from the right half of the second parent. These reads are done in chunks of four to take advantage of our memory bus width. Finally, one thread combines the chunks in which the cross point lies.

Up to this point, all of our crossover chunks are stored in registers. Performing mutation and crossover in the same kernel allows us to avoid a write-back to global memory between the two operations. All threads apply uniform mutation to their chunks of values. Finally, we write the values back to the global memory position buffer.

The swapping and crossover operators both periodically overwrite particles. Since both crossover and swapping are applied every 100 iterations, their applications coincide on the same iteration and were found to interfere with each other. Since crossover is applied before swapping, some particles that are crossed may immediately be overwritten in the swap, before their fitness has been evaluated. Therefore we staggered the application of crossover by 50 iterations.

a) Alternating:: Finally, on the host side, we obtain a pointer to the stagnation buffer and check to see if all swarms have a value greater than our threshold number of iterations. If so, the alternate algorithm will begin on the following iteration. Otherwise, the current algorithm will continue. This host-device communication requires a synchronization barrier before each check. This ensures that the host does not check the buffer before the current algorithm's latest iteration (running on the GPU) completes.

VI. Results

The results for the dynamic hybrid algorithm are shown in Table II. The first two columns in the table show the function and average fitness across 20 runs (each run is allowed to continue for 400000 iterations). The "solve count" column displays the number of runs on which the optimal solution was found. For each run, we also record the iteration at which the system-wide best fitness stopped improving. The "Avg Solve Flatline" column displays this value, averaged across those runs in which the optimal solution was found (i.e the average iteration at which the optimal solution was found). The "Avg Fail Flatline" column shows this value averaged across those runs in which the optimal solution was not found (i.e. the average iteration at which system-wide stagnation occurred).

Using some simple empirical tests, we found an acceptable switch point for the algorithms near 1000 iterations. This means that in order for a switch to occur, the entire system must not attain a new global best, $gbest$, fitness value for a minimum of 1000 iterations.

Fcn	Avg Fitness		Solve Count	Avg Solve Flatline	Avg Fail Flatline
F_1	0.0000000000	± 0.0000000000	20	547.5	-
F_2	0.0000812054	± 0.0003539660	19	2268.421143	8150
F_3	0.0000009537	± 0.0000000000	0	-	3032.5
F_4	1487.1175537109	± 5305.7036132813	0	-	22375
F_5	0.0000000000	± 0.0000000000	20	2532.5	-
F_6	0.9536750913	± 0.0000001788	0	-	2597.5
F_7	0.0000000000	± 0.0000000000	20	617.5	-
F_8	0.0000763976	± 0.0003330093	19	5431.579102	2800
F_9	28.3636283875	±52.1113624573	0	-	39677.5
F_{10}	1.9836571217	± 1.4924528599	0	-	24967.5
F_{11}	0.0054430007	± 0.0237130206	0	-	820
F_{12}	0.0000000000	± 0.0000000000	20	605	-
F_{13}	1.0905876160	± 0.0000000961	0	-	15485
F_{14}	2975.0483398438	± 4049.8256835938	0	-	39975
F_{15}	8.4520912170	± 2.8283038139	0	-	31735
F_{16}	0.0920106918	± 0.4010486901	0	-	2270
F_{17}	0.0000000000	± 0.0000000000	20	820	-
F_{18}	4.8699765205	± 7.7608213425	0	-	37597.5
F_{19}	0.0000000000	± 0.0000000000	20	1437.5	-
F_{20}	0.5036863089	± 2.1955180168	1	18150	25155.26367

TABLE II: Dynamic switching: solution quality.

Fcn	Avg GA Runs	Avg MPSO Runs	Avg Total Time
F_1	14.95	15.55	39.029573
F_2	10.00	10.55	36.015223
F_3	15.45	16.10	37.671725
F_4	9.60	9.95	49.634658
F_5	13.90	14.45	46.022383
F_6	15.95	16.70	49.926449
F_7	14.80	15.20	41.003188
F_8	8.70	9.35	40.095976
F_9	0.00	1.00	77.386322
F_{10}	2.60	3.45	58.184283
F_{14}	0.00	1.00	72.416916
F_{15}	0.90	1.85	68.358946
F_{16}	18.25	18.55	68.942839
F_{17}	14.50	15.00	48.075199
F_{18}	0.10	1.10	48.109142
F_{19}	13.35	13.90	40.547843
F_{20}	3.40	4.20	37.690814

TABLE III: Dynamic switching: execution time.

In general, the dynamically-switching hybrid performed better than the static GA and MPSO-MCS algorithm in [13]. For functions $F_3, F_4, F_9, F_{11}, F_{13}, F_{14}$, and F_{18} this algorithm also either tied or outperformed the MPSO-MCS algorithm in terms of average solution quality.

Table III shows the cumulative amount of time for which each component of the dynamic switching algorithm (GA or MPSO-MCS) was executed, for each function. The synchronization barriers required at each iteration (since we are using the host to evaluate the switch heuristic) made this a time-consuming algorithm.

We observed several interesting phenomena when comparing the dynamic switching algorithm with the static approach: For the F_4 function, the dynamic switching algorithm achieved better solution quality than the static algorithm, but slightly poorer average solution quality than MPSO-MCS. Although the function was not fully solved in any of the three algorithms, we observed that both of the better performing algorithms (MPSO-MCS and dynamic switching) flat-lined much later in the run (about two times farther) than the static algorithm.

The average number of triggered mutation applications for the MPSO-MCS algorithm was 2481.489746 $\pm$ 4.815617, while the dynamic switching algorithm went through only 1975.179688 $\pm$ 384.816162 applications of mutation. The fact that the latter number is smaller seems to suggest that the swarms may be stagnating less often in the dynamic switching algorithm. However, the larger standard deviation would seem to indicate that this effect varies much more from run to run than in the MPSO-MCS algorithm.

The F_9 and F_{14} functions are elliptic-based functions. In the dynamic switching algorithm the GA was never triggered for these functions (note the high value of the average fail flatline) because the stagnation did not occur. This suggests that particles have not yet converged, and so switching was not really necessary.

A. Speedup

Table IV shows the average time required to execute standard MPSO using a quad-core CPU, the average time required using the GPU, and the resulting (relative) speedup. Due to the length of time required to execute a simulation of this size on the CPU, we limited the experiment to a representative set of the benchmark functions. We notice that the CPU execution times are higher than the GPU execution times.

Fcn	Avg CPU time	Avg GPU time	Speedup
F_1	379.3708	18.3397	20.69
F_4	514.4256	28.8616	17.82
F_9	742.7202	58.2920	12.74
F_{14}	896.3262	52.9344	16.93
F_{20}	304.1031	304.1031	16.87

TABLE IV: Relative Speedup.

This is because, it is important to remember that while OpenCL allows us to run the same code and kernels on both devices, the way in which the hardware executes this code differs significantly between them.

First, for the CPU, the maximum workgroup size is 1024, four times larger than the GPUs maximum of 256. Each core will execute a workgroup using a single thread that iteratively runs through each instruction across all work-items. In addition, the OpenCL compiler will attempt to pack instructions into groups of four in order to employ SSE instructions.

Second, the CPU does not provide local or constant memory, so the runtime emulates it using regular system memory, which is approximately two orders of magnitude slower. This means that kernels will receive more detriment than benefit from executing kernels containing local or constant memory-based optimization.

Third, in order to compensate for the large difference in the clock rates of the processor and the memory system, the CPU must rely on the cache hierarchy. GPU kernels are not coded with this in mind. When a CPU thread executes a workgroup, it loops across all work-items for a single instruction at a time. Therefore it is conceivable that the cache may become polluted with data from other work items before moving on to the next instruction.

We believe that this illustrates the importance of taking the hardware architecture into account when studying metaheuristic algorithms. As the architecture has a direct influence on the execution time, the degree to which the algorithm is suited to (or successfully adapted and optimized for) the architecture can significantly sway its cost-benefit ratio. There may exist some hybrid metaheuristic algorithms which would not be practical (from the perspective of cost-benefit ratio) when run on one particular architecture, but could be on others.

In spite of these factors, we believe that the speedups shown in the table are large enough to demonstrate a clear benefit to using the GPU at our swarm size and dimensionality.

VII. Solution Quality versus Execution Time

Metaheuristic algorithms' main utility is trading solution quality for execution time. Therefore in the event that modifications are made to improve the former, it is important to measure the latter in order to ensure that the balance between the two remains acceptable.

Table V presents a summary of the results for each of the algorithmic variations we have tested, MPSO-MCS, Static and Dynamic algorithms. The first column in the table was obtained by averaging the time taken across 20 runs. This was done independently for each function, and the results were summed. The second column presents the sum of the number of times the algorithm solved a function completely. The maximum (best) possible value here is 400 (20 runs $\times$ 20 functions). Finally, the third column presents the ratio between these two values (Time/Solve Count). This ratio allows us to see the net effect of the trade off between execution time and solution quality. Lower values indicate a more favorable cost-benefit ratio, while larger values indicate that at least one of the factors was negatively impacted to a larger degree. The results are ordered on this column, from best to worst.

The table shows compared to the static hybrid algorithm, the dynamic hybrid algorithm produced better solution quality

Alg	Avg Time (sec)	Avg Solve Counts	Ratio
MPSO-MCS	692.258	167	4.145
Static	783.613	139	5.638
Dynamic	1012.061	159	6.365

TABLE V: Sum of average execution time per algorithm.

(solve count) but with a significant increase in execution time. Unlike the static approach, the dynamic approach introduced switching between algorithms that incorporated synchronization between local and global memories, in order to maintain memory coherency. We also observed that the MPSO-MCS algorithm provides the highest solve count, with a moderate increase in execution time. This is because the algorithm is executed on the GPU only combining another algorithm's ideas (in this case genetic operators) into the heuristic.

VIII. Conclusion

In this paper, we developed a dynamically-switching parallel hybrid heuristic algorithm on the APU. We applied a co-operative approach in which the MPSO-MCS and GA were implemented on the GPU using the CPU to co-ordinate the tasks to the GPU. We observed the dynamic algorithm produced better solution quality than the static algorithm at the expense of execution time. Synchronization in unavoidable on the parallel machines. Combining two meta-heuristics together, such as MPSO-MCS and implementing the algorithm on the data parallel architecture such as the GPU maybe an alternate way of designing parallel algorithms. This is for future work.

References

[1] Cantú-Paz, E.: A Survey of Parallel Genetic Algorithms. Calculateurs Paralleles 10 (1998)

[2] Chen, S.: Particle Swarm Optimization with pbest Crossover. In: IEEE Congress on Evolutionary Computation. pp. 1–6. Brisbane, Australia (Jun 2012)

[3] Computation, N.I., of Science, A.L.U., of China, T.: Special Session on Large Scale Global Optimization: 2010 IEEE World Congress on Computational Intelligence. http://nical.ustc.edu.cn/cec10ss.php (accessed Feb. 19, 2014) (Nov 2009)

[4] Daga, M., Aji, A.M., Wu-chun Feng title = On the efficacy of a fused CPU+GPU processor (or APU) for parallel Computing, booktitle = Symp on Appl. Accelerators in HPC, m..J.y...:

[5] Daga, M., Nutter, M.: Exploiting coarse-grained parallelism in B+ tree searches on an APU. In: IEEE High Performance Computing, Networking, Storage and Analysis (2012)

[6] Devices, A.M.: OpenCL: The Future of Accelerated Application Performance Is Now. http://www.amd.com/us/Documents/FirePro_OpenCL_Whitepaper.pdf (Accessed Sept. 2012) (2011)

[7] Devices, A.M.: AMD Accelerated Parallel Processing OpenCL Programming Guide. http://developer.amd.com/download/AMD_Accelerated_Parallel_Processing_OpenCL_Programming_Guide.pdf (Accessed July 2014) (Nov 2013)

[8] Dorigo, M., Stützle, T.: The Ant Colony Optimization Metaheuristic: Algorithms, Applications, and Advances. In: Glover, F., Kochenberger, G.A. (eds.) Handbook of Metaheuristics, International Series in Operations Research & Management Science, vol. 57, pp. 250–285. Springer US (2003)

[9] Durstenfeld, R.: Algorithm 235: Random Permutation. Commununications of the ACM 7(7), 420 (Jul 1964)

[10] Engelbrecht, A.: Particle Swarm Optimization: Velocity Initialization. In: 2012 IEEE Congress on Evolutionary Computation. pp. 1–8. Brisbane, Australia (Jun 2012)

[11] Fatih Tasgetiren, M., Liang, Y.C., Sevkli, M., Gencyilmaz, G.: Particle Swarm Optimization and Differential Evolution for the Single Machine Total Weighted Tardiness Problem. International Journal of Production Research 44(22), 4737–4754 (2006)

[12] Feichtinger, C., Habich, J., Köstler, H., Rüde, U., Aoki, T.: Performance modeling and analysis of heterogeneous lattice boltzmann simulations on CPUGPU clusters. Parallel Computing 46, 1–13 (2015)

[13] Franz, W., Thulasiraman, P., Thulasiram, R.K.: Exploration/exploitation of a hybrid-enhanced MPSO-GA algorithm on a fused CPU-GPU architecture. Concurrency and Computation: Practice and Experience (2014)

[14] Franz, W., Thulasiraman, P.: Effect of communication topologies on hybrid evolutionary algorithms. In: Sixth World Congress on Nature and Biologically Inspired Computing. pp. 232–237. Porto, Portugal (2014)

[15] Group, K.O.W.: The OpenCL Specification (v1.2) (Nov 2012)

[16] Nilakant, K., Yoneki, E.: On the efficacy of APUs for heterogeneous graph computation. In: Fourth Workshop on Systems for Future Multicore Architectures. Amsterdam, Netherlands (2014)

[17] Pablo, V., Enrique, A., Francisco, L.: Solving optimization problems using a hybrid systolic search on GPU plus CPU. Soft Computing (Jan 2016)

[18] Poli, R.: Analysis of the Publications on the Applications of Particle Swarm Optimisation. Journal Artificial Evolution and Applications pp. 4:1–4:10 (Jan 2008)

[19] Salmon, J., Moraes, M., Dror, R., Shaw, D.: Parallel Random Numbers: as easy as 1, 2, 3. In: Proceedings of 2011 International Conference for High Performance Computing, Networking, Storage and Analysis. pp. 16:1–16:12. Seattle, WA, USA (Nov 2011)

[20] Shi, X.H., Lu, Y.H., Zhou, C.G., Lee, H.P., Lin, W.Z., Liang, Y.C.: Hybrid Evolutionary Algorithms Based on PSO and GA. In: The IEEE Congress on Evolutionary Computation. vol. 4, pp. 2393–2399. Canberra, Australia (Dec 2003)

[21] Shi, Y., Eberhart, R.: A Modified Particle Swarm Optimizer. In: Proceedings of the 1998 IEEE International Conference on Evolutionary Computation. pp. 69–73. Anchorage, AK, USA (May 1998)

[22] Sinha, A., Goldberg, D.E.: A survey of hybrid genetic and evolutionary algorithms. Tech. rep., University of Illinois at Urbana Champaign (2003)

[23] Tang, K., Li, X., Suganthan, P.N., Yang, Z., Weise, T.: Benchmark Functions for the CEC'2010 Special Session and Competition on Large-Scale Global Optimization. Tech. rep., University of Science and Technology of China (USTC), School of Computer Science and Technology, Nature Inspired Computation and Applications Laboratory (NICAL): Héféi, Ānhuī, China (2010)

[24] Teodore, G., Pan, T., Kurc, T.M., Kong, J., Cooper, L.A.D., Saltz, J.L.: Efficient irregular wavefront propagation algorithms on hybrid CPU-GPU machines. Parallel Computing 39, 189–211 (2013)

[25] Van, L.T., Nouredine, M., El-Ghazali, T.: Parallel hybrid evolutionary algorithms on GPU. In: Proceedings of the IEEE Congress Evolutionary Computation. pp. 1–8 (2010)

[26] Van, L.T., Nouredine, M., El-Ghazali, T.: GPU computing for parallel local search meta-heuristic algorithms. IEEE Transactions on Computers 62(1), 173 185 (2013)

[27] Yang, X.S., He, X.: Firefly Algorithm: Recent Advances and Applications. International Journal of Swarm Intelligence 1(1), 36–50 (Jan 2013)

[28] Yang, X.S.: Engineering Optimization: An Introduction with Metaheuristic Applications. Wiley Publishing, Hoboken, NJ, USA, 1st edn. (2010)

[29] Zhou, Y., Tan, Y.: GPU-Based Parallel Particle Swarm Optimization. In: IEEE Proceedings of the Eleventh Conference on Congress on Evolutionary Computation. pp. 1493–1500 (May 2009)

[30] Zhou, Y., Tan, Y.: Particle Swarm Optimization with Triggered Mutation and its Implementation based on GPU. In: ACM Proceedings of the 12th annual Conference on Genetic and Evolutionary Computation. pp. 1–8 (Jul 2010)