Optimizing Shape Design with Distributed Parallel Genetic Programming on GPUs

Simon Harding and W. Banzhaf

Abstract. Optimized shape design is used for such applications as wing design in aircraft, hull design in ships, and more generally rotor optimization in turbomachinery such as that of aircraft, ships, and wind turbines. We present work on optimized shape design using a technique from the area of Genetic Programming, self-modifying Cartesian Genetic Programming (SMCGP), to evolve shapes with specific criteria, such as minimized drag or maximized lift. This technique is well suited for a distributed parallel system to increase efficiency. Fitness evaluation of the genetic programming technique is accomplished through a custom implementation of a fluid dynamics solver running on graphics processing units (GPUs). Solving fluid dynamics systems is a computationally expensive task and requires optimization in order for the evolution to complete in a practical period of time. In this chapter, we shall describe both the SMCGP technique and the GPU fluid dynamics solver that together provide a robust and efficient shape design system.

1 Introduction

Optimized shape design (OSD) is a problem of optimal control theory, where the task is to find a shape that minimizes certain parameters while satisfying a set of constraints. In this chapter we describe an OSD technique that uses inspiration from biological evolution to design hydrodynamic shapes (such as wing surfaces) which meet certain criteria like the minimization of drag or the

Simon Harding
IDSIA (Istituto Dalle Molle di Studi sull'Intelligenza Artificiale), Switzerland
e-mail: simon@idsia.ch

Wolfgang Banzhaf
Memorial University of Newfoundland, Canada
e-mail: banzhaf@mun.ca

F.F. de Vega et al. (Eds.): Parallel Architectures & Bioinspired Algorithms, SCI 415, pp. 51–75.
springerlink.com

maximization of lift. The approach we shall consider is general and could be used to find good shapes for car bodies, aeroplane fuselages, boat hulls, etc. The technique uses a so-called distributed parallel evolutionary algorithm to optimize the solution, along with a general purpose parallel fluid dynamics solver to evaluate the shape parameters.

Evolutionary algorithms (EAs) have a long history of being used to generate designs for physical objects. In fact, one of the branches of this field, Evolutionary Strategies, started out with a problem for nozzle design [25, 27]. Generally, EAs mimic mechanisms of biological evolution (populations of solutions under mutation and recombination, using fitness evaluation to determine what is being promoted from one generation of solutions to the next) to solve optimization problems. In the area of design previous examples include electronic circuit design, furniture design, or the design of structural features of buildings, and aerodynamic shape design. Section 2 describes previous attempts to evolve aerodynamic shapes using similar approaches. Section 3 then introduces a genetic programming technique for the optimized shape design. Genetic Programming [14, 23] is another branch of EAs that has recently become more prominent due to its ability to adapt solutions to the complexity of a problem at hand.

Generally, the work flow of the method includes steps in which such designs need to be tested in a simulated environment, complete with models of physics. This will result in the assignment of solution quality to each of the individual solutions in the population, provided quality ("fitness" criteria, in EA speak) have been defined beforehand. Physical simulations, however, are notoriously expensive in terms of computation time - but with recent advances in Graphics Processing Units (GPUs) it is now possible to speed up these simulations and hence the fitness evaluation of solutions consisting of complicated objects within a complex physical system at relatively low cost.

Graphics processing units are a specific type of parallel many-core processing units. GPUs are cheap and ubiquitous, they are now present in almost all modern PCs and laptops to enhance performance. Originally designed for gaming and graphics processing, they have evolved into general purpose processing units with advantages for the solution of many types of scientific problems. Section 4 describes the hardware and software models of GPUs and their advantages for this type of problem.

Computational Fluid Dynamics (CFD) is an area of fluid mechanics that uses algorithms and numerical methods to solve fluid flow problems. Within CFD there are many different approaches, involving different solution methods. The choice of a solution method is largely dependent on the problem, on the context of the problem, and on what is required of a solution for later analysis (post-processing). Section 5 will describe a technique to simulate and evaluate general purpose fluid design environments using GPUs to increase efficiency.

2 Evolving Aerodynamic Shapes

Nature is full of examples of creatures that are adapted to operate with aerodynamic or hydrodynamic requirements. Wings, fins, streamlined bodies, textures to minimize turbulence, and feathers are all examples of this. Given the variety and efficiency of what nature produces it seems appropriate to use the same principles to solve man-made challenges.

Evolutionary algorithms have already been successfully applied to optimizing the design of objects that interact with a fluid environment. For example, in [22] genetic algorithms, another branch of EAs, are used to optimize the design of airfoils. Two-dimensional (2D) representations of airfoils were specified as a set of control points for B-Splines. The evolutionary algorithm adjusted these control points until a satisfactory arrangement was found. Fitness of candidate airfoils was determined by simulating the design and measuring variables such as pressure rise. Using a similar technique, nozzles for rocket engines have been optimized, too[3].

In [24], the authors applied a hybrid system of a genetic algorithm and a neural network to optimize the design of yacht keels. Here, the parameters for the keel design were optimized to reduce drag and maximize lift. In [4], airfoils were evolved, again using Bezier surfaces to define the sections. Recently, [1] built on this work employing a grid of computers to reduce the bottleneck of fitness evaluation.

Wing evolution has also been demonstrated using physical models, where the rotation of a number of connecting plates was altered and tested for lift in a wind tunnel [26]. Rechenberg has also used a similar method to examine other related systems such as the evolution of the wing tips of birds and nozzle designs.

Previous approaches have largely involved either optimizing parameters for known designs or using Bezier/Nurbs surfaces. This was well suited for genetic algorithms or evolutionary strategies, as it significantly reduced the search space. With such constrained representations, however, it is difficult to imagine how radically new designs could be produced or how these techniques could be expanded to evolve more complex, multi-component systems. In other words, as soon as more creativity in solutions is required, or one is prepared to test truly novel ideas, other techniques will need to be applied.

There have been several attempts to implement an evolutionary design of objects. The aim of the work described in this chapter is to allow for arbitrary structures to be evolved, and therefore the Bezier control point based representations mentioned above are not used. The requirements for this work include the ability to produce both 2D and 3D representations, single objects and non-connected designs, to produce vectorized objects that allow for distortion-free scaling and rotation, and the ability to produce curved/free form shapes. It is also envisaged that interesting designs would include concepts such as symmetry, repetition or repetition with variation. These

requirements suggest that a genetic programming approach in the context of a developmental system would be appropriate [17, 2].

3 Self-modifying Cartesian Genetic Programming (SMCGP)

Self-modifying Cartesian Genetic Programming (SMCGP) is a developmental version of Genetic Programming. In brief, SMCGP is a way of evolving computer programs that can change their own structure (and hence behaviour) at runtime. This method has been used for numerous applications, such as evolving digital circuits [9, 10, 6], finding algorithms that approximate physical constants [11, 12], discovering learning algorithms [8] and regression and classification [7].

As the name suggests, SMCGP is based on the Cartesian Genetic Programming (CGP) technique. In CGP, programs are encoded in a partially connected feed forward graph. (see [18]). The genotype encodes this graph, with each node represented as a function and connections to other nodes that this function connects to. The representation has a number of interesting features. For instance, not all of the nodes of a solution representation ("the genotype") need to be connected to the output node of the program, so there are nodes in the representation that have no effect on the output, a feature known in GP as "neutrality". This has been shown to be very useful [20] for the evolutionary process. Also, because the genotype encodes a graph, there can be reuse of nodes (revisiting of nodes is allowed), which makes the representation distinct from a classically tree-based GP representation.

Although CGP has been used in other developmental systems [19, 15], the programs that those approaches produced were not themselves developmental. SMCGP, on the other hand, was designed as an attempt to bring development into CGP so that CGP could be used as a general purpose developmental GP system.

The SMCGP representation is similar to CGP in some ways, but has extensions that allow it to exhibit self-modifying features. SMCGP genotypes are a linear string of nodes. Each node connects to two other nodes by way of a relative address, which states how many nodes back to connect. To prevent cycles, nodes can only connect to other nodes in one direction. Relative addressing allows entire sections of the graph to be moved, duplicated, deleted, etc, without breaking the reference structure, whilst allowing some sort of modularity.

In overview, each node in an SMCGP graph contains a number of elements:

- The computation function, represented in the genotype as an integer;
- A list of (relative) connection addresses, again represented as integers;
- A set of parameters, represented by 3 floating point numbers.

As with CGP, the number of nodes in the genotype is typically kept constant throughout an evolutionary run. However, this means care has to be taken to ensure that the genotype is large enough to store a possibly complex target program. Any kind of adjustment to the complexity would then come from the turning on and off of node execution paths through this graph which we shall explain next.

3.1 Executing a SMCGP Individual

SMCGP individuals are evaluated in a multi-step fashion, with the evolved program (the "phenotype") executed several times. An evolved program in SMCGP initially has the same structure as the genotype, which is supposed to represent it. The first step in producing the phenotype is to simply make a copy of the genotype and call it the initial phenotype. This graph is the 'working copy' of the program that will later be modified during further execution of nodes. Each time the program is executed, the phenotype graph is first run and then any self-modification operations encoded are invoked.

The graph is executed in the following manner: First, the node (or nodes) to use as output(s) are identified. This is done by parsing through the graph looking for nodes of type OUTPUT. Once a sufficient number of these nodes has been found, the various nodes that they connect to are identified using recursion. In case that there are not enough output nodes found in this way, the last n nodes in the graph are rededicated as output nodes, where n means the number of outputs required. If there are not even enough nodes to satisfy this condition, execution is aborted and the individual is discarded as lethal.

At this point, all the nodes that are used by the program have been identified and so their values can be calculated. For mathematical and binary functions, these operations are performed in the usual manner. However, SMCGP has a number of special functions (see Table 1) that allow for self-modification.

If a function is a self-modification function, then it may be activated. Binary functions are always activated, but numeric nodes are activated only if the first input is larger than the second input. The self-modification operation of an activated node is added to a list of pending operations - the 'ToDo' list. After execution, the self-modification functions on the ToDo list are applied to the current graph, up to a maximum number of self-modification operations which is a parameter of the system.

In turn, the self-modification functions usually require parameters, which are taken from the parameters part of the calling node. Many of the parameters are integers, so the parameters may need to be cast into integer numbers. For instance, parameters may be treated as relative addresses depending on the function. The program can now be iterated again, if necessary. It is important to note that modifications are only made to the phenotype, and not to the genotype.

In the current work, we extended SMCGP to allow for design generation. To do this, several changes to SMCGP are required. Extra functions are added to the function set that perform various drawing operations. To support this, each node in the genotype additionally encodes for a structure that can represent the parameters of these shape functions, the shape data type (SDT).

An SDT structure contains five vectors of four element each. One of these vectors is labeled as "source", another is labeled as "destination". Four-element vectors are used so that we can easily move to a 3D representation. In 3D geometry, 4 element vectors are very useful as they can represent rotations and transformations as quaternions. Elements are referred to as x, y, z and w. The five vectors in the SDT represent entry and exit location, rotation, size and value. Value can be considered as a holder for some additional parameters.

These source and destination vectors are used when mathematical operations are applied upon two SDT structures. For example, to ADD two structures A and B, a copy of A is made and the destination vector in A is set to the sum of the source vectors in A and B. The structure contains 5 vectors and each vector encodes a different parameter that is used to specify a shape location, size, rotation and connectivity, as well as an additional parameter.

When a shape is drawn, it is drawn with respect to a current position (origin) and rotation. Further parameters specify the size of each dimension of the shape. Two of the parameters encode additional position information as to where to start drawing the object (entry position) and where the next origin should be (exit position). The entry and exit positions are relative to the origin and rotation.

Consider Figure 1. The origin and initial rotation are determined by the previous drawing operation, or by a predetermined position for the first shape. The shape is then drawn relative to the origin and entry positions. The subsequent origin will be the exit position of the last node, and the subsequent

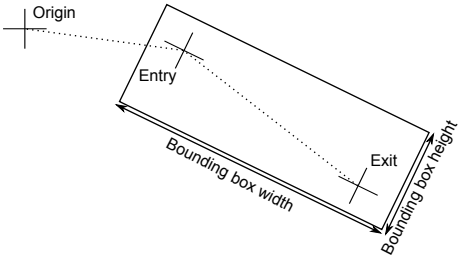

Fig. 1 Shapes are defined relative to an initial origin, with Entry and Exit positions defined relative to the shape. The entry and exit positions, shape rotation and shape geometry are under evolutionary control, and can be modified by the SMCGP program at run time.

initial rotation vector will be the current rotation added to this shape's rotation vector. Entry, exit and rotation values are taken from the SDT structure.

The shape parameters are calculated from the value of the SDT passed to that node. Hence, they can be affected by computations performed by mathematical functions. This allows for more complex transformations to be performed.

To simplify the shape generation, only one shape function is allowed. This function, called the "Superformula", is able to generate a wide variety of shapes, including many that have a very biological feel [5]. Conveniently, the function also can be extended to 3D which will be useful in later work. In polar form the equation is:

$$r(\phi) = \left[\left| \frac{\cos(\frac{m\phi}{4})}{a} \right|^{n_2} + \left| \frac{\sin(\frac{m\phi}{4})}{b} \right|^{n_3} \right]^{-\left(\frac{1}{n_1}\right)}$$

Each of the parameters a, b, m, n_1, n_2 and n_3 is under evolutionary control, defined by the values stored in the SDT. a, b are taken from the z and w component of the size vector. The other four parameters are taken from the value vector.

Because the formula is written in polar coordinates, results needs to be converted to Cartesian coordinates (p and q). In this transformation, the x and y values from the size vector are used to specify the radii of the transform:

$$p = x \sin(\phi)$$

$$q = y \cos(\phi)$$

The function set also contains other functions for manipulating the current origin and rotation. The MOVE command specifies a simple translation of the current origin. TRANSMC allows for the origin to be moved and scaled. A stack of origin and rotation values is also provided. The PUSHTRANSFORM and POPTRANSFORM perform operations on this stack.

Figure 2 shows the developmental steps of a simple object (a rough outline of an aeroplane). Each frame in the sequence shows the next time step in the developmental process. Here, all but the last time step add new shapes to the figure.

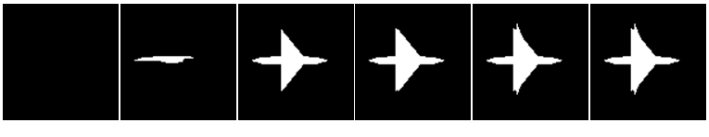

Fig. 2 Developmental steps in drawing a simple object. Each frame represents one time step in the developmental process.

Table 1 The SMCGP function set.

Function	Description
MOVE	Move the origin
POLYGON	Draw a polygon
PUSHTRANSFORM	Push origin/rotation to stack
POPTRANSFORM	Pop origin/rotation from stack
TRANSMC	Translate and scale the current origin.
ADD, SUB, DIV, MUL	Perform the relevant mathematical operation on the source vectors
PRC	Executes a subgraph as a procedure
MOVESRCTODEST	Moves a vector from the source register to the output register
INDEX	Returns a STD that represents the current index of the node in the graph
CONST	Returns a STD that represents a set of evolved mathematical constants
OUTPUT	Labels this node as being the output, i.e. final connected node in the graph
SMDUP	Duplicates a set of nodes, inserts copy elsewhere in the graph
SMDEL	Deletes a set of nodes
SMDUPREV	Same as SMDUP, but reverses the order of the inserted nodes

4 Graphics Processing Units (GPUs)

Graphics Processing Units (GPUs) have a many-core parallel architecture. They consist of a set of stream processors that execute programs (also called kernels) in parallel. GPUs were originally designed for graphics processing, so the stream processors are designed for small and fast operations (per stream processor) such as filtering a texture. A simple description of GPU programming and hardware models is given in this section. For more information about the both GPU architecture(s) and programming models readers should consult [21].

The programming model used for GPUs is built around a SIMT (single-instruction multiple-thread) architecture concept. SIMT is not the same as the traditional SIMD (single instruction multiple data) concept in that SIMD applies the same instruction to multiple pieces of data simultaneously, while SIMT executes the same thread (code block) simultaneously with a single instruction. A typical program execution on a GPU consists of a mapping of the threads (or kernels) to a two-level grid of a user-specified size (see Figure 3). The threads are mapped as a set of threads, grouped into blocks. The number of blocks in the grid is called the *grid size* and the number of threads per block is called the *block size*. Once the threads are mapped they are then enumerated and distributed to the available cores on the device. Scheduling of these kernels, as threads of the grid are terminated and new ones are executed, is performed automatically on the GPU itself.

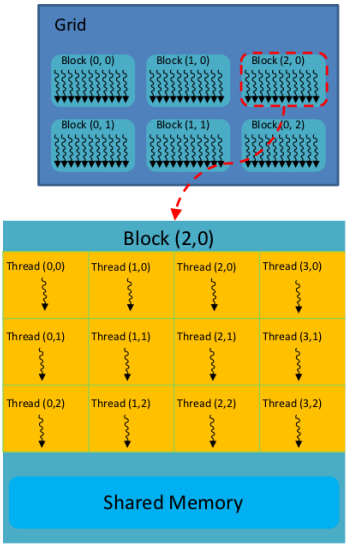

Fig. 3 GPU Program Model

The hardware model of most GPUs, illustrated in Figure 4, are generally designed as an array of multi-threaded Streaming Multiprocessors (SMs). Each of these processors contain a set of Scalar Processor (SP) cores (currently all NVIDIA devices contain eight cores per SM), a multi threaded instruction unit, and a shared memory unit for that multiprocessor. Outside of the array of SMs there is at least one memory space, most importantly the main device memory, that is in use by all components of the GPU (other memory spaces are not relevant to this chapter). Device memory is the slowest on-card memory, while shared memory (per SM) is the fastest on-card, next to the registers of course but not far behind [21].

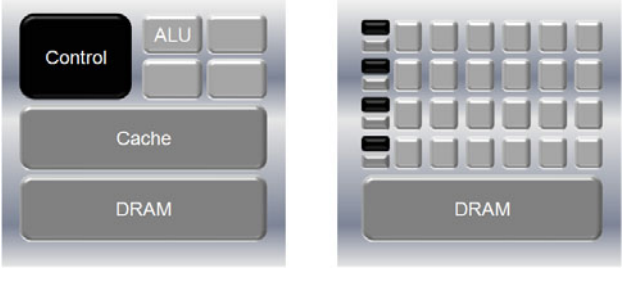

Fig. 4 GPU Hardware Model

A general set of optimization rules for developing any type of algorithm for GPUs is:

Memory Transfers. Memory transfer to or from device memory is the slowest individual operation that can be performed on a GPU. For this reason memory transfers should be kept to a minimum for any algorithm developed for the GPU. An optimal design approach is to design your algorithm to perform all operations on the main data in device memory and transfer only the results back to host memory.

Memory Coalescence. Memory coalescence is the pattern of reads performed on the device memory. For example, if text is read one word at a time (each kernel reads one word, and assuming words are stored in memory in the order that they are written) to perform some operation on each word, then each kernel should read blocks of memory locations that are contiguous. In contrast, an algorithm that would have kernels read a set of random words from the text would cause many random read locations per kernel and therefore be inefficient. The former is an example of good memory coalescence which is ultimately a consequence of the architectural design choices made for GPUs optimized for a uniform memory access strategy.

Domain Decomposition. When developing a parallel algorithm care should be taken to decompose the domain in order to allow separate thread blocks to run on separate sub-domains of a discretized physical system. This will allow memory access and thread usage/scheduling for thread blocks on multiprocessors to be optimized. Keeping with the text reading example, one would decompose a text into separate paragraphs and map one word to one thread and one paragraph to one thread block so that each multiprocessor can process a paragraph at a time.

Shared Memory. Shared memory is much faster than device memory on GPUs. If an algorithm involves reads of the same subset of data for multiple kernels with a thread block, this data should be loaded into shared memory at the start of the thread block (which would require a thread sync call in order for all threads within the block to be processed up to the point where shared memory is loaded). Using the text reading example again, it would be the requirement that each thread would have to have read access to three words (the working word, the one prior it and the one directly following) which would cause an overlap in the read of surrounding words. The optimal approach would be to load all words in a paragraph into the shared memory of that thread block and then process the data.

Multiprocessor Occupancy. The GPU occupancy (CUDA Occupancy for NVIDIA GPUs) is a measure of kernel invocation that describes how well the kernels make use of the multiprocessor resources located on GPUs, such as allocated registers and shared memory. This concept is best described by NVIDIA [21] and is related to domain decomposition and algorithm design. Care must be taken to divide a sequential algorithm into operations that can be converted to kernel calls so that each kernel does not require too much limited resources (shared memory, registers, etc). Otherwise, the algorithm is not optimally designed and resources could be wasted. The overall goal is to maximize the multiprocessor occupancy measure.

5 Computational Fluid Dynamics (CFD) on Graphics Processing Units

The governing equations of a fluid system are at a minimum the continuity equation for mass and the Navier-Stokes equation, although others may be applied as required by the system in question, such as the equation of state, conservation of mass, conservation of energy, and/or boundary condition equations. The continuity equation for mass and the Navier-Stokes equation will be all we can discuss in this chapter, but the method can be easily extended to other equations using the same techniques.

The continuity equation is a description of the transport of mass under mass conservation,

$$\frac{\partial \rho}{\partial t} + \nabla \cdot (\rho \mathbf{u}) = 0 \tag{1}$$

where ρ is the density of the fluid, t is the time, and $\mathbf{u}$ is the velocity vector. Since we are only concerned with incompressible fluid flow, the incompressible Navier-Stokes equation is relevant,

$$\rho \left(\frac{\partial \mathbf{u}}{\partial t} + \mathbf{u} \cdot \nabla \mathbf{u} \right) = -\nabla p + \mu \nabla^2 \mathbf{u} + \mathbf{f} \tag{2}$$

where p is the pressure, and $\mathbf{f}$ symbolizes external forces. Equations (1) and (2) comprise the required equations for solving incompressible transient (time dependent) fluid flow. It is important to understand that not all solutions to fluid flow are required to be transient, some simple flows have time independent solutions, or steady-state solutions. The steady-state equations are similar to the above equations but do not contain explicit time dependence.

The main issue in solving these equations is that they are coupled nonlinear differential equations. Solving these types of equations usually requires an iterative method that optimizes an approximation to the equation solution. Next, we will describe the iterative methods used to solve both the steady-state and transient fluid flow equations.

5.1 Method for Solving Fluid Equations

The iterative method used to solve transient fluid flow equations is called the PISO (Pressure Implicit Splitting of Operators) method. This method requires that the system be discretized, we take the example of a finite volume (FV) discretization here. The PISO (Pressure Implicit with Splitting of Operators) method is described in Algorithm 1.

From this algorithm, the most computationally expensive step for each iteration is solving the momentum and pressure correction equations, which are systems of linear equations. Another iterative method can be used to solve these systems of linear equations. It is called the successive over-relaxation (SOR) method and we will discuss it in the next section, specifically a version for GPUs (the SOR-GPU method).

5.2 Solving Fluid Equations on GPUs

The fluid flow simulation algorithm discussed in Section 5.1 requires a general set of operations:

- Construct coefficient matrices for systems of linear equations
- Solve systems of linear equations
- Apply corrections to flow fields
- Check convergence (residual sum).

In order to ensure speed optimization on GPUs it is best to keep all data in GPU memory with minimal swapping to host or main memory because host-to-device and device-to-host memory transfers are the slowest single operation on GPUs. The fluid simulation method discussed in this chapter keeps all relevant data in the GPU memory at all times. While this limits the size of the system we can simulate (to the amount of memory available on the GPU), it ensures optimal simulation speed. The following sections describe the design of these operations optimized for GPUs.

Algorithm 1. PISO algorithm

Initialize guesses for p^*, u^*, v^*.
repeat
  {STEP 1: Solve discretized momentum equations to get u^*, and v^*}
  $a_{i,j}u^* = \sum a_{nb}u^*_{nb} + \frac{1}{2}(p^*_{i-1,j} - p^*_{i+1,j})A_{i,j} + b_{i,j}$
  $a_{i,j}v^* = \sum a_{nb}v^*_{nb} + \frac{1}{2}(p^*_{i,j-1} - p^*_{i,j+1})A_{i,j} + b_{i,j}$

  {STEP 2: Solve pressure correction equation to get p'}
  $a_{i,j}p'_{i,j} = a_{i-1,j}p'_{i-1,j} + a_{i+1,j}p'_{i+,j} + a_{i,j-1}p'_{i,j-1} + a_{i,j+1}p'_{i,j+1} + b_{i,j}$

  {STEP 3: Correct pressure and velocities}
  $p_{i,j} = p^*_{i,j} + p'_{i,j}$
  $u_{i,j} = u^*_{i,j} + \frac{1}{2}d\_u_{i,j}(p'_{i-1,j} - p'_{i+1,j})$
  $v_{i,j} = v^*_{i,j} + \frac{1}{2}d\_v_{i,j}(p'_{i,j-1} - p'_{i,j-1})$

  $p^* = p$; $u^* = u$; $v^* = v$

  {STEP 4: Solve second pressure correction equation to get p'}
  $a_{i,j}p''_{i,j} = a_{i-1,j}p''_{i-1,j} + a_{i+1,j}p''_{i+,j} + a_{i,j-1}p''_{i,j-1} + a_{i,j+1}p''_{i,j+1} + b_{i,j}$

  {STEP 5: Correct pressure and velocities using second pressure correction}
  $p_{i,j} = p^*_{i,j} + p''_{i,j}$
  $u_{i,j} = u^*_{i,j} + \frac{1}{2}d\_u_{i,j}(p''_{i-1,j} - p''_{i+1,j})$
  $v_{i,j} = v^*_{i,j} + \frac{1}{2}d\_v_{i,j}(p''_{i,j-1} - p''_{i,j-1})$

  $p^* = p$; $u^* = u$; $v^* = v$
until convergence

5.2.1 Construction of Coefficient Matrices

Constructing coefficient matrices for each system of linear equations is the first operation to be parallelized on the GPU architecture. This operation can be performed with a single GPU program, or kernel, for each type of equation, e.g each velocity component, pressure correction, second pressure correction. Since the matrix is sparse involving only coefficients for direct neighbor nodes memory on the device needs only to be allocated for neighboring coefficients, not for the full matrix. Access to field values at a local node and at direct neighbors is also required, and since global memory access on GPUs is their most important bottleneck, shared memory is used to store nodes per block in order to reduce the number of duplicate memory accesses.

5.2.2 Solving Systems of Linear Equations

The most important part of the implementation is the solution method used for solving systems of linear equations, since this operation is performed up to *2 + number of dimensions* times for each iteration. Normally (on a CPU), a type of preconditioned conjugate gradient (CG) method would the best choice for these linear solvers, but the CG method involves a matrix-vector multiplication and a vector-vector summation, which, compared to a linear solution method such as the Gauss-Seidel (GS) or successive over-relaxation (SOR) methods (with a single update per iteration), is much more expensive computationally on a GPU. The reason this is so much more expensive on a GPU is that GPUs do not handle random memory access very well, that is they are built and optimized for a uniform memory access strategy (coalesced memory access). For this reason, the SOR method is used for all linear solvers discussed in this chapter.

Successive Over-Relaxation Method on GPUs. The successive over-relaxation (SOR) method is an iterative method for solving linear systems of equations. It involves a single update per node in the system. The general algorithm is described in Algorithm 2, a full description of the method and the terms can be found in [16]. For the purpose of this chapter, we note that the algorithm is composed of an outer iteration loop for all nodes in the system with a single update to each node.

Algorithm 2. Successive over-relaxation algorithm

```
1: for iter=0:maxiter do
2:    for i = 2:(m+1) do
3:       for j = 2:(n+1) do
4:          u(i,j) = ω((a_W(i,j)u(i−1,j)+a_E(i,j)u(i+1,j)+a_S(i,j)u(i,j−
            1) + a_N(i,j)u(i,j + 1) + b(i,j))/a_P(i,j)) + (1 − ω)u(i,j)
5:       end for
6:    end for
7:    if convergence then
8:       break
9:    end if
10: end for
```

The GPU implementation of the Gauss-Seidel (GS) or successive over-relaxation method (both methods are very similar and the terms will be used interchangeably in the rest of this chapter) is a type of domain decomposition of the numerical fluid system. The implementation is not a straightforward domain decomposition, however, it involves making two passes over the system per iteration, although each node is only updated once per iteration.

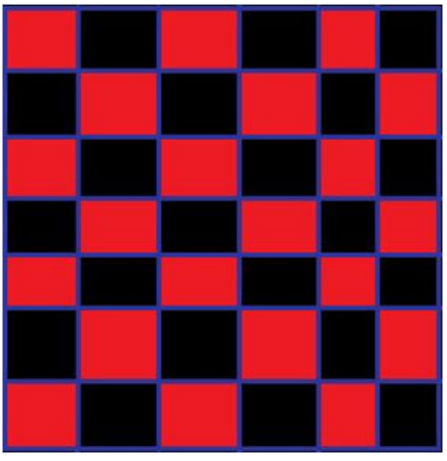

Fig. 5 Red Black Nodes

Initially the method "colors" each node in the system by two alternating colors so that no node has neighboring nodes of the same color, such as in Figure 5. This coloring of nodes (two colors for a uniform two dimensional mesh) is why this parallel technique for the GS is also known as the red-black or the checkerboard method. Once each node is assigned a virtual color, we continue as we would in the sequential version of the method, with for one change: at each iteration there are two passes over the nodes, the first pass updates one set of colored nodes (the red nodes) and the second pass updates the second set of colored nodes (black nodes). Then we iterate as normal until convergence is reached. So far the parallel algorithm may look something like (in sequential form for now) Algorithm 3.

Algorithm 3. Parallel (Red-Black) Gauss-Seidel algorithm

```
1: for iter=0:maxiter do
2:    for i = all RED nodes do
3:       update u(i)
4:    end for
5:    for i = all BLACK nodes do
6:       update u(i)
7:    end for
8:    if convergence then
9:       break
10:   end if
11: end for
```

The advantage of this algorithm, in a parallel sense, is that all RED nodes can be updated simultaneously and all BLACK nodes can be updated simultaneously since from the sequential Algorithm 2 we know that only neighboring nodes are read during each node update. Since neighboring nodes will definitely not be updated at the same time (because of the different coloring), this allows us to perform updates on all nodes of the same color simultaneously.

Now that we have the general idea of the algorithm, we can move on to a more custom implementation for GPUs. First of all, we can map each node to a single thread. With the implementation up to this point, for a two dimensional system, five reads and one write per node update are required. The write is to the node that is mapped to the thread (the local node) and the reads are from the local node and its direct neighbors, as indicated by the white dots in Figure 6.

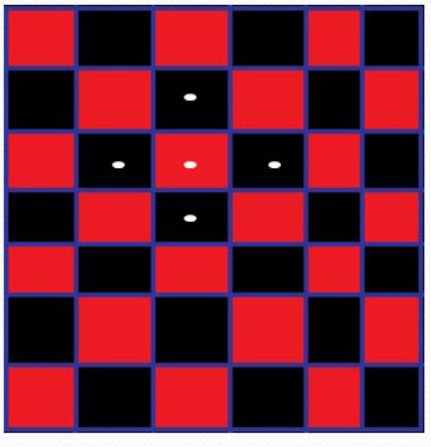

Fig. 6 Red Black nodes with local and neighboring nodes

As of the algorithm developed so far, we use global GPU memory for the five reads per node update, which requires many duplicate reads per update since all neighboring nodes of a single local node are being read at least one more time and up to four more times per half iteration (per single color update pass). If we recall Section 4, the GPU programming model uses a set of blocks, where each block contains a set of threads, and each block has access to more efficient memory (called shared memory in the section above). If we make use of this shared memory per block we can remove nearly all of these duplicate reads by loading all nodes in a block into shared memory before we do the update. The set of nodes required for a block to update all of its associated threads (from the running example) are indicated by white lines in Figure 7. If we load all of these nodes into shared memory, including the ghost layer which is the layer of non local nodes (nodes that do not need to be updated by this current block) that surround the edges of the block,

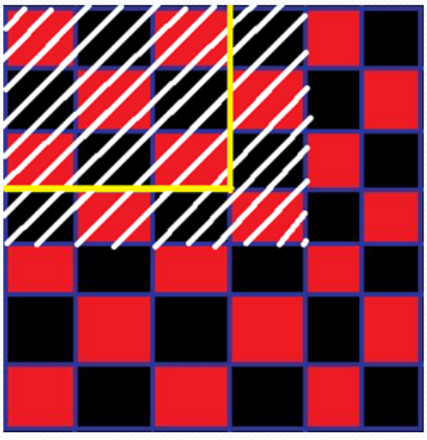

Fig. 7 Red Black nodes that must be read per block (block is highlighted in yellow)

we can reduce the number of reads by a factor of almost four along with the memory access time for these reads since shared memory is much more efficient.

The pseudo code for the kernel (see Section 4) that is executed for each thread would then look something like Algorithm 4. Lines 2 to 15 load all local nodes and the ghost layer into shared memory. Line 17 uses the *__syncthreads()* function, which causes all threads in the block to wait at this location in the code until all of them have reached that point, this way all required data is loaded into shared memory before we start to do any updates (reads and writes) using this data.

5.2.3 Applying Corrections to Flow Fields

The application of the corrections to the flow fields is simply a kernel that applies the correction to each node. Since these corrections require only access to local field values at each node (as opposed to field values at neighbor nodes), a simple update per kernel is most efficient.

5.2.4 Convergence Check

Convergence of the PISO method can be determined in many different ways, depending on the application of the method. The most popular methods are a check of the velocity residual sum against a tolerance, or a check of the norm of pressure correction against a tolerance.

For both residual sum and norm calculations on the GPU we must perform a sum. This may seem simple but to efficiently do this on a GPU a little work is required. To do an efficient sum of a large vector on the GPU we do a

parallel sum reduction. The method used in this work is defined in [28], and uses a tree based approach within each thread block, as illustrated in Figure 8. This algorithm works by assigning a uniform and contiguous subset of the vector to each thread block, each thread block then performs the sum of its associated subset and stores the result in the first memory location of its subset (denoted by the child node in the figure). It recursively does this until only one value is left (moved down the tree in the figure), which is the sum of the original vector. The time complexity of this technique is $O(N/\#Blocks + logN)$, vs $O(N)$ if we were to use a simple loop for summation.

Algorithm 4. GPU Gauss-Seidel algorithm

```
{Load local node into shared memory}
u_shared[s_i][s_j] = u[ij];
{check if on edge node, if yes then load ghost layer}
if threadIdx.x == 0 then
    u_shared[s_i-1][s_j] = u[i-1][j];
end if
if threadIdx.x == BLOCK_SIZE_X-1 then
    u_shared[s_i+1][s_j] = u[i+1][j];
end if
if threadIdx.y == 0 then
    u_shared[s_i][s_j-1] = u[i][j-1];
end if
if threadIdx.y == BLOCK_SIZE_Y-1 then
    u_shared[s_i][s_j+1] = u[i][j+1];
end if
{wait for all threads in block to finish loading shared memory}
__syncthreads();
for i,j = all RED or BLACK nodes only do
    update u[i][j]
end for
if convergence then
    break
end if
```

Advantages of this technique are not only that the majority of the computations are performed on the GPU but that the vector itself never needs to leave the GPU (which is preferred since all other calculations for the PISO method are on the GPU). Further, only one value needs to be copied from GPU memory to host memory. As we have noted in Section 4, copying from device to host memory is one of the most serious bottlenecks in any GPU implementation.

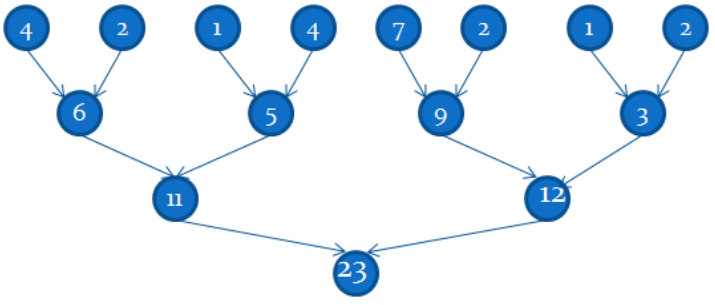

Fig. 8 Parallel sum reduction using tree based approach within each thread block

6 Optimized Shape Design with SMCGP and CFD-GPU

Optimized shape design is the optimization of shapes in order to minimize and/or maximize specific parameters of the shape. The technique described throughout this chapter is an optimization of a shape to minimize drag and maximize lift within a fluid. The results of this experiment can be easily predicted to develop a shape that is more aerodynamically "smooth", such as that of an airfoil or hydrofoil at some optimal angle of attack.

The technique used to drive the optimization is genetic programming as described in Section 3. The fluid simulation technique described in Section 5 is used to evaluate fitness. Since the GP method could potentially require millions of evaluations in order to evolve an optimal solution we have to use this fluid simulation technique on a parallel architecture to increase performance. This is required to achieve an optimal solution in a practical period of time.

To tackle the optimized shape design problem we expect that about a million evaluations are required. Table 2 illustrates run time estimates for GP to converge with a 1024×512 discretized fluid system. As shown in this table, it is evident that to perform this shape optimization on a single CPU is very impractical (as it would require 10 years). But it requires only 10 days on a cluster of 50 (average) GPUs.[1]

Figure 9 illustrates a simple result of this technique for a partial evolution. This figure shows only a very small subset of the shapes as the evolution progresses in order to illustrate the effectiveness of the technique.

It is interesting to observe changes in the design of shapes during evolution. For example, Figure 10 shows one experimental run. In the initial population, a simple triangle is found, and this shape forms the basis for further evolution. At first evolution modifies the parameters of this shape, making it

[1] For example a nVidia GeFore 9800 GT, with 120 cores and 1 GB of memory produces about 336 GFLOPS, is average at the time of this writing.

Table 2 Optimized Shape Design Problem: Estimated Times

Hardware	GP Convergence Time
1 CPU	10 years
50 CPU Cluster	70 days
1 GPU	1.4 years
50 GPU Cluster	10 days

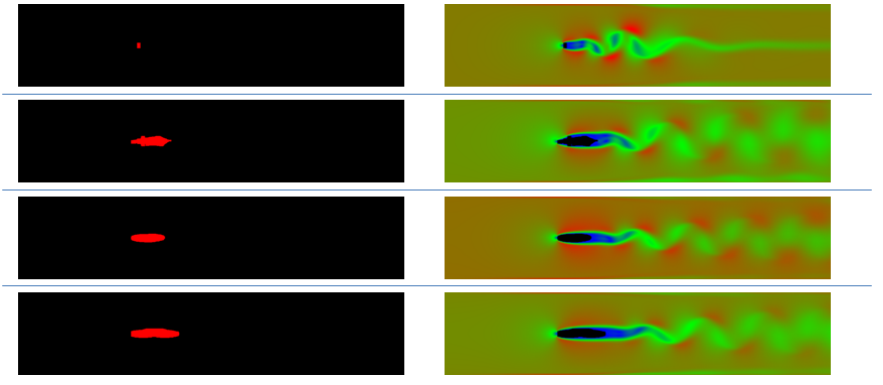

Fig. 9 Shape evolution for drag minimization

more angled at the front to reduce drag. Next evolution introduces a small spike on the top of the shape, which will alter the flow over the top of the surface. Eventually, this spike is smoothed into a small lump, that will have less drag. This shape then further changes into larger, rounder shape which encompasses the entire front of the triangle further reducing drag. Finally, the evolution approximates a shape very similar to the familiar shape of a wing cross-section.

6.1 Measuring Fitness Using CFD

The fitness function uses the CFD simulation to obtain drag and lift coefficient for a design. The fitness score is the absolute lift minus the absolute drag. Hence, fitness scores above zero represent objects with more lift than drag, and therefore indicate that a lifting body has been evolved. Here we also look for shapes that have a minimal size. In order to enforce this, we insert a block into the environment that the shapes must form around. The block has zero lift and a large drag. To get a good fitness score, this block must somehow be incorporated into the design.

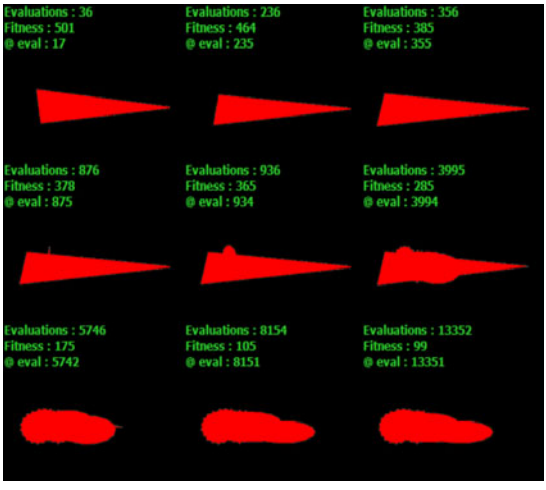

Fig. 10 Sequence showing the best shape at different times within an evolutionary experiment.

For the fitness function to work correctly, shape designs that will not simulate correctly are discarded. In particular, very small and very large structures are not tested. We also currently discard non-contiguous shapes which, if we were to construct the shape, would mean it could be fashioned from a single piece.

6.2 Evolutionary Algorithm

The actual evolutionary algorithm used was also parallel, and distributed in nature. As the evaluation time of individuals would be different (dependent not only on the convergence properties of the simulation, but also on the computing hardware used), to work efficiently, the algorithm also has to work in an asynchronous and non-blocking way, so that all available computing resources are always helping with the search. The evolutionary algorithm is based on [13], as this was found to work efficiently in an asynchronous environment. A central population of individuals is stored on the root computer, and when a client node finishes processing an individual it is returned to this population. When a client requires a new individual to process, individuals are selected from this population and crossover/mutation applied to produce a new individual for evaluation. As this shared population increases in time as evaluated individuals are added, it periodically needs to be reduced in size. In [13] this dynamic of a variable population size was found to be beneficial to evolution. Figure 11 shows an example of the algorithm running.

The most significant parameters are shown in Table 3.

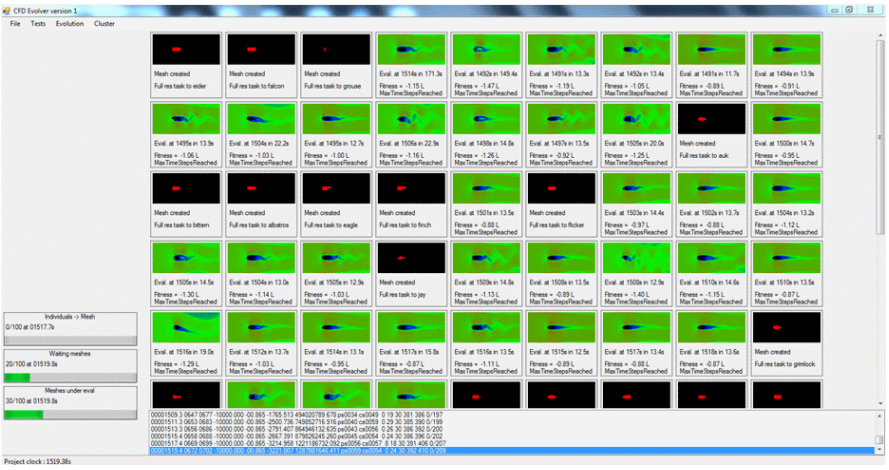

Fig. 11 Screen shot of the shape evolver. The small tiles show either an individual that is currently being processed, or the result of a simulation for an individual.

Table 3 Parameters of the evolutionary algorithm.

Parameter	Value
Initial genotype length	20 nodes
Mutation rate	0.05
Initial Population Size	50
Frequency of population resizes	60 seconds

7 Conclusions

This chapter described a technique inspired by natural evolution and applied to a shape optimization design problem. The evolutionary technique itself was developed as a parallel algorithm for a distributed system, along with the fitness evaluation on a GPU parallel architecture. A focus on efficient algorithm design is necessary since fluid simulations are a very computationally expensive task, while the evolutionary algorithm discussed would require on the order of millions of evaluations of fluid simulations.

The fluid dynamics solver described in this chapter applied a popular iterative method for solving the pressure-coupled governing fluid flow equations adapted to the GPU to allow for faster evaluation times. The adaptation to the GPU required several parallel optimization steps many of which are general purpose optimizations that can be extended to other problems to be solved on GPUs.

The optimized shape design method described in this chapter also fulfills the requirement of having the capability to produce fully general shapes, where there are minimal constraints on the design. This minimal constraint design results from the SMCGP method allowing any combination of shapes to produce the final result. This requirement is an advantage in design optimization since it allows for new designs that may not have been expected, and may not have been possible with an otherwise more constrained approach.

Acknowledgements. The authors would also like to thank Dr. Taras Kowaliw for suggesting the use of the 'super function' for shape encoding. W.B. acknowledges funding by NSERC under the Discovery Grant Program, RGPIN 283304-07. S.H. was supported by a postdoctoral grant from the Atlantic Canada Computational Excellence Network (ACENET).

References

1. Asouti, V.G., Giannakoglou, K.C.: Aerodynamic optimization using a parallel asynchronous evolutionary algorithm controlled by strongly interacting demes. Engineering Optimization 41(3), 241 (2009)
2. Banzhaf, W., Miller, J.: The challenge of complexity. In: Menon, A. (ed.) Frontiers of Evolutionary Computation, pp. 243–260. Springer (2004)
3. Billings, D.: PDE Nozzle Optimization Using a Genetic Algorithm. Technical report. Marshall Space Flight Center (2000)
4. Giannakoglou, K., Papadimitriou, D., Kampolis, I.: Aerodynamic shape design using evolutionary algorithms and new gradient-assisted metamodels. Computer Methods in Applied Mechanics and Engineering 195(44-47), 6312–6329 (2006)
5. Gielis, J.: A generic geometric transformation that unifies a wide range of natural and abstract shapes. Am. J. Bot. 90(3), 333–338 (2003)
6. Harding, S., Banzhaf, W., Miller, J.F.: A survey of self modifying cartesian genetic programming. In: Riolo, R., McConaghy, T., Vladislavleva, E. (eds.) Genetic Programming Theory and Practice VIII. Genetic and Evolutionary Computation, 20-22 May, ch. 6, vol. 8, pp. 91–107. Springer, Ann Arbor (2010)
7. Harding, S., Miller, J., Banzhaf, W.: Self Modifying Cartesian Genetic Programming: Fibonacci, Squares, Regression and Summing. In: Vanneschi, L., Gustafson, S., Moraglio, A., De Falco, I., Ebner, M. (eds.) EuroGP 2009. LNCS, vol. 5481, pp. 133–144. Springer, Heidelberg (2009)
8. Harding, S., Miller, J.F., Banzhaf, W.: Evolution, development and learning with self modifying cartesian genetic programming. In: GECCO 2009: Proceedings of the 11th Annual Conference on Genetic and Evolutionary Computation, pp. 699–706. ACM Press, New York (2009)
9. Harding, S., Miller, J.F., Banzhaf, W.: Self modifying cartesian genetic programming: Parity. In: Tyrrell, A. (ed.) 2009 IEEE Congress on Evolutionary Computation, Trondheim, Norway, May 18-21, pp. 285–292. IEEE Computational Intelligence Society, IEEE Press (2009)

10. Harding, S., Miller, J.F., Banzhaf, W.: Developments in cartesian genetic programming: self-modifying CGP. Genetic Programming and Evolvable Machines 11(3/4), 397–439 (2010); Tenth Anniversary Issue: Progress in Genetic Programming and Evolvable Machines
11. Harding, S., Miller, J.F., Banzhaf, W.: Self modifying cartesian genetic programming: finding algorithms that calculate pi and e to arbitrary precision. In: Branke, J., Pelikan, M., Alba, E., Arnold, D.V., Bongard, J., Brabazon, A., Branke, J., Butz, M.V., Clune, J., Cohen, M., Deb, K., Engelbrecht, A.P., Krasnogor, N., Miller, J.F., O'Neill, M., Sastry, K., Thierens, D., van Hemert, J., Vanneschi, L., Witt, C. (eds.) GECCO 2010: Proceedings of the 12th Annual Conference on Genetic and Evolutionary Computation, Portland, Oregon, USA, 7-11 July, pp. 579–586. ACM (2010)
12. Harding, S., Miller, J.F., Banzhaf, W.: SMCGP2: finding algorithms that approximate numerical constants using quaternions and complex numbers. In: Krasnogor, N., Lanzi, P.L., Engelbrecht, A., Pelta, D., Gershenson, C., Squillero, G., Freitas, A., Ritchie, M., Preuss, M., Gagne, C., Ong, Y.S., Raidl, G., Gallager, M., Lozano, J., Coello-Coello, C., Silva, D.L., Hansen, N., Meyer-Nieberg, S., Smith, J., Eiben, G., Bernado-Mansilla, E., Browne, W., Spector, L., Yu, T., Clune, J., Hornby, G., Wong, M.-L., Collet, P., Gustafson, S., Watson, J.-P., Sipper, M., Poulding, S., Ochoa, G., Schoenauer, M., Witt, C., Auger, A. (eds.) GECCO 2011: Proceedings of the 13th Annual Conference Companion on Genetic and Evolutionary Computation, Dublin, Ireland, July 12-16, pp. 197–198. ACM (2011)
13. Hu, T., Harding, S., Banzhaf, W.: Variable population size and evolution acceleration: a case study with a parallel evolutionary algorithm. Genetic Programming and Evolvable Machines 11(2), 205–225 (2010)
14. Koza, J.R.: Genetic Programming: On the Programming of Computers by Means of Natural Selection. MIT Press, Cambridge (1992)
15. Kumar, S., Bentley, P.: On Growth, Form and Computers. Academic Press (2003)
16. LeVeque, R.J.: Finite Difference Methods for Ordinary and Partial Differential Equations. In: Society for Industry and Applied Mathematics Proceedings (2007)
17. Miller, J., Banzhaf, W.: Evolving the program for a cell: from french flags to boolean circuits. In: Kumar, S., Bentley, P. (eds.) On Growth, Form and Computers, pp. 278–301. Academic Press, London (2003)
18. Miller, J.F.: An empirical study of the efficiency of learning boolean functions using a cartesian genetic programming approach. In: Proceedings of the 1999 Genetic and Evolutionary Computation Conference (GECCO), Orlando, Florida, pp. 1135–1142. Morgan Kaufmann (1999)
19. Miller, J.F.: Evolving Developmental Programs for Adaptation, Morphogenesis, and Self-Repair. In: Banzhaf, W., Ziegler, J., Christaller, T., Dittrich, P., Kim, J.T. (eds.) ECAL 2003. LNCS (LNAI), vol. 2801, pp. 256–265. Springer, Heidelberg (2003)
20. Miller, J.F., Smith, S.L.: Redundancy and computational efficiency in cartesian genetic programming. IEEE Transactions on Evoluationary Computation 10, 167–174 (2006)
21. nVidia: CUDA Programming Guide. nVidia Corporation, Version 2.3 (2009)

22. Obayashi, S., Tsukahara, T., Nakamura, T.: Multiobjective genetic algorithm applied to aerodynamic design of cascade airfoils. IEEE Transactions on Industrial Electronics 47(1), 211–216 (2000)
23. Poli, R., Langdon, W.B., McPhee, N.F.: A field guide to genetic programming (2008) (With contributions by Koza, J.R.), Published via, http://lulu.com, freely Available at, http://www.gp-field-guide.org.uk
24. Poloni, C., Giurgevich, A., Onesti, L., Pediroda, V.: Hybridization of a multiobjective genetic algorithm, a neural network and a classical optimizer for a complex design problem in fluid dynamics. Computer Methods in Applied Mechanics and Engineering 186(2-4), 403–420 (2000)
25. Rechenberg, I.: Evolutionsstrategie: optimierung technischer systeme nach prinzipien der biologischen evolution. Frommann-Holzboog (1973)
26. Rechenberg, I.: Case studies in evolutionary experimentation and computation. Computer Methods in Applied Mechanics and Engineering 186(2-4), 125–140 (2000)
27. Schwefel, H.-P.: Numerische Optimierung von Computer-Modellen mittels der Evolutionsstrategie. Mit einer vergleichenden Einführung in die Hill-Climbing- und Zufallsstrategien. Journal of Applied Mathematics and Mechanics/Zeitschrift für Angewandte Mathematik und Mechanik 60 (1977)
28. Sengupta, Harris, Garland: Efficient parallel scan algorithms for gpus. nVidia Technical Report NVR-2008-003. nVidia Corporation (2008)