Middleware for Scientific Computing on GPUs from Fortran 95

Yuancheng Luo, Ramani Duraiswami, Nail A. Gumerov, Kate Despain, William D. Doland
Perceptual Interfaces and Reality Laboratory,
Institute for Advanced Computer Studies,
University of Maryland, College Park

Abstract

Scientific computing and numerical analysis techniques have been widely adapted and implemented in the Fortran language since its inception. Many successive versions of Fortran have allowed new functionalities to be incorporated into the language while maintaining backward compatibility with older code. The current standard, Fortran 2003, is extensively used in high-performance computing on supercomputers and computer clusters. While Fortran users may have access to these nodes, fast processing times are still possible on a single desktop system. The graphical processing unit (GPU) on modern graphic cards can perform a large amount of instructions in parallel execution. By employing programmable GPUs and their respective frameworks, many scientific computations are sped-up over their CPU counterparts at a high cost of rewriting code for specific GPU architectures. To address this issue, we have developed a middleware library on top of Fortran 95 and later versions that interfaces with the GPU. Details of the middleware's design and model are presented with the results from a sample application. The source code is available as open source.

1 Introduction

Recent developments in high performance graphics hardware have given rise to fast parallel-processing processors capable of performing at several hundred gigaflops. These graphics hardware are relatively inexpensive and easily installable on most workstations while allowing for a multifold increase in potential computational power. However the hardware is not easily programmable which is a concerning issue for its use in scientific computing.

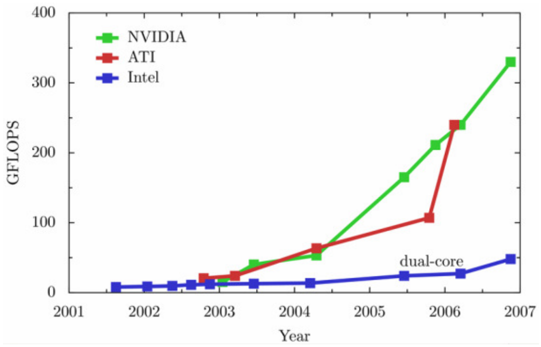

Figure 1: GPU and CPU growth in speed over the last 6 years.

In early 2007, GPU manufactures such as NVIDIA and AMD/ATI have released a type of *compute coprocessors* that have exposed certain functionalities of the graphics device through the use of new device models accessible via common low-level programming languages. The AMD/ATI Firestream [3] GPUs are programmable through an open-source C-based language *Brook+* and allow access to the Compute Abstraction Layer (CAL), a low level access to the GPU [4]. On the NVIDIA side, recent advances of its own C-supported programming model (Compute Unified Device Architecture or CUDA) [6] have yielded promising results in terms of speedups and programmability over previous generations. We have chosen the ladder architecture for development, and subsequently use NVIDIA graphics boards (G80 GPU series) for testing.

While NVIDIA's NVCC compiler for CUDA allows for compilation of host and kernel C code, its purpose is best suited for generating optimized programs on the GPU. This is ideal in the development of small applications and subroutines but not for large-scale programs. Nevertheless, GPUs should be considered as compute coprocessors capable of processing large data sets provided from a CPU host cluster. In [8], a 30 GPU cluster simulated a large scale airborne dispersion event using the lattice Boltzmann model. The implementation made use of CG [7], a high-level shading language, pixel-buffer ping-pong techniques, and OpenGL in C. However, familiarity with these specialized methods was a prerequisite to using such a system. To solve this problem, the middleware library that we developed adds GPU functionality to a high-level language, Fortran 95. The middleware library contains functions that allow the Fortran user to directly access and manipulate data on the GPU, perform a host of common mathematical operations on the data set, and add new kernel code into the library without exposing the core architecture of the GPU or necessitating background knowledge of traditional GPGPU computing methods. This middleware library thereby serves as an abstraction layer between the low-level CUDA model and the high-level Fortran applications used in scientific computing.

2 Background

In the CUDA model, the GPU is a highly multithreaded parallel processing device. Threads are defined in batches within thread blocks which are executed on individual multiprocessors on the GPU. Each multiprocessor contains a shared pool of memory that allows the threads of a block constant time (4 cycles) read and write access. A much larger pool of global and texture memory located on the graphical device is accessible to all threads across all blocks but with longer read and write access costs (400-600 cycles). Threads within the same thread block follow the principles of Simultaneous Instruction Multiple Data (SIMD) to achieve parallel data processing. Threads will also remain idle as they wait for other threads in the block to complete their instructions up to a synchronization point if divergence or branching occurs. Each thread is assigned a unique thread ID and blocks with a block ID for referencing in the code. Additional considerations such as shared memory allocation per thread block, local memory allocation per thread, number of threads per block, and global memory access all affect the overall efficiency of the program.

3 Design

The DevObject middleware is designed to prioritize ease of use for Fortran 95 users and to establish an efficient data access paradigm between the CPU and GPU hardware. The DevObject application model is divided into three layers: Fortran interface, C interface, and the CUDA kernels. While the Fortran interface is visible to the user, the C environment is not and is responsible for the management of the DevObject library itself. Typical Fortran function calls in DevObject are wrapper functions to C code which are supported by the CUDA architecture. Kernel functions, as well as NVIDIA included CUBLAS and CUFFT library functions are made accessible to the Fortran user through this process.

3.1 Framework

The upper level of the DevObject Fortran framework defines a data structure (devVar or device variable), that encapsulates a number of parameters associated with the device memory addresses, dimensions of data, alignment information, and memory allocation status. In practice, device variables, which are stored on the Fortran host side, allow for data migration between the GPU and CPU in vector and matrix storage formats. Disparities between the memory formats on the host and device are addressed in the next section.

DevObject Fortran implements a number of wrapper functions to interface with C level code. Included in the C level are function calls used to access particular CUDA kernels as well as loader functions used to run external modules (*cubin* files) or user created CUDA functions.

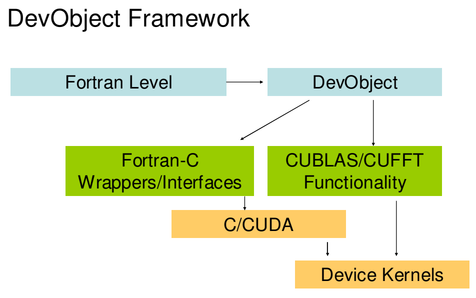

Figure 2: The DevObject framework contains three layers of interaction; the Fortran layer interacts with the DevObject library that interfaces with lower level C and CUBLAS functions which in turn allows operability to CUDA device kernels.

The DevObject library also has direct access to the released single-precision CUBLAS and CUFFT library functions. Similarly, DevObject uses a combination of Fortran wrappers and device variables to make successive calls to these libraries.

3.2 Memory Model

One notable disparity between Fortran and C is its use of different storage formats for multi-dimension arrays in linear memory. Fortran's column-major format conflicts with C and CUDA memory row-major storage formats. The solution presented itself in the CUBLAS library; the library was originally developed to interface with the Fortran language and consequently uses the same column-major 1-base indexing format. DevObject integrates these CUBLAS functions into its memory management process.

DevObject's memory model is optimized for fast processing on the GPU device. A number of considerations to CUDA's memory architecture are addressed so that proper data alignment, grid/block/thread sizes, and shared memory constraints are met prior to kernel execution. In terms of the memory model, DevObject forces the dimensions of data elements on the device to align with multiples of 256 for vectors, 16x16 for 2-D arrays, and 8x8x4 for 3-D matrices. No data padding is used due to the large overhead costs of realignment between memory transfers.

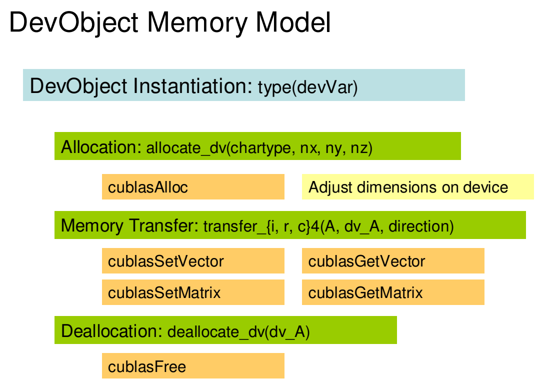
DevObject Memory Model
DevObject Instantiation: type(devVar)
Allocation: allocate_dv(chartype, nx, ny, nz)
cublasAlloc
Adjust dimensions on device
Memory Transfer: transfer_{i, r, c}4(A, dv_A, direction)
cublasSetVector
cublasGetVector
cublasSetMatrix
cublasGetMatrix
Deallocation: deallocate_dv(dv_A)
cublasFree

Figure 3: The memory model is based on the CUBLAS column-major storage and 1-based indexing. A number of helper functions from the CUBLAS library are used to facilitate data allocation and transfer on the GPU device.

The device variable encapsulates a number of parameters and attributes of the data structure transferred between host and device. The user is able to instantiate device variables in the Fortran code once the library is loaded. Use of DevObject's memory functions allow the device variables to track newly allocated space on the GPU, pass data between Fortran arrays and the device memory, and free up device variables or device memory when more space is needed.

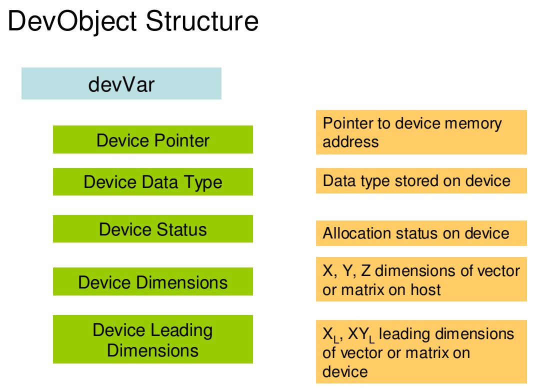
DevObject Structure
devVar
Device Pointer
Pointer to device memory address
Device Data Type
Data type stored on device
Device Status
Allocation status on device
Device Dimensions
X, Y, Z dimensions of vector or matrix on host
Device Leading Dimensions
X_L, XY_L leading dimensions of vector or matrix on device

Figure 4: The device variable data structure contains explicit attributes regarding its data type and dimensions as well as the memory address on the GPU device and memory allocation status.

3.3 CUDA Runtime and Driver API

The DevObject library concurrently uses the CUDA driver and runtime APIs. The low-level driver API is able to link to both generated host code or execute external *cubin* objects on the device. The high-level runtime API is only able to link to host code for execution. Aside from the aforementioned differences, both APIs have otherwise the same capabilities in terms of kernel speed, interoperability with other 3rd party APIs such as OpenGL and DirectX, and execution model.

The use of the two CUDA APIs addresses concerns regarding DevObject's development and ease of use. Compared to the Driver API, the runtime API contains a simplified kernel execution procedure with its predefined implicit initialization, context management, and module management via configuration syntax. Furthermore, all kernel host code generated by the NVCC compiler is based off CUDA runtime so; the driver API that links to kernel code compiled on the host may not be streamlined. The driver API is language independent but requires explicit configurations and kernel parameters for kernel launches. Additionally, it does not provide kernel or software emulation for debugging purposes.

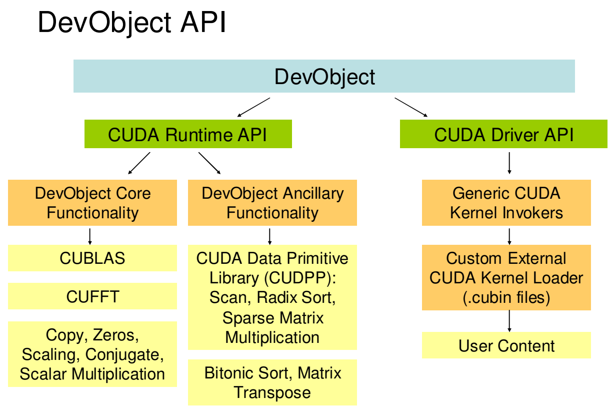
DevObject API
DevObject
CUDA Runtime API
CUDA Driver API
DevObject Core Functionality
DevObject Ancillary Functionality
Generic CUDA Kernel Invokers
CUBLAS
CUFFT
Copy, Zeros, Scaling, Conjugate, Scalar Multiplication
CUDA Data Primitive Library (CUDPP): Scan, Radix Sort, Sparse Matrix Multiplication
Bitonic Sort, Matrix Transpose
Custom External CUDA Kernel Loader (.cubin files)
User Content

Figure 5: DevObject uses both the CUDA runtime and driver API. The runtime API provides a more rigid yet manageable access to the integrated components of the DevObject library. The Driver API is used as an extension for calling user created external functions (within *cubin* files) without the need to directly interface with the C code base.

DevObject's use of both APIs gives it a flexible framework for managing and launching its own internal functions while providing options for external development and incorporation of custom CUDA functions. The runtime API manages all the core and ancillary functionalities related to the CUBLAS/CUFFT libraries, memory control, and data scaling. The ancillary branch includes additional integrated libraries such as the CUDA Data Parallel Primitives (CUDPP) and other matrix based operations.

The driver API is designed to load and execute *cubin* objects that are written externally and imported by the user. Module management of multiple *cubin* objects is done on the C level and remains independent of any Fortran processes. The DevObject library also contains a number of generic function invokers, callable from the Fortran that pre-computes the kernel block-shape, passes the necessary arguments to the kernel object, and launches the grid for execution. User developed CUDA functions that follow a particular parameters format are thereby executable by our kernel caller.

3.4 Work-Cycle

The DevObject work-cycle begins by compiling and linking our library with any existing Fortran code. To use the library, the library's initialization function must be called. Here, the particular GPU device is set for use and the driver API context is created. Once the library is loaded for use, DevObject subroutines and functions can be called from any Fortran environment.

Device variables, which are DevObject's functional link between host and device, are created by the user. The data type, dimension, and size of the device variables are all specified by user parameters and dynamically allocated in the GPU global memory space. Any number of device variables can be created in this fashion while memory is available on the GPU. DevObject will not garbage collect unused device variables or device memory so an explicit deallocation subroutine must be called to free additional space. Once the device variables are allocated, host data can be quickly copied onto the device.

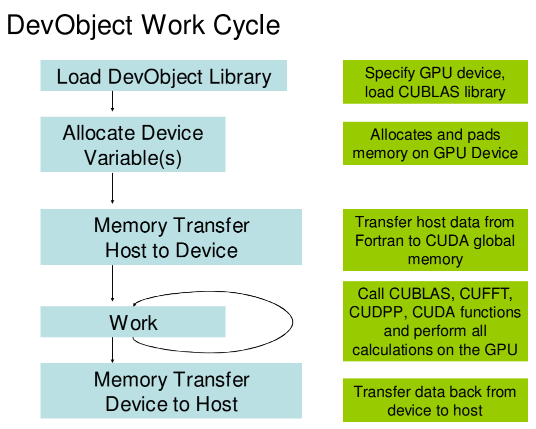
DevObject Work Cycle
Load DevObject Library
Allocate Device Variable(s)
Memory Transfer Host to Device
Work
Memory Transfer Device to Host
Specify GPU device, load CUBLAS library
Allocates and pads memory on GPU Device
Transfer host data from Fortran to CUDA global memory
Call CUBLAS, CUFFT, CUDPP, CUDA functions and perform all calculations on the GPU
Transfer data back from device to host

Figure 6: The Fortran-DevObject work cycle allows for data computation to occur solely on the GPU without write-backs to the host CPU.

The work-cycle primarily consists of calling subsequent CUDA function kernels to perform operations on the GPU data. The advantage of this method lays in the faster parallel computation of large data sets as a result of asynchronous kernel executions and the preclusion of any memory transferring operations between the host and device during this time.

3.5 Usage

All DevObject library functions are placed in sub-modules defined under the main module devObject. To access the DevObject library functions in Fortran, include or *use* devObject in any subroutine or module and compile the code with the library.

The library functions become available only after a call to subroutine open_devObjects. Likewise, a call to close_devObjects will cease all of DevObject's functionalities. These subroutines also initialize and close the CUBLAS library so after the call open_devObjects or close_devObjects, all CUBLAS functions become available or unavailable so additional calls to CUBLAS_INIT and CUBLAS_SHUTDOWN are unnecessary.

Fortran 95 allows operator overloading between derived types. The DevObject library has overloaded its devVar structure with custom point-wise operations that are used to evaluate expressions assigned to other device variables. The evaluation function and memory management of intermediate computations of expressions run on Fortran's implicit evaluation stack. Intermediate calculations are carried out in temporarily allocated device variables on this stack and deallocated while parts of the expression are evaluated and concatenated. This is to ensure that other device variables defined by the user are not affected in the process.

```
!-----------------------main program
  use devObject
  implicit none
  !----------------------Executables
  call open_devObjects
  !...
  call usersub
  !...
  call close_devObjects
  end
!-----------------------end of main

subroutine usersub
    use devObject
    implicit none

    integer,parameter:: size=1024
    type(devVar) dv1,dv2,dv3,dv4

    real ar1d1(size),ar1d2(size),ar1d3(size), ar1dresult(size)

    call random_number(ar1d1)
    call random_number(ar1d2)
    call random_number(ar1d3)

    dv1=allocate_dv('real',size)
    dv2=allocate_dv('real',size)
    dv3=allocate_dv('real',size)

    call transfer_r4(ar1d1,dv1,.true.)
    call transfer_r4(ar1d2,dv2,.true.)
    call transfer_r4(ar1d3,dv3,.true.)

    !pointwise mulitplication division addition subtraction test
    dv4=(3.14159*dv1*(dv2+dv1)*(dv1-(.553+dv3))*dv2+(-.244))/dv1

    call transfer_r4(ar1dresult,dv4,.false.)

    call deallocate_dv(dv1)
    call deallocate_dv(dv2)
    call deallocate_dv(dv3)
    call deallocate_dv(dv4)
end subroutine usersub
```

Figure 7: Example of Fortran code and device variable point-wise expression evaluations on 4 arrays of type real and dimensions 1024x1024

4 Applications

The DevObject library is extensible to many fields in mathematics and physics. In [5], an application of plasma turbulence using the CUFFT library achieved a speedup of 25 over the native Fortran code having been ported to the DevObject library. Another application of radial basis fitting to scattered data using the iterative [1] achieved a remarkable speedup of 662 over the serial CPU code. Here, we present a simple mathematical application of the Mandelbrot set [2] as defined by the set of complex numbers c such that the complex quadratic polynomial $z_{i+1} = z_i^{\,2} + c$ remains bounded after a finite number of iterations n specified by the user.

4.1 Implementation

An implementation of the Mandelbrot code is developed in CUDA and callable as a custom kernel function by DevObject for the GPU. The function takes as input the domain of the Mandelbrot set, the number of iterations until convergence, and the image dimensions for the output. The output format consists of densely packed 32bit integer channels representing RGB.

```
 __global__ void MandelbrotGPU1(unsigned int *rgb,
                               int iWidth,int iHeight,int maxIterations,
                               float realMin,float cmplxMin,
                               float realScal,float cmplxScal){

     unsigned char tx=threadIdx.x;
     unsigned char ty=threadIdx.y;
     unsigned short int bx=blockIdx.x;
     unsigned short int by=blockIdx.y;
     unsigned short int x=BLOCK_DIM_2D_X*bx+tx;
     unsigned short int y=BLOCK_DIM_2D_Y*by+ty;


     //Setup mandelbrot set z and constant c
     Complex z={0,0};
     Complex c={realMin+realScal*y,cmplxMin+cmplxScal*x};
     unsigned int convergence=0;

     do{//Compute mandelbrot for at most maxIterations
         z=ComplexAdd(ComplexMul(z,z),c);
     }while(z.x*z.x+z.y*z.y<=4&&(++convergence)<maxIterations);

     /*
     Coalesed writes:
     Convert convergence iterations to colors and store in shared memory
     */
     __shared__ unsigned int smem[BLOCK_DIM_2D_Y][BLOCK_DIM_2D_X][3];
     smem[ty][tx][0]=(convergence==maxIterations)?255:((convergence*17)&255);
     smem[ty][tx][1]=(convergence==maxIterations)?0:((convergence*13)&255);
     smem[ty][tx][2]=(convergence==maxIterations)?0:((convergence*7)&255);

     //Write the output to memory
     unsigned int ref=3*(y*iWidth+bx*BLOCK_DIM_2D_X);
     __syncthreads();
     rgb[ref+tx]=*(smem[ty][0]+tx);
     rgb[ref+BLOCK_DIM_2D_X+tx]=*(smem[ty][0]+BLOCK_DIM_2D_X+tx);
     rgb[ref+2*BLOCK_DIM_2D_X+tx]=*(smem[ty][0]+2*BLOCK_DIM_2D_X+tx);
 }
```

Figure 8: The kernel code used for computing the Mandelbrot set.

The Mandelbrot set Z and constants C are setup by a thread by pixel mapping. Each thread corresponds to a discrete position scaled along the domain and maps to a pixel in the final output. Computation of the set is done as each thread computes the next element in its series until numeric divergence occurs. Divergence of the Mandelbrot set is defined when the Z value's norm is greater than 2. Convergence is defined as having not diverged after *n* iterations. The final coloring is a function of the number of iterations before divergence.

4.2 Results

The same implementation is reproduced in a serialized Fortran algorithm that runs on the CPU. The timings from both the DevObject and native Fortran sources are recorded and compared. For a control, the CUDA Mandelbrot code from the NVIDIA SDK provides a lower bound for kernel runtime on the device. For specificity, the hardware used is given below.

• CPU: Intel Core2 CPU 6600 @ 2.40 GHz, 2 GB RAM
• GPU: NVIDIA GeForce 8800 GTX, 768 MB global memory (not over-clocked)

• Single Precision, Domain of Real in [-2.1, 1.1], Complex in [-1.6, 1.6], Bounded iterations 512, timing in seconds

Image Size	4096x4096	2048x2048	1024x1024
DevObject Kernel	0.11232	0.03125	0.01054
DevObject + Memory Operations:	0.28125	0.07812	0.01562
Fortran	44.5000	11.14062	2.78125
NVIDIA	0.09985	0.03315	0.01652

Table 1: Mandelbrot Results: DevObject kernel refers to the kernel processing time, DevObject + Memory Operations include times from memory allocation and transferring operations from the host to the device, Fortran refers to times produced by the CPU algorithm, and NVIDIA refers to the kernel times produced from the Mandelbrot sample code in the SDK.

A speedup of over 140X is found between the Fortran and DevObject Mandelbrot implementations. The DevObject kernel runtimes are nearly identical to that of SDK's due to the similarities in the code. The cost of memory allocation and transfer are nontrivial as the runtimes scale non-linearly as the dimension sizes increase.

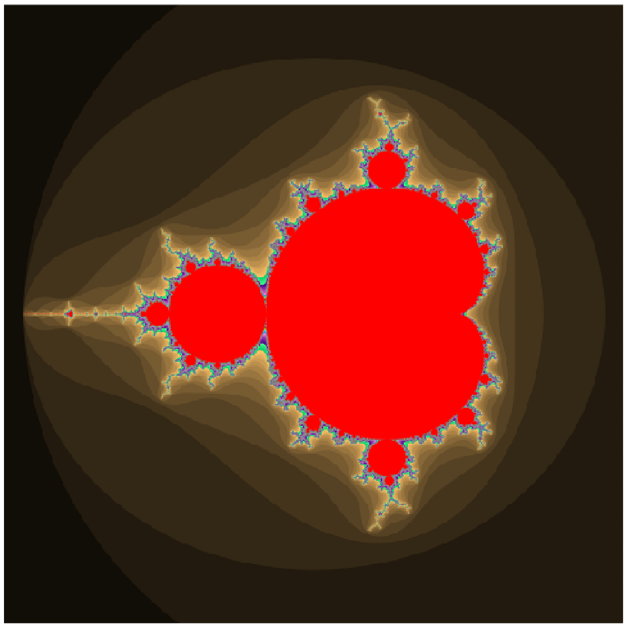

Figure 9: The Mandelbrot set from domain R in [-2.1, 1.1], C in [-1.6, 1.6] generated by DevObject for image size 4096x4096 after 512 iterations in .28125 seconds.

5 Conclusions

We have demonstrated that a middleware over Fortran can efficiently interfaces with the CUDA architecture on NVIDIA GPUs. DevObject's memory model allows pointers to device memory to be managed on the host within an encapsulated device variable. DevObject's callable functions in Fortran allow operations to be performed on the device variables that in turn operate on the device memory. The two runtime and driver APIs preserve core functionalities within the library while allowing custom CUDA code to be interfaced with the library. Finally, a sample application of the Mandelbrot set is shown to have significant speedups when computed with our middleware. We plan to add new applications of numerical analysis as we expand DevObject's functionalities.

6 References

[1] A. Faul, G. Goodsell, M.J. Powell, "A Krylov subspace algorithm for multiquadric interpolation in many dimensions," IMA J. Numer. Anal., 25, 1-24, 2005.

[2] Benoit B. Mandelbrot. "Fractal aspects of the iteration of $z \mapsto \lambda z(1-z)$ for complex λ and *z"*. In R. H. G. Helleman, editor, Non-Linear Dynamics*, volume 357 of* Annals of the New York Academy of Sciences, pages 249-259. New York Academy of Sciences, 1980.

[3] http://ati.amd.com/products/streamprocessor/specs.html

[4] http://ati.amd.com/technology/streamcomputing/faq.html

[5] N.A. Gumerov, R. Duraiswami, and W. Dorland. "Middle-ware for programming NVIDIA GPUs from Fortran 9X"., Supercomputing 2007.

[6] "NVIDIA Cuda Compute Unified Device Architecture Programming Guide, V2.0", NVIDIA Corp. 2008.

[7] W. R. Mark, R. S. Glanville, K. Akeley, and M. J. Kilgard. "Cg: a system for programming graphics hardware in a C-like language". ACM Trans. Graph. (SIGGRAPH), 22(3):896–907, 2003.

[8] Zhe Fan, Feng Qiu, Arie Kaufman, Suzanne Yoakum-Stover. "GPU Cluster for High Performance Computing". Proceedings of Supercomputing, 2004.