1. INTRODUCTION

There is a revolution taking place in computing with the availability of commodity graphical processors capable of several hundred gigaflop performance and with multicore CPUs. Using these it should be possible to have relatively inexpensive teraflop workstations. These machines however are not so easily programmed, and scientific computing may not map on to them.

Over the past year, the major graphics vendors have realized there is a market for these in HPC, and have started producing tools to allow programming them from high level languages, ATI’s Close-To-Machine (CTM) and NVIDIA’s CUDA [1]. We are more familiar with the latter architecture, for which both a beta middleware function calls. Fig. 3 shows the speedup on this code, stepping loop. Fig. 2 shows the translation of this routine using code is that evaluating nonlinear evolution terms in a time

2. GPU PROGRAMMING MIDDLEWARE

CUDA views the GPU as a set of multiprocessors, each with some local cache memory (of various types), and all able to talk to global device memory. Because of the architecture, applications that map to the shared multiprocessor parallel environment, effectively use the available number of processors, and mostly use cache memory, can see significant speedups.

While the NVCC compiler provided with CUDA does compile host C code, it is mostly focused at producing software that runs on the GPU. While this is useful to develop small programs to run on the GPU, when GPUs will be used for high performance computing they should be more properly viewed as compute coprocessors, to which data from a large program running on the CPU host/cluster is farmed out. Of course, since host-GPU communication is relatively slow, back and forth data exchange should be avoided. Instead, our viewpoint of GPU programming is to provide a high level language such as Fortran 9X with a set of functions that give it the ability to manipulate data on the GPU via a middleware library, and augment the middleware functions with a small number of problem specific functions written in CU.

With the understanding that the performance of the GPU substantially depends on the thread block size, number of multiprocessors employed, and even position of the elements in the arrays, we introduce the concept of device variables, which have sizes and allocations on the GPU to provide high performance operations. We implement device variables as structures, which encapsulate information about the pointer, size, and other parameters, e.g., the type, dimension, leading dims, allocation status, etc. Fortran modules allow wrapping of function calls. Overloaded functions suitable for the use with different types, shapes, and optional parameters are developed. Several device functions, callable via wrappers are also provided. These are for initializing variables, copying them, and performing other operations. The NVIDIA provided CUBLAS/CUFFT functions are also encapsulated in a convenient overloaded syntax, which avoids bugs due to calling errors.

3. RESULTS & CONCLUSIONS

We accelerated two sample sci. comp. applications. In each case, we had original Fortran 90 code available, and we translated this original code to run on the GPU. The first application was from plasma turbulence using a simplified but relevant 2D model [3]. This is a pseudospectral code that makes use of the wrapped CUFFT library. Computationally the most important part of this code is that evaluating nonlinear evolution terms in a time stepping loop. Fig. 2 shows the translation of this routine using middleware function calls. Fig. 3 shows the speedup on this code, for N=16, ..., 1024. A speedup of about 25 is achieved vis-à-vis the serial CPU code, executed on an Intel XQ6700 CPU processor and the expected scaling $N^2 \log N$ is seen.

The second application is from the fitting of radial basis functions to scattered data, using an iterative algorithm [2]. This is representative of many applications in iterative methods that should see significant speedups. Here an incredible speedup of 662 times over a serial CPU code is seen.

We are continuing to extend this environment, and have recently used it to develop a version of the FMM on the GPU [4]. In the future, programming of multiple GPUs, and shared computation on distributed CPUs and GPUs will be developed.


Fig. 1: GPU and CPU growth in speed over the last 6 years.
Fig. 2: Example of code conversion using Fantalgo’s middleware: Left: original Fortran-90 code; Right: ported code.

Original Fortran Code

```fortran
subroutine nltterms(isave,a,b,nl)
use com_module

complex,dimension(:,:),intent(in) :: a,b
complex,dimension(:,:),intent(out):: nl
integer :: isave

if(isave /= 1) then
    kxa2=ikx*a
    kya2=iky*a
    call fftwd_f77_one(plan_f,kxa2,kxa2)
    call fftwd_f77_one(plan_f,kya2,kya2)
endif

if(isave /= 2) then
    kxb2=ikx*b
    kyb2=iky*b
    call fftwd_f77_one(plan_f,kxb2,kxb2)
    call fftwd_f77_one(plan_f,kyb2,kyb2)
end if

nl=kxa2*kyb2-kxb2*kya2
call fftwd_f77_one(plan_b,nl,nl)
nl=nl+scale

end subroutine nltterms
```

Modified Fortran Code

```fortran
subroutine dev_nltterms(isave,dv_a,dv_b,dv_nl)
use dv_com_module
use mod_devObject

type(dvVar),intent(in) :: dv_a,dv_b
type(dvVar),intent(out) :: dv_nl
integer :: isave

if(isave /= 1) then
    call devf_mul1(dv_kxa2,dv_kya2,dv_ikx,dv_iky,a)
    call devf_fft(dv_kxa2,fftplan)
    call devf_fft(dv_kya2,fftplan)
endif

if(isave /= 2) then
    call devf_mul1(dv_kxb2,dv_kyb2,dv_ikx,dv_iky,b)
    call devf_fft(dv_kxb2,fftplan)
    call devf_fft(dv_kyb2,fftplan)
end if

call devf_mul2(dv_nl,dv_kxa2,dv_kyb2,dv_kya2,dv_kxb2)
call devf_ifft(dv_nl,fftplan)
call devf_mul(dv_nl,dv_nl,dv_scale)

end subroutine dev_nltterms
```

Fig. 3: Acceleration of a 2D plasma turbulence code on the GPU using the developed middleware.

Fig. 4: Acceleration of a RBF fitting code using the developed middleware.