CPU computing w/ applications examples for special methods.

Threading

A thread is a set of instructions resulting from a program fork. That is, when a program creates a copy of itself, a new thread is launched.

The processor can "multithread" by time division multiplexing, switching the thread being processed. Multi-core processors will run multiple threads concurrently in separate cores.

CPU vs. GPU

The CPU through years of evolutionary pressure has been optimized for fast serial execution of tasks. Over generalizing, but true to point, a CPU will execute one set of instructions on one or two pieces of data per clock cycle. This makes sense because of the serial logic inherent in many tasks.

Co-processing of graphics through is a fundamentally parallel process. Rendering triangles, rectangles, & arcs to a computer screen...
...does not depend on all of the other
triangles, rectangles, and arcs.

The operations to render graphics primitives
involve primarily floating
point operations in single precision.

The fundamental idea is that cpus will
do one thing fast while gpus will
do many things moderately fast.

Modern CPU Architecture

- Host
  - Input ASSEMBLER
  - Thread Manager

- Data Cache
- Thread Processor

Global Memory (e.g. devices)
CUDA

The Compute Unified Device Architecture is a collection of C-extensions that allow direct access to the computing capabilities of the GPU through "device kernels."

A device kernel is a recursion & function pointer free that launches a (typically lightweight) instruction set, of several thousand threads. We will see these in the code that follows.

For general users (only looking for a 10x-20x speed up), there are BLAS & FFT libraries that optimized for device operations.

There are disadvantages though:

* Until next generation (announced this week), double precision is emulated by single precision. This is quite a bit slower than single per se.
* Double precision does not conform to IEEE standard in the last bit.
(this is also changing in next generation)

- No normals & signalling NaN's not supported.

(Biggie) The bandwidth from CPU to device in transmitting inputs of outputs is often a bottleneck.

(Biggie for Matlab users) Wrapping Matlab code requires mex interfacing which I find clunky and ugly.

Limitations of the architecture and programming model will gradually ease. This is the beginning of this methodology, it is fastly improving. Now to the code.
A basic spectral method for viscous Burgers.

1. \[ \partial_t u + u \partial_x u = \nu \Delta u \]

2. \[ u = \sum_{k=0}^{\infty} \hat{u}_k(t) e^{ikx}, \quad x \in [-\pi, \pi] \]

Subbing this into 1.

\[ \partial_t \hat{u}_k(t) + F_k(u \partial_x u) = -u_k \nu \hat{u}_k(t) \]

where \( F_k(u \partial_x u) \) is the \( k \)th Fourier coefficient of \( u \partial_x u \). We can time step by Forward - Euler yielding the numerical scheme:

\[ \hat{u}_k^{(n+1)} = -\Delta t F_k(u^{(n)}(x_k)) \frac{u^{(n)}(x_{k+1}) - u^{(n)}(x_{k-1})}{2 \Delta x} \]

\[ + (1 - \Delta t \nu k^2) \hat{u}_k^{(n)} \]

The above abuses notation a bit.

The \( F_k(\cdot) \) is a DFT of the vector defined by the discretization scheme on which it is operating.

Matlab code implementing this on the CPU is attached, the GPU implementation follows.
function burgers_cpu(N)

   nu=.1;
   tstop = 3;

   spacemesh = linspace(0,2*pi,N);
   k = [linspace(0,N/2,N/2+1) linspace(-N/2+1,-1,N/2-1)];
   initial = max(1,pi - abs(pi-spacemesh));
   u = initial;

   dx=2*pi/N;
   cfl = dx/(max(initial));
   dt=1/100*cfl;
   \%not the correct cfl condition, 1/100 admits up to N=1024, 1/200 admits up
   \%to N=2048. Beyond N=2048 has not been enabled by my choice of memory
   \%management in CUDAFFFT.
   iter_per_plot = ceil(1/100*tstop/dt); \%may be able to select this on fly as function of dt

   M = u.*(u(2:end) u(1)) - [u(end) u(1:end-1)]/(2*dx);
   uhat = fft(u);
   Mhat = fft(M);
   uhat = -dt*Mhat+(1-dt*nu*k.^2).*uhat;

   \%Calculate and plot u
   fig1=figure(1); iter=2;
   for i=2:(tstop/dt)

      u = ifft(uhat);
      M = u.*(u(2:end) u(1)) - [u(end) u(1:end-1)]/(2*dx);
      Mhat = fft(M);
      uhat = -dt*Mhat+(1-dt*nu*k.^2).*uhat;

      if i/iter == 1 \%plot ith frame of movie
         subplot(2,1,1); plot(spacemesh,real(u));
         axis([0 2*pi min(0,min(initial)) max(initial)])
         subplot(2,1,2); semilogy(spacemesh,abs(uhat)); \%plot ith frame of movie
         axis([0 pi 10^-20 100]);
         B=getframe(fig1); \%play frames in a 'movie' fashion
         iter = iter+iter_per_plot;
      end
   end
end
function burgers_gpu(N)

nu = .1;
tstop = 3;

spacemesh = linspace(0,2*pi,N);

initial = max(1, pi - abs(pi - spacemesh));
u = initial;

dx = 2*pi/N;
cfl = dx/(max(initial));
dt = 1/100*cfl;
% not the correct cfl condition, 1/100 admits up to N=1024, 1/200 admits up
% to N=2048. Beyond N=2048 has not been enabled by my choice of memory
% management in CUDAFFT.
iter_per_call = ceil(1/100*tstop/dt); % may be able to select this on fly as function of dt

% Calculate and plot u
fig1 = figure(1);
for i = 1:ceil(tstop/dt/iter_per_call)
    u = b_iter_cuda_rev(u, dx, dt, nu, iter_per_call);

    % plot ith frame of movie
    subplot(2,1,1); plot(spacemesh, real(u));
    axis([0 2*pi min(0,min(initial)) max(initial)])
    subplot(2,1,2); semilogy(spacemesh(1:N), abs(fft(u)));
    axis([0 pi 10^-10 100]);
    B = getframe(fig1); % play frames in a 'movie' fashion
end
// This is a file to iterate the 1-d viscous burger's equation in matlab

#include <stdlib.h>
#include "mex.h"
#include "cuda.h"
#include "cuda_runtime.h"
#include "cufft.h"
#include "mxcufft.h"

'* MATLAB stores complex numbers in separate arrays for the real and
imaginary parts. The following functions take the data in
this format and pack it into a complex work array, or
unpack it, respectively.
We are using cufftComplex defined in cufft.h to handle complex on Windows and Linux

static __global__ void der_M_cuda(cuftfComplex *u_in,
cuftfComplex *M_out,
float dx,
int N)
{
int index = blockDim.x*blockIdx.x+threadIdx.x;
if(index==0)
{
M_out[index].x = u_in[index].x*(u_in[1].x-u_in[N-1].x)/(2*dx);
M_out[index].y = 0;
}
if(index==(N-1))
{
M_out[index].x = u_in[index].x*(u_in[0].x-u_in[N-2].x)/(2*dx);
M_out[index].y = 0;
}
if(0<index && index<(N-1))
{
M_out[index].x = u_in[index].x*(u_in[index+1].x-u_in[index-1].x)/(2*dx);
M_out[index].y = 0;
}

static __global__ void step_u_cuda(cuftfComplex *u_hat,
cuftfComplex *M_hat,
float dt,
float nu,
int N)
{
int index = blockDim.x*blockIdx.x+threadIdx.x;
float k;
if(index<N/2)
{
k = (float) index;
}
else
{

k = (float) index - N;

if(index<N)
{
    u_hat[index].x = (-dt*(M_hat[index].x)+(1-dt*nu*k*k)*(u_hat[index].x));
    u_hat[index].y = (-dt*(M_hat[index].y)+(1-dt*nu*k*k)*(u_hat[index].y));
}

static __global__ void norm_u_cuda(cufftComplex *u_in,
    cufftComplex *u_out,
    int N)
{
    int index = blockDim.x*blockIdx.x+threadIdx.x;
    if(index<N)
    {
        u_out[index].x = u_in[index].x/N;
        u_out[index].y = u_in[index].y/N;
    }
}

/*%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%*/

void mexFunction( int nlhs, mxArray *plhs[],
    int nrhs, const mxArray *prhs[])
{
    int N;
    double *ar,*ai,*dx_double, *dt_double, *nu_double, *iterations_double;
    cufftComplex *input_single ;
    cufftHandle plan;

    /*
    Find out the dimension of the array we want to transform:
    */
    originalN = Number of columns in the mxArray prhs
    */
    N = mxGetN(prhs[0]);

    /*
    Find out if the input array was real or complex.
    Matlab is passing two separate pointers for the real and imaginary part.
    The current version of CUDAFFT is expecting interleaved data.
    We will need to pack and unpack the data.
    */
    /*
    The current version of CUDAFFT supports only single precision,
    so the original double precision data need to be converted.
    */
    /* Allocating working array on host */
    input_single = (cufftComplex*) mxMalloc(sizeof(cufftComplex)*N);


/* Get pointers to the matlab arrays */

ar = (double *) mxGetData(prhs[0]);
if(mxIsComplex(prhs[0]))
{
    /* The input array is complex */
    ai = (double *) mxGetImagData(prhs[0]);

    /* Input and output have same dimensions */
    pack_c2c(input_single, ar, ai, N);
}

else
{
    /* The input array is real */

    /* Input and output have same dimensions */
    pack_r2c(input_single, ar, N);
}

dx_double = (double *) mxGetData(prhs[1]);
dt_double = (double *) mxGetData(prhs[2]);
u_double = (double *) mxGetData(prhs[3]);
iterations_double = (double *) mxGetData(prhs[4]);

/* Kernels don't like doubles, cast to float and int */
float dx = (float) dx_double[0];
float dt = (float) dt_double[0];
float nu = (float) nu_double[0];
int iterations = (int) iterations_double[0];

cudaMalloc((void **) &u, sizeof(cufftComplex)*N);
cudaMalloc((void **) &what, sizeof(cufftComplex)*N);
cudaMalloc((void **) &M, sizeof(cufftComplex)*N);
cudaMalloc((void **) &Mhat, sizeof(cufftComplex)*N);

/* Copy input array in interleaved format to the device */
cudaMemcpy(u, input_single, sizeof(cufftComplex)*N, cudaMemcpyHostToDevice);

/* Set execution configuration */
int block_size = 16;
dim3 dimBlock(block_size,1);
dim3 dimGrid(128,1);

/* Calculate M as a function of u */
der_M_cuda<<dimGrid,dimBlock>>>(u, M, dx, N);

// Create plan for CUDA FFT
cufftPlan1d(&plan, N, CUFFT_C2C,1);

// Execute FFT on device
cufftExecC2C(plan, u, what, CUFFT_FORWARD);
cufftExecC2C(plan, M, Mhat, CUFFT_FORWARD);
step_u_cuda<<<dimGrid, dimBlock>>>(uhat, Mhat, dt, nu, N);

int i;
for(i=1; i<iterations; i++) {
    //norm_u_cuda<<<dimGrid, dimBlock>>>(uhat, what_to_u, N); // what_to_u should then be used for calculating u
    //in next line
    cufftExecC2C(plan, uhat, u, CUFFT_INVERSE);
    norm_u_cuda<<<dimGrid, dimBlock>>>(u, u, N); //normalizing after inverse admits overflow for n>2048
    der_M_cuda<<<dimGrid, dimBlock>>>(u, M, dx, N);
    cufftExecC2C(plan, M, Mhat, CUFFT_FORWARD);
    step_u_cuda<<<dimGrid, dimBlock>>>(uhat, Mhat, dt, nu, N);
}

cufftExecC2C(plan, uhat, u, CUFFT_INVERSE);
norm_u_cuda<<<dimGrid, dimBlock>>>(u, u, N);

// Destroy plan
    cufftDestroy(plan);

// Copy result back to host
    cudaMemcpy(input_single, u, sizeof(cufftComplex)*N, cudaMemcpyDeviceToHost);

plhs[0]=mxCreateDoubleMatrix(1, N, mxCOMPLEX);
ar = mxGetPr(plhs[0]);
ai = mxGetPi(plhs[0]);
unpack_c2c(input_single, ar, ai, N);
mxFree(input_single);
cudaFree(u);
cudaFree(uhat);
cudaFree(M);
cudaFree(Mhat);
return;
Implementation of Spectral Method Grows Slower in Compute Time

- Time to compute on CPU
- Time to Compute on GPU