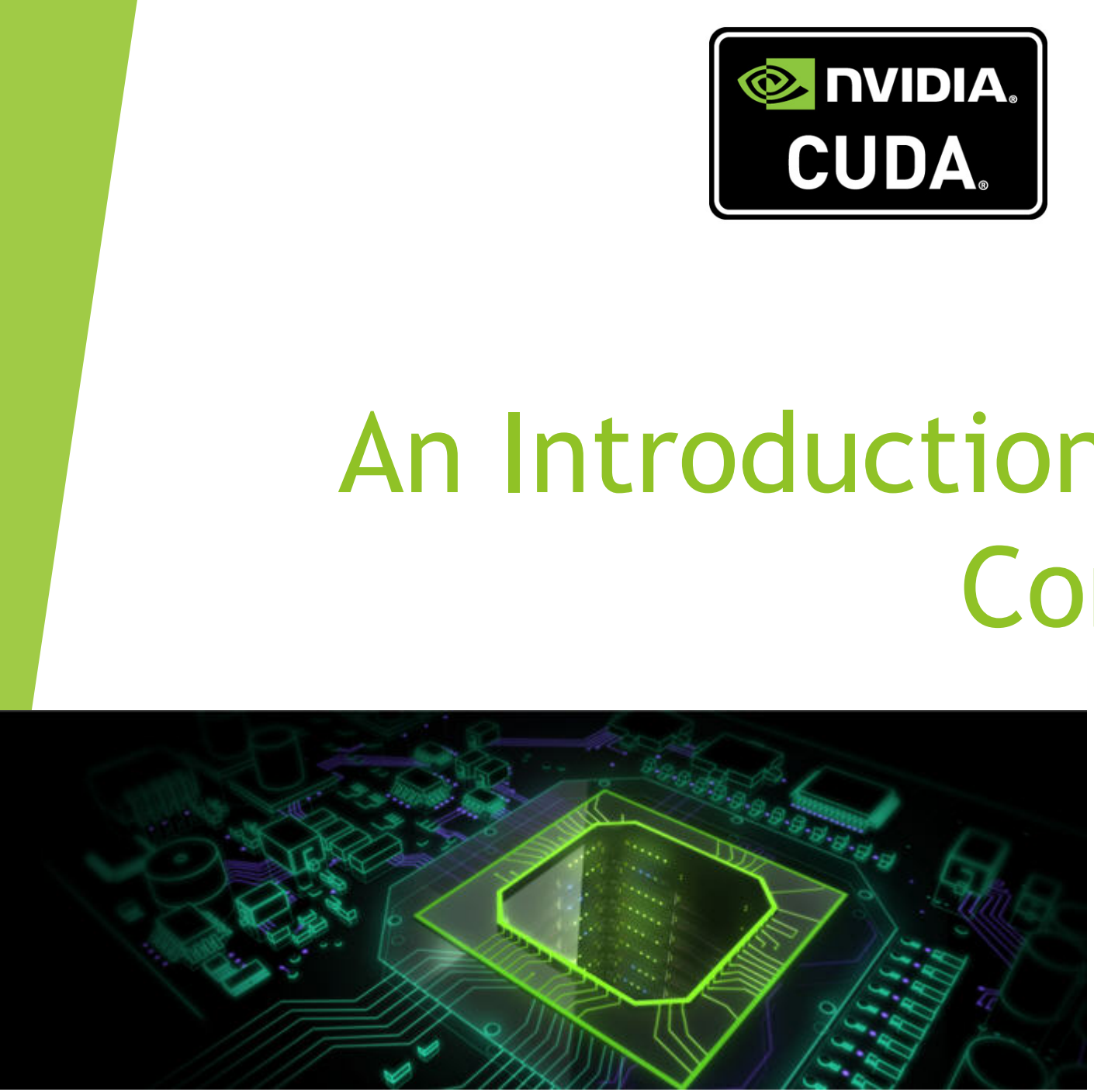


An Introduction to GPU Computing

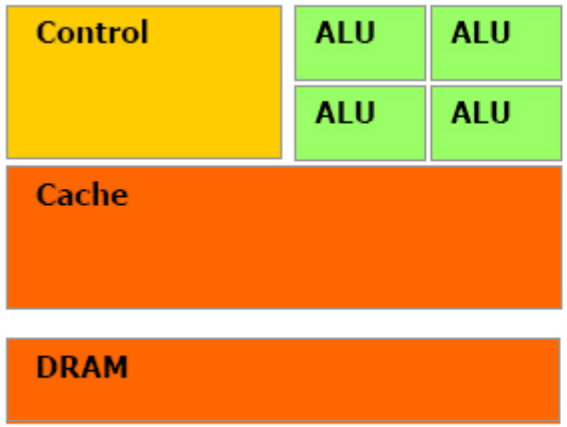
Aaron Coutino

acoutino@uwaterloo.ca

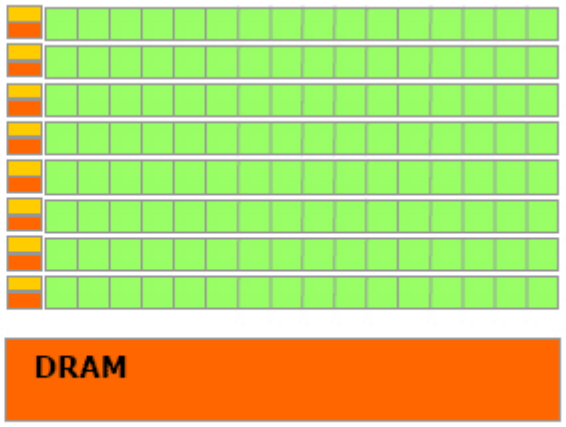
MFCF

What is a GPU?

- A GPU (Graphical Processing Unit) is a special type of processor that was designed to render and manipulate textures. They were originally designed for rendering video games.
- The key differences between a CPU and a GPU is how many transistors are allocated to processing compared to memory.



CPU



GPU

GPU Computing Applications

Libraries and Middleware						
cuFFT cuBLAS cuRAND cuSPARSE	CULA MAGMA	Thrust NPP	VSIPL SVM OpenCurrent	PhysX OptiX iRay	cuDNN TensorRT	MATLAB Mathematica

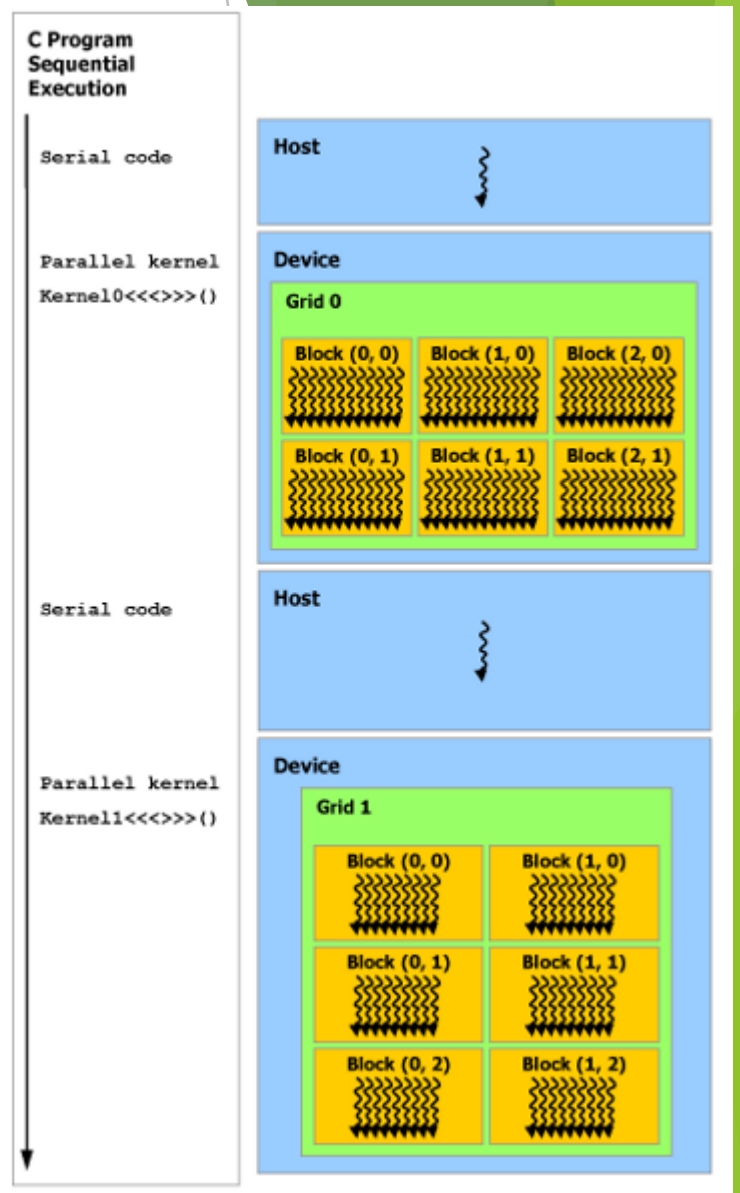
Programming Languages					
C	C++	Fortran	Java Python Wrappers	DirectCompute	Directives (e.g. OpenACC)

CUDA-enabled NVIDIA GPUs

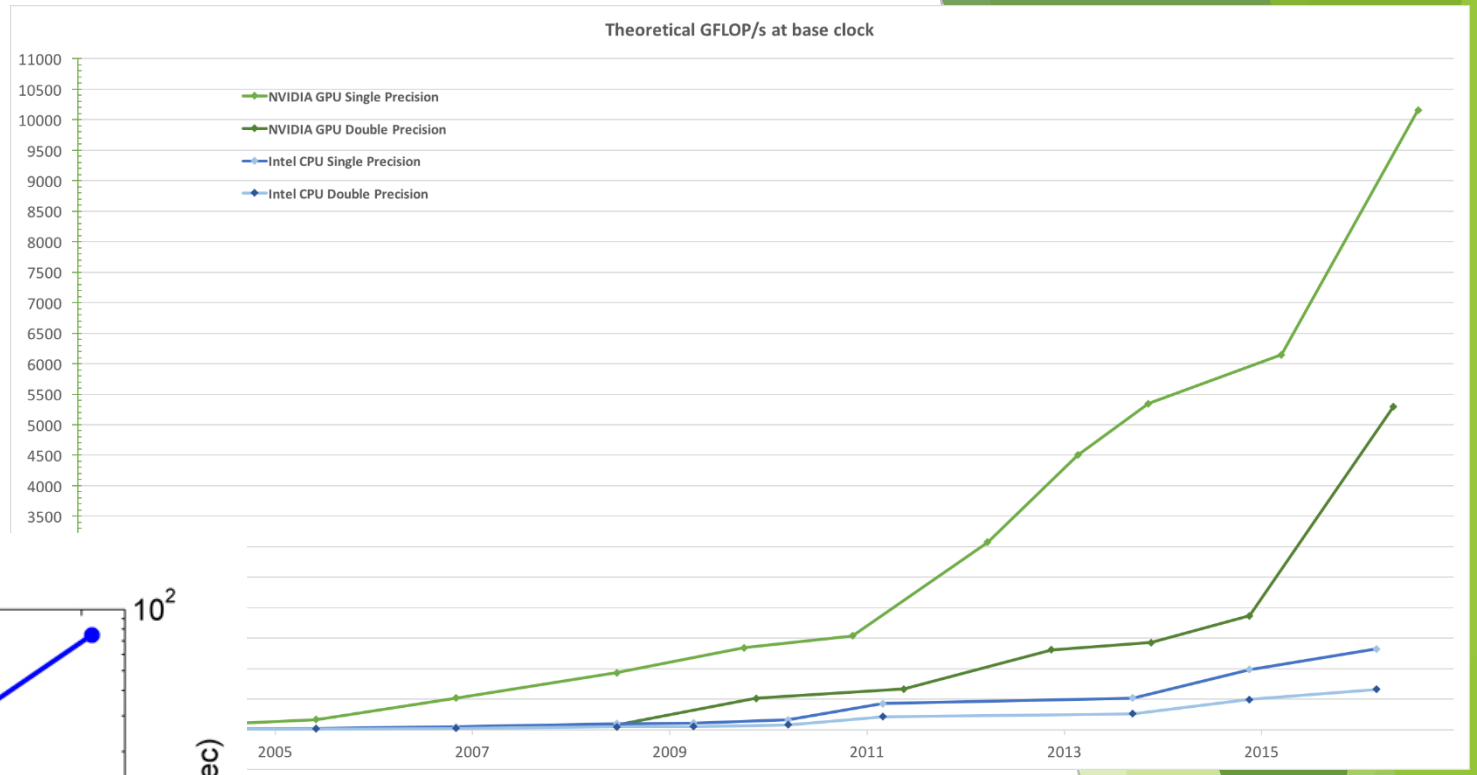
Pascal Architecture (compute capabilities 6.x)	GeForce 1000 Series	Quadro P Series	Tesla P Series
Maxwell Architecture (compute capabilities 5.x)	GeForce 900 Series	Quadro M Series	Tesla M Series
Kepler Architecture (compute capabilities 3.x)	GeForce 700 Series GeForce 600 Series	Quadro K Series	Tesla K Series
Fermi Architecture (compute capabilities 2.x)	GeForce 500 Series GeForce 400 Series	Quadro Fermi Series	Tesla 20 Series
			

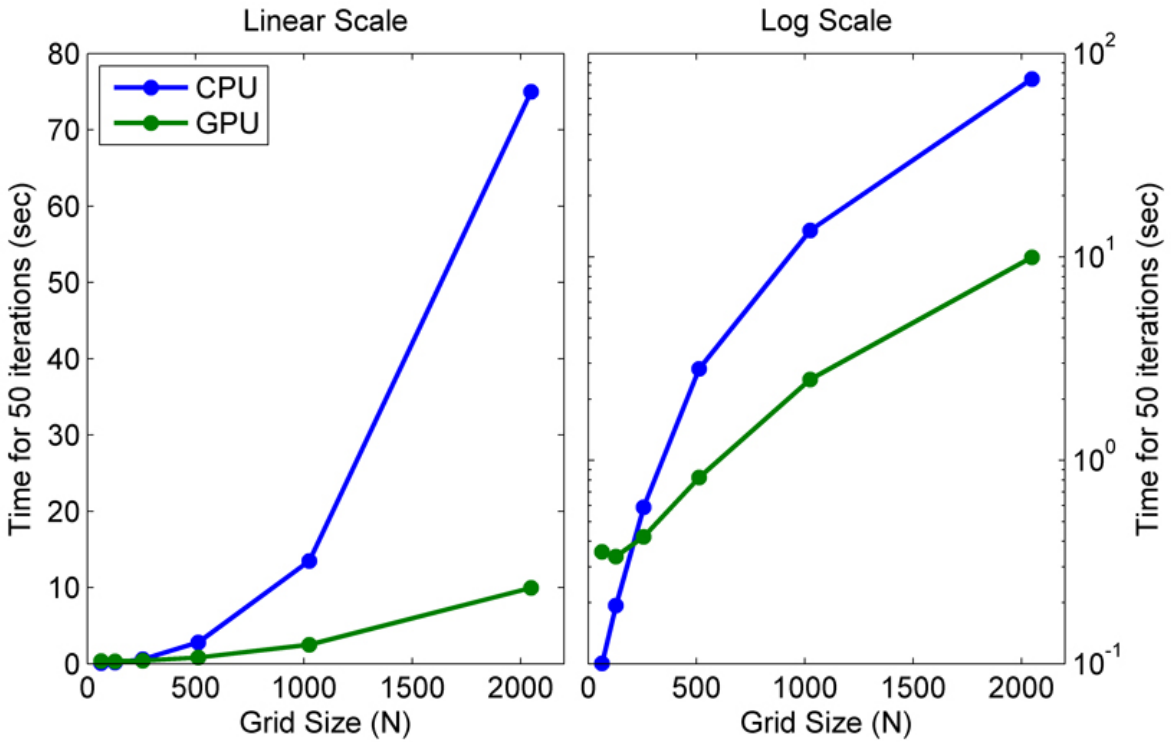
Why to use a GPU

- While originally designed for graphical processing, the architecture of GPUs are suited for any application which perform many of the same simple task.
- In 2008, Nvidia developed the first general purpose API for interfacing with GPUs, CUDA. This has made it easy to interface with the GPU and take advantage of this type of computation.
- The figure on the right shows a typical work flow for a code utilizing a GPU.
- When used correctly GPU computing can provide a massive speed-up.



Speed up


Theoretical GFLOP/s at base clock
NVIDIA GPU Single Precision
NVIDIA GPU Double Precision
Intel CPU Single Precision
Intel CPU Double Precision
11000
10500
10000
9500
9000
8500
8000
7500
7000
6500
6000
5500
5000
4500
4000
3500
2005
2007
2009
2011
2013
2015


Linear Scale
Log Scale
CPU
GPU
Time for 50 iterations (sec)
Grid Size (N)
0
10
20
30
40
50
60
70
80
500
1000
1500
2000
10^2
10^1
10^0
10^{-1}

When *not* to use a GPU

- The key to taking advantage of GPU computing is utilizing the mass parallelization of processes.
- If your program or application cannot be made to perform many small tasks then it will not be improved by GPU computing (it would probably make it worse).
- Another issue to look for is that memory transfer is not the time-limiting factor of your code. To process data, the CPU needs to transfer the data to the GPU through the motherboard PCIe interface, which is much slower than inter processor communication. In many cases this communication cost is the limiting factor of the code and may get worse by transferring to the GPU.

GPU Computing in Matlab

- Included in the Parallel Computing Toolbox.
- Extremely easy to use. To create a variable that can be processed using the GPU, use the *gpuArray* function.
- This function transfers the storage location of the argument to the GPU. Any functions which use this argument will then be computed by the GPU.
- Many of the built in functions have been overloaded to accept gpuArray arguments and will be evaluated on the GPU.
- For a list of these functions see: https://www.mathworks.com/help/distcomp/run-built-in-functions-on-a-gpu.html

Example

- N=256;
- h_A = magic(N);
- d_A = gpuArray(h_A);
- d_B = abs(A.*A);
- h_B = gather(d_B);

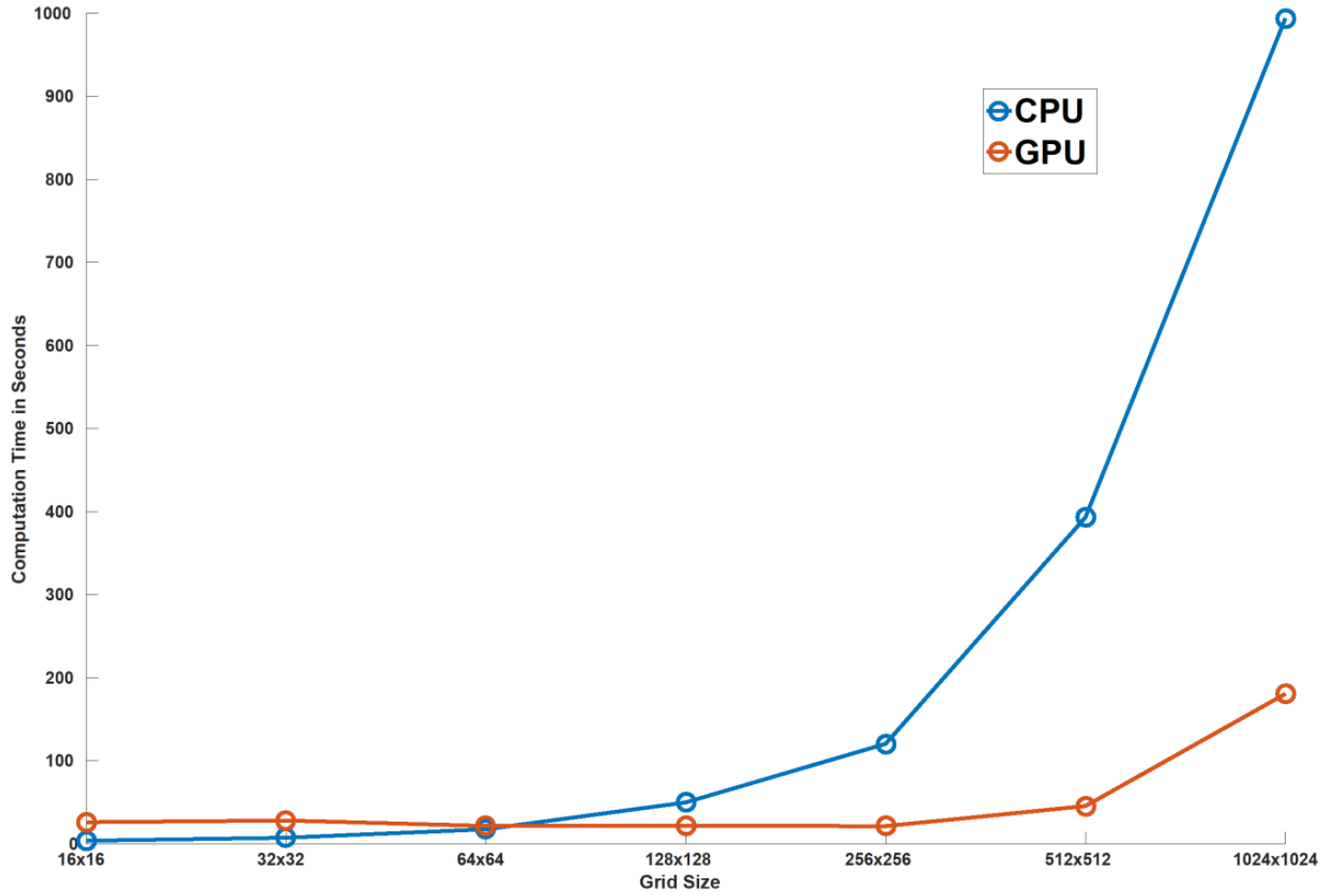
GPU Computing in Matlab

- The final step in the previous example used the *gather* function. This function returns the variable created on the GPU (d_B) to the CPU so that it can be further processed (plotted).
- That's it!
- A key use where GPUs can be particularly useful is the FFT (Fast-Fourier Transform). To take advantage of this, the only change that needs to be made is using the gpuArray to transfer the variable to the GPU and gather to bring it back.

Spectral Shallow Water Equation Solver

```
[6 unmodified lines hidden]
 7 %define a spatial grid                                  . %define a spatial grid                                  7
 8 xmin = -100;                                           . xmin = -100;                                           8
 9 xmax = 100;                                            . xmax = 100;                                            9
10 x = linspace(xmin,xmax,Nx);                            x x = gpuArray(linspace(xmin,xmax,Nx));                  10
11 dx = x(2)-x(1);                                        . dx = x(2)-x(1);                                        11
12 ymin = -100;                                           . ymin = -100;                                           12
13 ymax = 100;                                            . ymax = 100;                                            13
14 y = linspace(ymin,ymax,Ny);                            x y = gpuArray(linspace(ymin,ymax,Ny));                  14
15 dy = y(2)-y(1);                                        . dy = y(2)-y(1);                                        15
16 [xx,yy]=meshgrid(x,y);                                 . [xx,yy]=meshgrid(x,y);                                 16
17                                                        .                                                        17
18 %make initial condition                                . %make initial condition                                18
19 % Sech for traditional shock/rarefaction calculation   . % Sech for traditional shock/rarefaction calculation   19
20 r=sqrt(xx.^2+yy.^2);                                   x r=sqrt(xx.^2+yy.^2);                                   20
21 myamp=0.8*H;                                           . myamp=0.8*H;                                           21
22 eta = myamp*sech(r/(0.1*max(r(:)))).^8;                . eta = myamp*sech(r/(0.1*max(r(:)))).^8;                22
23 u=zeros(size(eta));                                    x u=gpuArray(zeros(size(eta)));                          23
24 v=zeros(size(eta));                                    x v=gpuArray(zeros(size(eta)));                          24
25 eta2 = myamp*sech(r/(0.1*max(r(:)))).^8;               . eta2 = myamp*sech(r/(0.1*max(r(:)))).^8;               25
26 u2=zeros(size(eta));                                   x u2=gpuArray(zeros(size(eta)));                         26
27 v2=zeros(size(eta));                                   x v2=gpuArray(zeros(size(eta)));                         27
28                                                        .                                                        28
29 u0=u;                                                  . u0=u;                                                  29
30 eta0=eta;                                              . eta0=eta;                                              30
31 v0=v;                                                  . v0=v;                                                  31
32                                                        .                                                        32
33 %make wave numbers                                     . %make wave numbers                                     33
34 nyquist_freqx = 2*pi/(xmax-xmin);                      . nyquist_freqx = 2*pi/(xmax-xmin);                      34
35 ks=[0:Nx/2-1 0 -Nx/2+1:-1]*nyquist_freqx;              x ks=gpuArray([0:Nx/2-1 0 -Nx/2+1:-1]*nyquist_freqx);    35
36 nyquist_freqy = 2*pi/(ymax-ymin);                      . nyquist_freqy = 2*pi/(ymax-ymin);                      36
37 ls=[0:Ny/2-1 0 -Ny/2+1:-1]*nyquist_freqy;              x ls=gpuArray([0:Ny/2-1 0 -Ny/2+1:-1]*nyquist_freqy);    37
38 [kk,ll]=meshgrid(ks,ls);                               . [kk,ll]=meshgrid(ks,ls);                               38
39 kk2=kk.*kk;                                           . kk2=kk.*kk;                                            39
40 ll2=ll.*ll;                                           . ll2=ll.*ll;                                            40
[10 unmodified lines hidden]
51 lcut=0.35*lnyq;                                       . lcut=0.35*lnyq;                                        51
52 %kcut=0.25*knyq;                                      . %kcut=0.25*knyq;                                       52
53 filtbeta=2;                                           . filtbeta=2;                                            53
54 dummy=ones(size(ks));                                 x dummy=gpuArray(ones(size(ks)));                        54
55 myfiltax = f_bump(0.5, filtbeta, kcut/knyq, ks/knyq);  . myfiltax = f_bump(0.5, filtbeta, kcut/knyq, ks/knyq);  55
56 myfiltay = f_bump(0.5, filtbeta, lcut/lnyq, ls/lnyq);  . myfiltay = f_bump(0.5, filtbeta, lcut/lnyq, ls/lnyq);  56
57 myfilta=repmat(myfiltay',[1 Nx]).*repmat(myfiltax,[Ny 1]); . myfilta=repmat(myfiltay',[1 Nx]).*repmat(myfiltax,[Ny 1]); 57
[98 unmodified lines hidden]
```

Performance Difference



Monte Carlo Simulations

- Monte Carlo simulations are particularly suited to GPU computing as you can perform an ensemble of 1000s in roughly the same time as a single simulation.
- To perform them in Matlab you use the *arrayfun* function. The usage of this function is: *vector_answers =arrayfun(@somefunction,vector_input1,vector_input2,…).*
- This will perform the function *somefunction* using the inputs *vector_input1(1), vector_input2(1),…* for each vector set.
- If these vector inputs are gpuArrays then each application of the function will be performed on a separate thread.

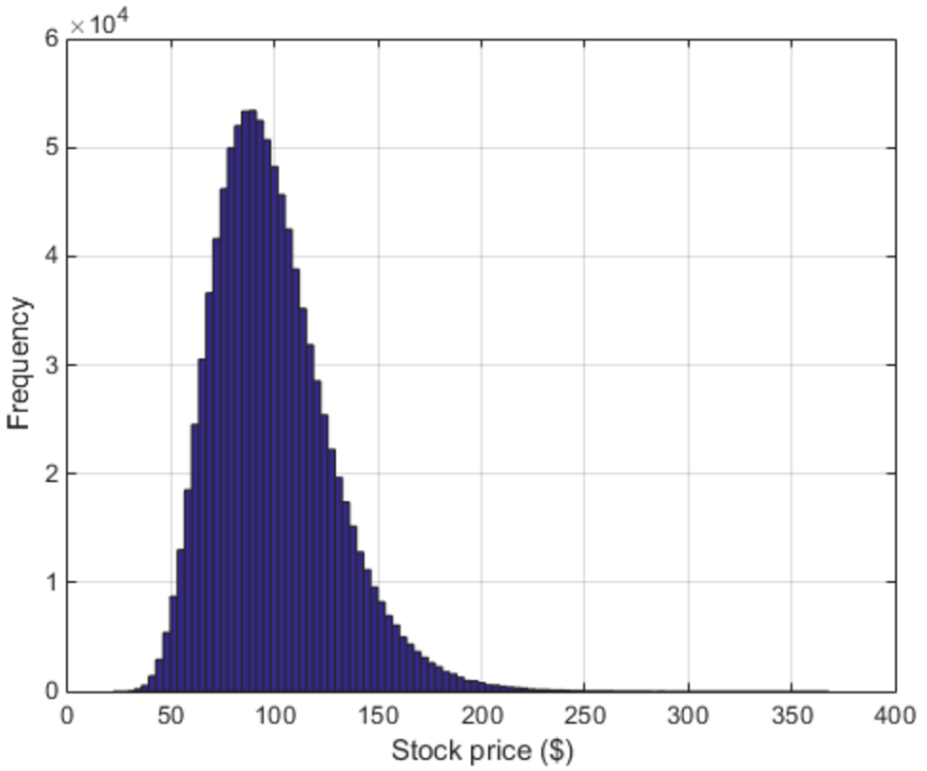
Running on the GPU

To run stock price simulations on the GPU we first need to put the simulation loop inside a helper function:

```
function finalStockPrice = simulateStockPrice(S,r,d,v,T,dT)
    t = 0;
    while t < T
        t = t + dT;
        dr = (r - d - v*v/2)*dT;
        pert = v*sqrt( dT )*randn();
        S = S*exp(dr + pert);
    end
    finalStockPrice = S;
end
```

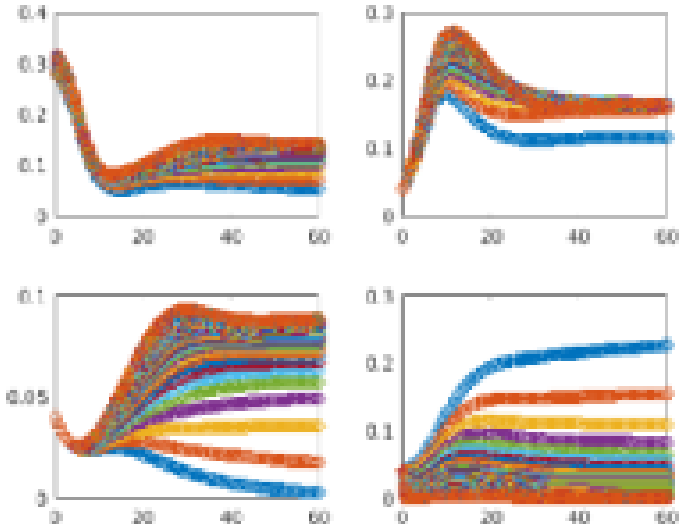
We can then call it thousands of times using `arrayfun`. To ensure the calculations happen on the GPU we make the input prices a GPU vector with one element per simulation. To accurately measure the calculation time on the GPU we use the `gputimeit` function.

```
% Create the input data.
N = 1000000;
startStockPrices = stockPrice*ones(N,1,'gpuArray');

% Run the simulations.
finalStockPrices = arrayfun( @simulateStockPrice, ...
    startStockPrices, riskFreeRate, dividend, volatility, ...
    timeToExpiry, sampleRate );
meanFinalPrice = mean(finalStockPrices);

% Measure the execution time of the function on the GPU using gputimeit.
% This requires us to store the |arrayfun| call in a function handle.
functionToTime = @() arrayfun(@simulateStockPrice, ...
    startStockPrices, riskFreeRate, dividend, volatility, ...
    timeToExpiry, sampleRate );
timeTaken = gputimeit(functionToTime);

fprintf( 'Calculated average price of $%1.4f in %1.3f secs.\n', ...
    meanFinalPrice, timeTaken );

clf;
hist( finalStockPrices, 100 );
xlabel( 'Stock price ($)' )
ylabel( 'Frequency' )
grid on;
```



Parameter Sweep

- This is an NPZD model where we are investigating the dynamics of changing the parameter *k*.
- *k* goes from 0.05 to 10 by steps of 0.05 so there are 199 values.
- The loop steps time forward and if it is a print step it will gather the variables back from the GPU.



```
close all;clear

seed='shuffle';
rng(seed);
parallel.gpu.rng(seed,'Philox4x32-10');
k = linspace(0.05, 10, (10-0.05)/0.05);
num_sims = length(k);
N0 = 1.4*0.2;
P0 = 0.2*0.2;
Z0 = 0.2*0.2;
D0 = 0.2*0.2;

t0 = 0;

step2print=5;
cntr=1;
cntr2=1;
dt = 0.1;
tf = 60;
tn = t0;
Nn = gpuArray(N0);
Pn = gpuArray(P0);
Zn = gpuArray(Z0);
Dn = gpuArray(D0);
Ns = zeros(num_sims,(tf/dt)/step2print);
Ns(:,1) = N0;
Ps = zeros(num_sims,(tf/dt)/step2print);
Ps(:,1) = P0;
Zs = zeros(num_sims,(tf/dt)/step2print);
Zs(:,1) = Z0;
Ds = zeros(num_sims,(tf/dt)/step2print);
Ds(:,1) = D0;
ts = zeros((tf/dt)/step2print,1);

while tn < tf
    [Np,Pp,Zp,Dp,tp] = arrayfun(@eulernpzdgpu, dt, tn, Nn, Pn, Zn, Dn, k);
    if mod(cntr,step2print)==0
        Ns(:,cntr2+1) = gather(Np);
        Ps(:,cntr2+1) = gather(Pp);
        Zs(:,cntr2+1) = gather(Zp);
        Ds(:,cntr2+1) = gather(Dp);
        ts(cntr2+1) = gather(tp(1));
        cntr2=cntr2+1;
    end
    Nn = Np;
    Pn = Pp;
    Zn = Zp;
    Dn = Dp;
    tn = tp(1);
    cntr = cntr+1;
end
```

GPU Computing in R

- Computing with GPUs is also extremely easy in R. All that is required is to download one of the packages that have prebuilt functions. There are a number of different packages that can be downloaded that allow for GPU computing. Some examples are: gpuR, rpud, gputools, cudaBayesreg.
- Once the package is downloaded you can call the provided functions which will operate on the GPU.

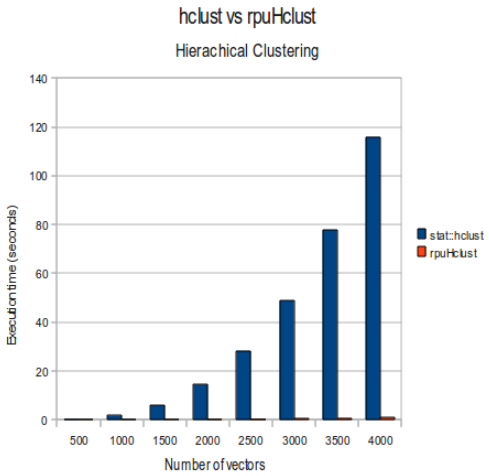
gputools

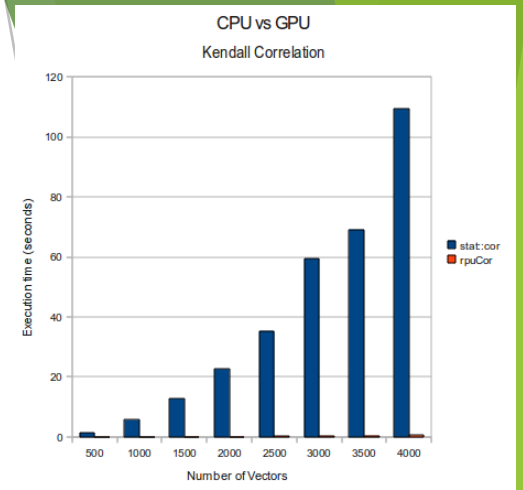
- *gpuMatMult* -- Perform Matrix Multiplication with a GPU
  - matA <- matrix(runif(2*3), 2, 3)
  - matB <- matrix(runif(3*4), 3, 4)
  - gpuMatMult(matA, matB)
- *gpuQr* -- Estimate the QR decomposition for a matrix
  - # get some random data of any shape at all
  - x <- matrix(runif(25), 5, 5)
  - qr <- gpuQr(x)

gputools

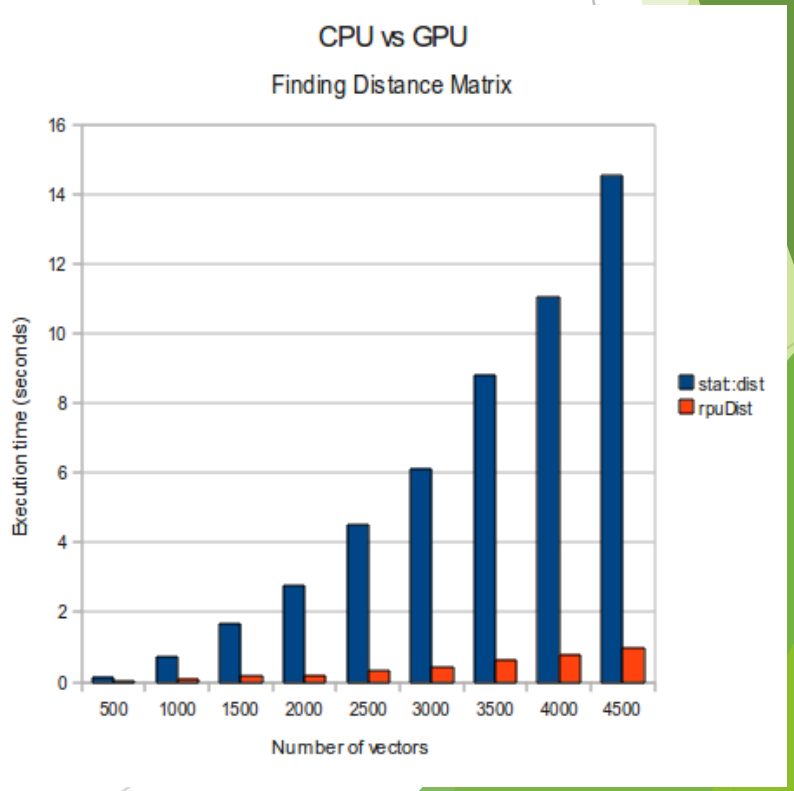
- *gpuDistClust* -- Compute Distances and Hierarchical Clustering for Vectors on a GPU
  - numVectors <- 5
  - dimension <- 10
  - Vectors <- matrix(runif(numVectors*dimension), numVectors, dimension)
  - myClust <- gpuDistClust(Vectors, "maximum", "mcquitty“)
- *gpuLm* -- Fitting Linear Models using a GPU-enabled QR
  - ctl <- c(4.17,5.58,5.18,6.11,4.50,4.61,5.17,4.53,5.33,5.14)
  - trt <- c(4.81,4.17,4.41,3.59,5.87,3.83,6.03,4.89,4.32,4.69)
  - group <- gl(2,10,20, labels=c("Ctl","Trt"))
  - weight <- c(ctl, trt)
  - anova(lm.D9 <- gpuLm(weight ~ group))
  - summary(lm.D90 <- gpuLm(weight ~ group - 1))
  - summary(resid(lm.D9) - resid(lm.D90))

rpud





- *Rpuchol* -- GPU Accelerated Cholesky Decomposition
  - N <- 20
  - x <- matrix(runif(N*5), ncol=N)
  - A <- t(x)
  - rpuchol(A)
- *rpucor.test* -- Compute the p-values of the correlation matrix
  - num <- 5
  - dim1 <- 6
  - dim2 <- 8
  - x <- matrix(runif(num*dim1), num, dim1)
  - y <- matrix(runif(num*dim2), num, dim2)
  - rpucor.test(x, y, method = "kendall")
  - # introduce missing values
  - x[3,5] <- NA
  - y[4,1] <- NA
  - rpucor.test(x, y, method = "kendall", use="pairwise.complete.obs")



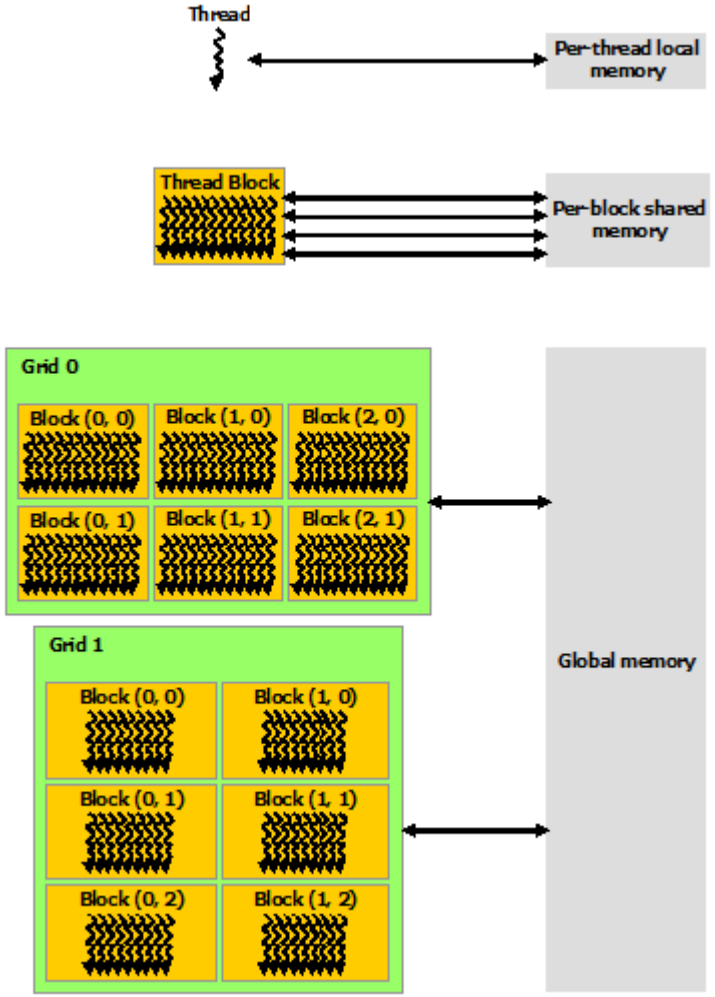
Behind the scenes…

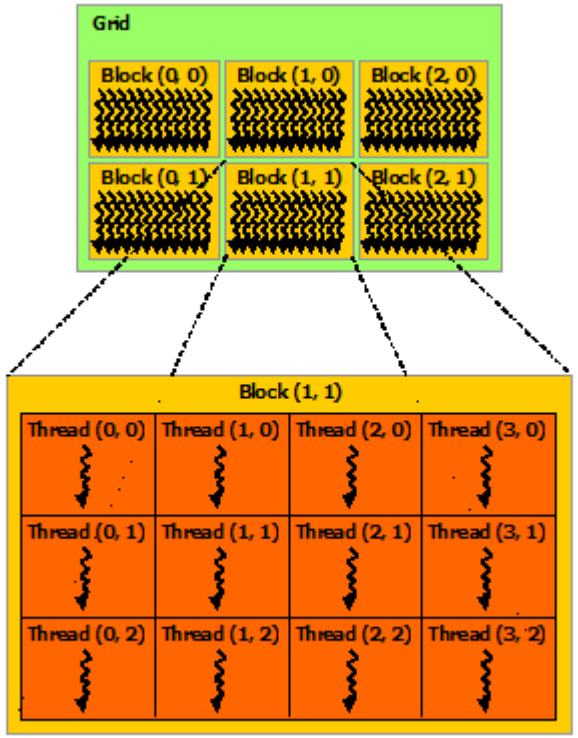
- In both Matlab and R the actual interface with the GPU is performed using CUDA C++.
- As such in order to use the functionality I have outlined before you need to download CUDA. It can be found at: https://developer.nvidia.com/cuda-downloads
- For making your own applications, beyond what you can do within these languages, you may wish to work in C++ so as to directly control the computation.
- For compiling CUDA code you must use the compiler provided by Nvidia, nvcc. It is based on g++ and has most of its features.

CUDA C++

- Assuming that you have installed CUDA in your path, you need to provide the linker –lcudart (CUDA runtime).
- In your header you will then need to add #include<cuda>.
- The way coding in CUDA C++ works is that you create “kernels” which are functions that will be evaluated on the GPU. These functions are preceded by the declaration specifier __global__.
- To invoke a kernel you call: SOME_KERNEL<<<n_blocks,size_blocks>>>(input1, input2, …)
- You may be wondering what n_blocks and size_blocks are…

GPU Breakdown





Example

```
// Kernel definition
__global__ void MatAdd(float A[N][N], float B[N][N],
float C[N][N])
{
    int i = blockIdx.x * blockDim.x + threadIdx.x;
    int j = blockIdx.y * blockDim.y + threadIdx.y;
    if (i < N && j < N)
        C[i][j] = A[i][j] + B[i][j];
}

int main()
{
    ...
    // Kernel invocation
    dim3 threadsPerBlock(16, 16);
    dim3 numBlocks(N / threadsPerBlock.x, N / threadsPerBlock.y);
    MatAdd<<<numBlocks, threadsPerBlock>>>(A, B, C);
    ...
}
```

CUDA C++

- In order to create a variable that is stored on the GPU you must first allocate is memory. This is done through the cudaMalloc function.
- This function takes in the address of your declared variable (&variable) and the size to allocate. For a vector of length N this would be N*sizeof(float,int,double,etc…).
- Together this is cudaMalloc(&variable,N*sizeof(float)).
- Generally you will have a corresponding copy of this variable on the CPU which is allocated using the normal malloc() function.

CUDA C++

- As with in Matlab, you have to transfer your data to and from the GPU if you want to perform any other operations on it (for example write it out).
- To do this you use the *cudaMemcpy* function. It takes in: target_vector, original_vector, size of vector, and direction.
- The target_vector and original_vector are both pointers to vectors where one of them is located on the GPU while the other is located on the CPU/RAM.
- The size is just number of elements * sizeof(float,int,double,etc…).
- The direction is either cudaMemcpyHostToDevice or cudaMemcpyDeviceToHost depending on which way the data is moving.

Example

```
// Device code
__global__ void VecAdd(float* A, float* B, float* C, int N)
{
    int i = blockDim.x * blockIdx.x + threadIdx.x;
    if (i < N)
        C[i] = A[i] + B[i];
}

// Host code
int main()
{
    int N = ...;
    size_t size = N * sizeof(float);

    // Allocate input vectors h_A and h_B in host memory
    float* h_A = (float*)malloc(size);
    float* h_B = (float*)malloc(size);

    // Initialize input vectors
    ...

    // Allocate vectors in device memory
    float* d_A;
    cudaMalloc(&d_A, size);
    float* d_B;
    cudaMalloc(&d_B, size);
    float* d_C;
    cudaMalloc(&d_C, size);

    // Copy vectors from host memory to device memory
    cudaMemcpy(d_A, h_A, size, cudaMemcpyHostToDevice);
    cudaMemcpy(d_B, h_B, size, cudaMemcpyHostToDevice);

    // Invoke kernel
    int threadsPerBlock = 256;
    int blocksPerGrid =
            (N + threadsPerBlock - 1) / threadsPerBlock;
    VecAdd<<<blocksPerGrid, threadsPerBlock>>>(d_A, d_B, d_C, N);

    // Copy result from device memory to host memory
    // h_C contains the result in host memory
    cudaMemcpy(h_C, d_C, size, cudaMemcpyDeviceToHost);

    // Free device memory
    cudaFree(d_A);
    cudaFree(d_B);
    cudaFree(d_C);

    // Free host memory
    ...
}
```

CUDA Libraries

- In addition to the C++ API, CUDA comes with several libraries that are CUDA versions of standard libraries.
- Some examples include: cuFFTW, cuBLAS, cuRAND, and cuSPARSE.
- Some of these libraries even provide built in understanding of the original library, eg. cuFFTW understands most FFTW3 syntax.
- In addition to this there are several specific guides for optimizing your code for a given GPU architecture. These can all be found at: http://docs.nvidia.com/cuda/index.html#axzz4bKHKYSVN

MFCF Resources

- We currently have a GPU server with 8 Telsa K80s providing up to 128 TFLOPS of computational power and 16GB of memory each.
- This server can be accessed through ssh at yourusername@gpu01.student.math.uwaterloo.ca.
- You use your Nexus login information.
- If you are connecting from off-campus you need to use the CISCO VPN Client before you connect.
- This sever can be used to test your code.

Summary

- GPU computing is an extremely powerful tool for massively parallelizable problems.
- It can provide huge speed ups for very little investment compared to traditional CPU computing.
- Both Matlab and R provide very easy to learn and use interfaces for taking advantage of this.
- The general design idea's can be relatively easily implemented in CUDA C++.
- It should also be mentioned that CUDA is not the only option. ATI have developed an open source API called OpenCL which will work with any GPU.
- MFCF currently has a GPU cluster which anyone can access to perform GPU computations.

References

- http://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html#axzz4bUyxiASo
- http://docs.nvidia.com/cuda/cuda-quick-start-guide/index.html#axzz4bUyxiASo
- https://www.mathworks.com/help/distcomp/examples/using-gpu-arrayfun-for-monte-carlo-simulations.html
- https://www.mathworks.com/help/distcomp/gpu-computing-in-matlab.html
- http://www.r-tutor.com/gpu-computing
- https://www.r-bloggers.com/r-gpu-programming-for-all-with-gpur/
- https://www.r-project.org/conferences/useR-2011/TalkSlides/Contributed/16Aug_1115_FocusI_3-HighPerfComp_1-Ligtenberg.pdf