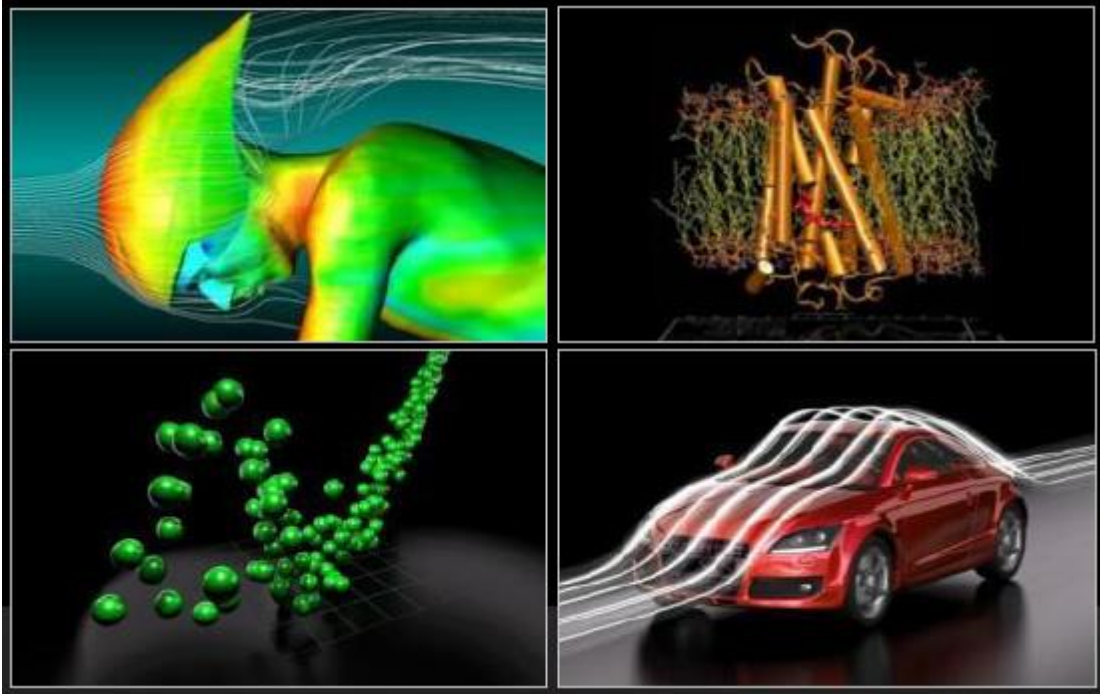# *A Deeper Exploration of HPC GPU Computing:*
CUDA and Beyond

Antony Drew – LUC – COMP 439
Fall 2014
adrew@luc.edu

Table of Contents

Introduction – What are we trying to accomplish?

The aim of this project is to explore the potential benefits of GPU computing in its various forms. By laying the groundwork through working examples and exposition, various speed benefits will be demonstrated and also shown to be surmountable by the average practitioner. This subject should not be daunting or deemed necessarily out-of-reach by the typical person. If certain rules can be kept straight, code samples of increasing complexity are attainable. Hopefully, this demonstration is persuasive and can induce others into scientific computing, especially, younger generations.

What is HPC Hybrid GPU Computing? Matrices, vectors and more…

"HPC" stands for high-performance computing. "Hybrid" means that both the CPU and GPU are being targeted for various calculations in order to share the workload. Previously, pools of "workers" or CPU's (e.g., AMD/INTEL chips) were coordinated together to perform vast numerical computations in a distributed fashion. These are often referred to as "clusters" or "farms". However, in the late 1990's, academics on the West Coast along with companies like Nvidia began to explore libraries to take advantage of the GPU – 'GPU' here means nothing more than the chip on a video card typically used to render graphics on a computer screen – when several of these are encased in a server, then we have a cluster. As GPU technology became more advanced by 1999[1], practitioners in the science and finance fields began to incorporate GPU clusters into their operations. Remember

---

[1] Feb. 2012 < http://www.nvidia.com/object/cuda_home_new.html >.

that graphics rendering typically involves mathematical vector or matrix operations to draw lines and curves. This word 'matrix' (e.g., a bunch of columns or vectors) is important because many computations in various fields involve matrices, so, it is a natural extension to then use these video cards for other purposes besides graphics. In fact, portfolio theory in finance is nothing more than a series of matrix operations and manipulations though we tend to forget this coincidence.

To see matrices in action, let's remember back to some simple algebra from high school. Recall the formula for **covariance** in statistics of two variables:

σ(a, u) = (a – ā)(u – ū)   ← *algebraic representation of formula*

Now, let's represent the same formula in **matrix** notation[2]:

$$cov(x, x') = cov(Q'y, y'Q) = Q'cov(y, y')Q = Q'Q\Lambda Q'Q = \Lambda.$$

$$\begin{bmatrix} \sigma^2(x^{(1)}) & \sigma(x^{(1)}, x^{(2)}) & \sigma(x^{(1)}, x^{(3)}) \\ & \sigma^2(x^{(2)}) & \sigma(x^{(2)}, x^{(3)}) \\ & & \sigma^2(x^{(2)}) \end{bmatrix} = \begin{bmatrix} \lambda_1 & 0 & 0 \\ 0 & \lambda_2 & 0 \\ 0 & 0 & \lambda_3 \end{bmatrix}$$

This type of inverse matrix operation, **QΛQ'**, is actually called "**Eigenvalue Decomposition**" and is nothing more than the transposition of a matrices forming, **Λ**, to produce the covariance or lambda, **λ**, along the diagonal axis. These types of operations are used frequently in finance and science so we can see the parallel between simple algebra and matrix algebra. Just like in graphics rendering, the GPU is all about matrices so there is a natural intersection here though not apparent at first glance. So, in hybrid computing, the computer is simply doing some of the calculations on the CPU (with larger cores) and others on the GPU (with smaller

---

[2] John Cochrane, "Time Series for Macroeconomics and Finance," University of Chicago January 2005: 262.

cores) – by incorporating pointers and arrays, code can typically be "vectorized" in terms of executing memory and math operations in "blocks" (versus single elements) - here is a nice picture:
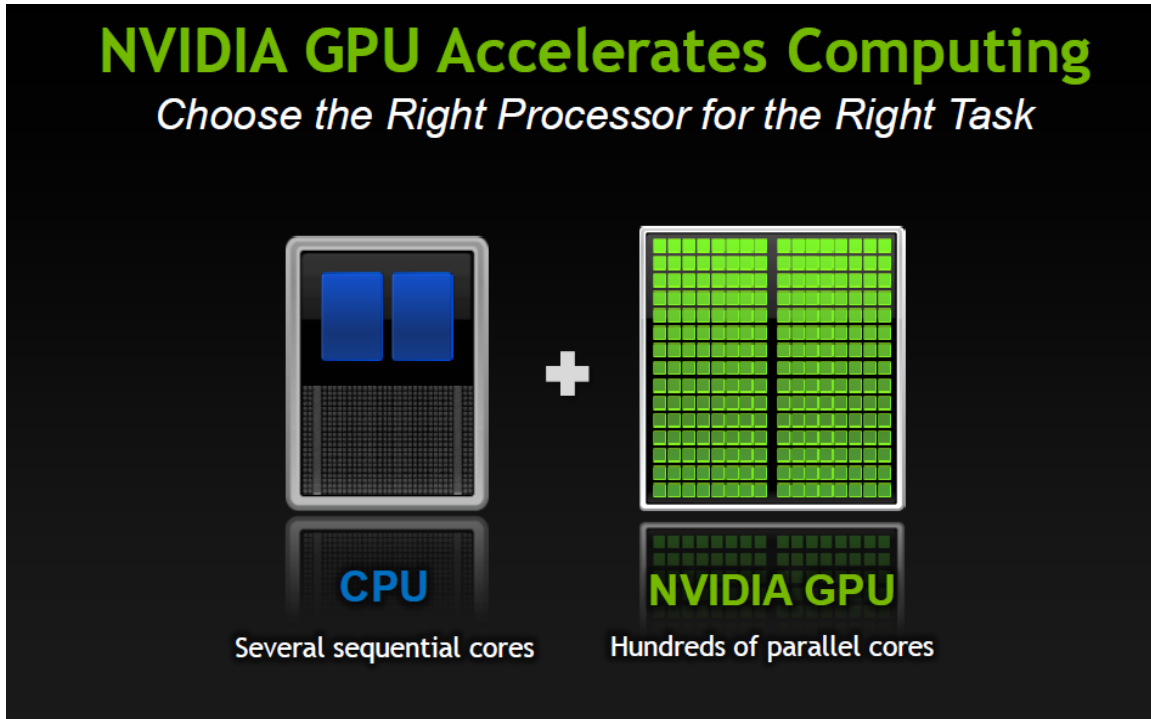
*Figure 1 – **Hybrid Computing in a Picture** Nvidia, "uchicago_intro_apps_libs.pdf," 2012: 21.*

Here is a better layout of one of the most powerful GPU setups in the world at Oak Ridge National Laboratory, the *Titan*:
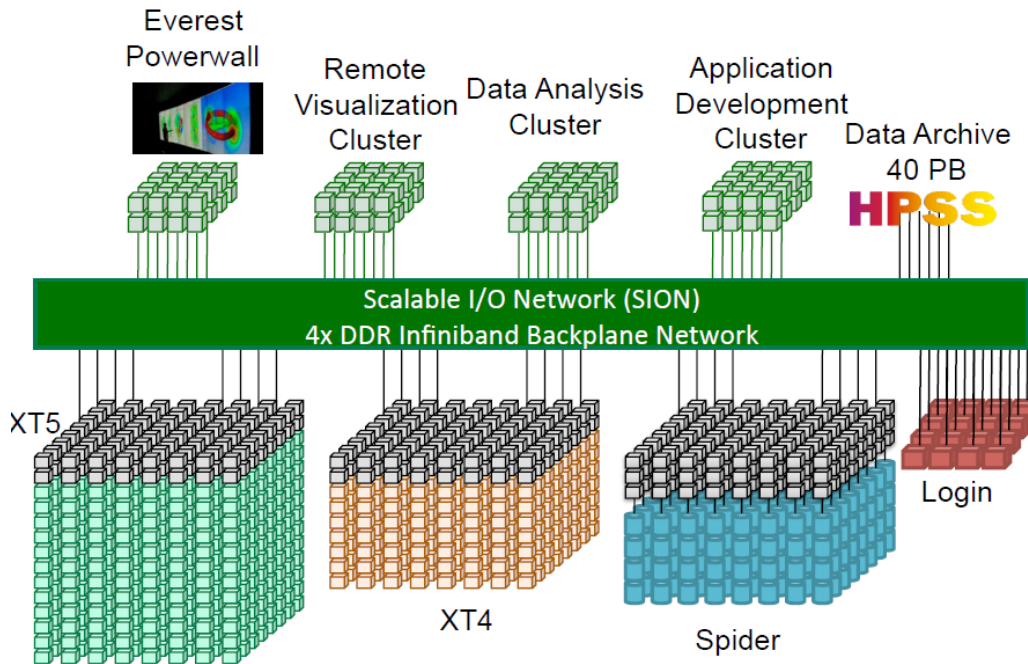
Titan – Cray XK7 , Opteron 6274 16C 2.200GHz, Cray Gemini interconnect, NVIDIA K20x

Site:	DOE/SC/Oak Ridge National Laboratory (/site/48553)
System URL:	http://www.olcf.ornl.gov/titan/ (http://www.olcf.ornl.gov/titan/)
Manufacturer:	Cray Inc.
Cores:	560,640
Linpack Performance (Rmax)	17,590 TFlop/s
Theoretical Peak (Rpeak)	27,112.5 TFlop/s
Power:	8,209.00 kW
Memory:	710,144 GB
Processor:	Opteron 6274 16C 2.2GHz
Interconnect:	Cray Gemini interconnect
Operating System:	Cray Linux Environment

Ranking

List	Rank	System	Vendor	Total Cores	Rmax (TFlops)	Rpeak (TFlops)	Power (kW)
06/2014 (/lists/2014/06)	2	Cray XK7 , Opteron 6274 16C 2.200GHz, Cray Gemini interconnect, NVIDIA K20x	Cray Inc.	560,640	17,590.0	27,112.5	8,209.00
11/2013 (/lists/2013/11)	2	Cray XK7 , Opteron 6274 16C 2.200GHz, Cray Gemini interconnect, NVIDIA K20x	Cray Inc.	560,640	17,590.0	27,112.5	8,209.00

Figure 2 – Robust GPU Server Cluster Oak Ridge, "SC10_Booth_Talk_Bland.pdf," 2010: 5.

Distributed System Summary of GPU Server:

Scale:	Large
Heterogeneity:	Radically Diverse
Openness:	Mixed/Challenging - SSH/Tunneling (secure)
QOS:	Mixed/Challenging
Storage:	Distributed File System; DSM (Shared Memory) Async/Sync
Communication:	IPC - Message Passing
Network:	Physical Servers; WAN - SSH/Tunneling Access

Why do we need GPU Computing? Hitting the wall…

As modern scientific research becomes more complex incorporating ever larger data sets, the classic CPU-centric paradigm runs out of computing resources. Before GPU computing, programs using intensive computation might take several days to finish running and calculating the results. If we were dependent on those results before specifying the next iteration of a model for testing purposes, we literally had to wait until proceeding further with research. Think about the wasted time of such a process. With hybrid computing, we can now get results back in a matter of hours or even minutes – some firms are even calculating and executing complex operations in milliseconds/microseconds. These firms were literally hitting a wall in terms of speed and so needed other alternatives. GPU computing is massively parallel and scalable and can speed up applications by 2X to 100X. If you have enough resources and hardware, you can keep reducing computation time by adding more GPU's. This does cost a lot of money, however, and some firms spend

tens of millions of dollars on hardware including both CPU and GPU clusters (e.g.,

collocated servers). Remember that a GPU is full of thousands of smaller cores

whereas the CPU has fewer, but larger cores. Think of it as many little bicycles

versus fewer large trucks. However, for certain tasks, the CPU simply cannot keep

up with the GPU. Typically, relative performance between the CPU and GPU is

measured in the amount of single-and-double precision operations per second

(FLOPS). The next chart shows how fast the divergence in speed is progressing

towards a 100X advantage by 2021:
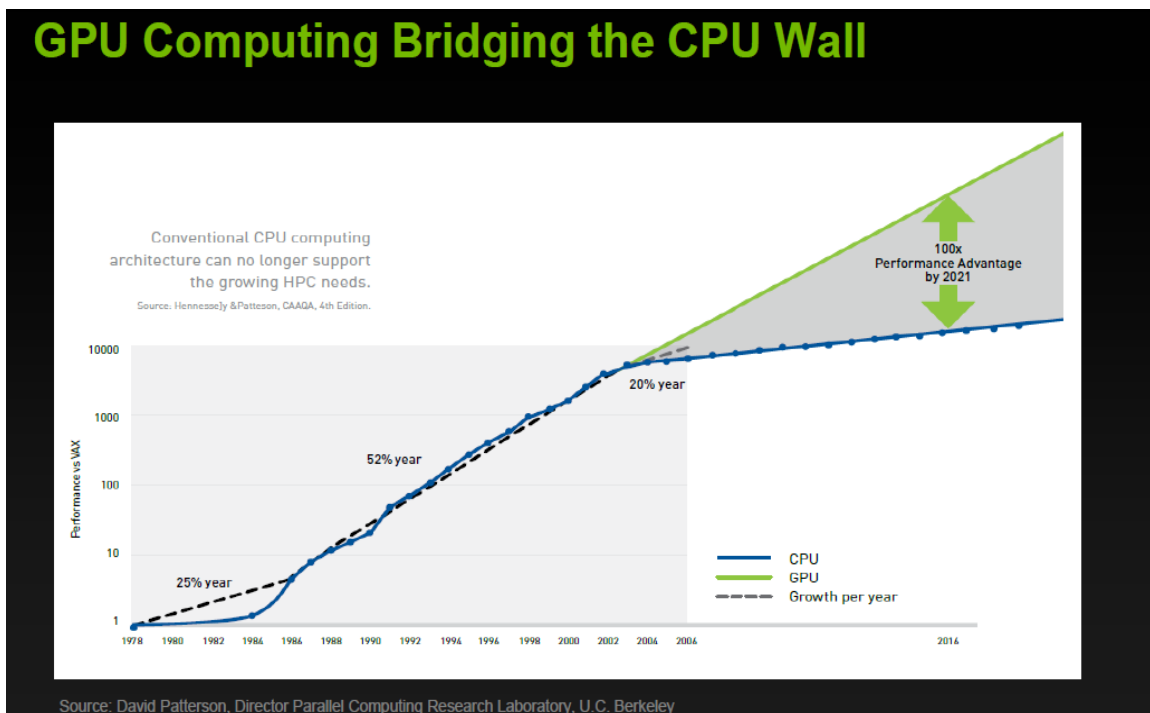
Figure 3 – GPU versus CPU Nvidia, 2012: 6.
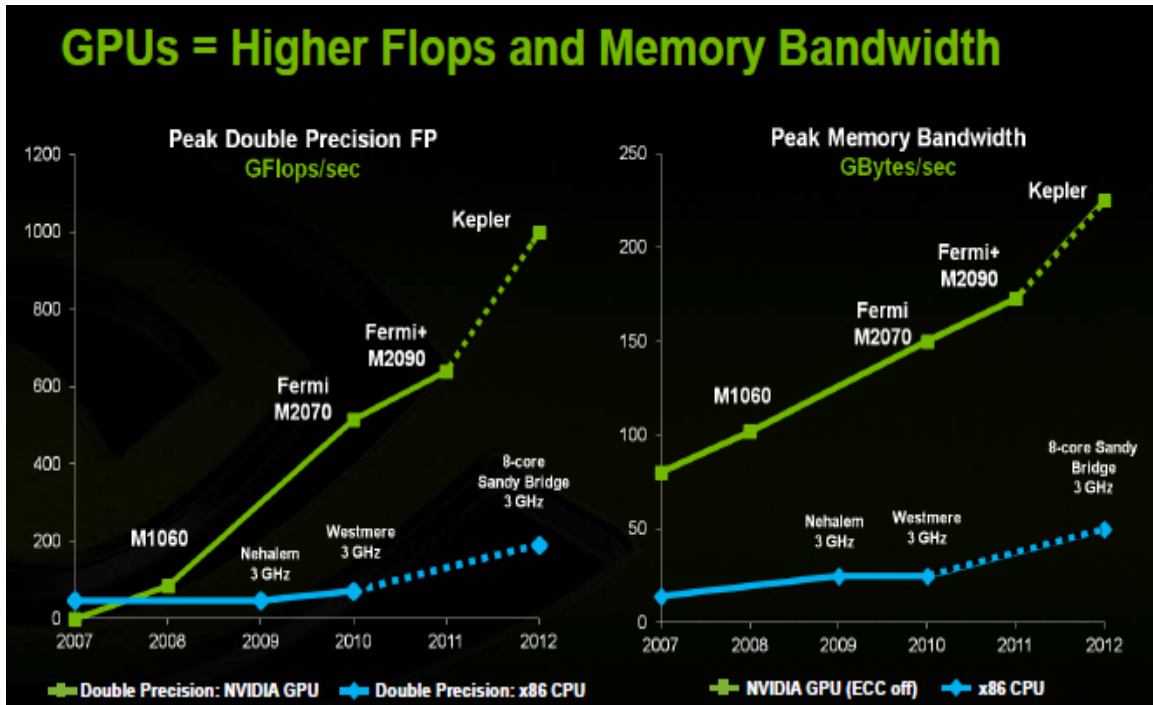
Figure 4 – Organizations Using GPU for Speed Nvidia, 2012: 7.

Features	Tesla K20X	Tesla K20	Tesla K10	Tesla M2090	Tesla M2075
Number and Type of GPU	1 Kepler GK110		2 Kepler GK104s	1 Fermi GPU	1 Fermi GPU
GPU Computing Applications	Seismic processing, CFD, CAE, Financial computing, Computational chemistry and Physics, Data analytics, Satellite imaging, Weather modeling		Seismic processing, signal and image processing, video analytics	Seismic processing, CFD, CAE, Financial computing, Computational chemistry and Physics, Data analytics, Satellite imaging, Weather modeling	
Peak double precision floating point performance	1.31 Tflops	1.17 Tflops	190 Gigaflops (95 Gflops per GPU)	665 Gigaflops	515 Gigaflops
Peak single precision floating point performance	3.95 Tflops	3.52 Tflops	4577 Gigaflops (2288 Gflops per GPU)	1331 Gigaflops	1030 Gigaflops
Memory bandwidth (ECC off)	250 GB/sec	208 GB/sec	320 GB/sec (160 GB/sec per GPU)	177 GB/sec	150 GB/sec
Memory size (GDDR5)	6 GB	5 GB	8GB (4 GB per GPU)	6 GigaBytes	6 GigaBytes
CUDA cores	2688	2496	3072 (1536 per GPU)	512	448

Figure 5 – Single versus Double Precision Yields More Gains in Speed Nvidia, 2012: 8.
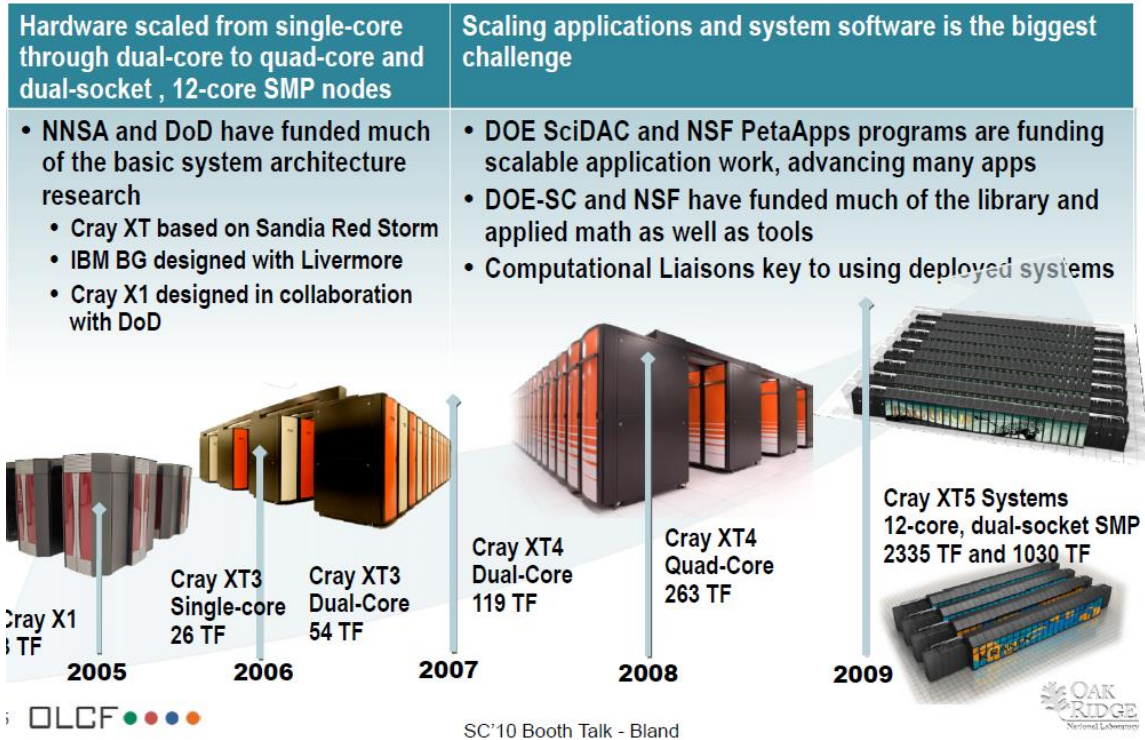
Figure 6 – Evolution of HPC Networks from CPU to GPU Oak Ridge, "SC10_Booth_Talk_Bland.pdf," 2010: 6.

Testing Approach, Setup & Hypothesis

The goal is prove out the speed gains from GPU computing by designing and running a complex program across several platforms. This program will involve large data sets and complex math operations (see financial modelling section below for the exact test design). Since this study involves distributed systems, the program MUST compile and run on a non-local GPU server cluster (and not simply at home on a single GPU setup) – getting code to run on foreign, unfamiliar hardware is a difficult challenge. We will then employ heuristics like timers to

measure the speed across various platforms. To recap, here is a brief overview of the test:

1. *Design a complex program (multi-streams, threads and GPUs) in C/C++ and make it run on CPU and GPU server (LUC TESLA)*
2. *Then test variations of the this same program in other languages/platforms such as JCUDA, OpenACC/MP & MATLAB [TODO]*
3. *Keep track of ALL CPU-only and GPU results to determine conclusion [TODO]*

Getting Setup for GPU Computing – CUDA C & Independent Tasks

Because the early pioneers of GPU computing worked in C/C++, this means that we need to use either C/C++ for the most robust programming in order to take full advantage of the GPU's capabilities. C++ is an object-oriented extension of the C language – they are similar though C is probably used more often in GPU programming. There are several typed of libraries (or DLL's) that are full of GPU functionality, but the oldest and most powerful one is made by Nvidia and is referred to as "CUDA". Because Nvidia maintains this library, they also require that an Nvidia video card be used to take advantage of CUDA. It does not have to be this way, but they made it this way for business reasons so that they can sell more product. There are some other alternatives that will be explored later on such as "Open ACC" and various "wrappers" like JCUDA, but let's start with CUDA since it has the most developed functionality. So, we will need the following to explore CUDA C GPU programming:

- A laptop or desktop with an Nvidia video card or GPU – powerful desktops are typically easier to work with here

- Install Nvidia SDK Toolkit and Drivers – available here: https://developer.nvidia.com/cuda-downloads – choose either a Windows (**Visual Studio** IDE) or Linux (**Eclipse** IDE) setup

- To see the "**trade-off**" between regular (CPU) computing and hybrid GPU computing, we'll need to code up several examples in C/C++. This means we'll have to "port" over programs into C/C++ from other languages with which we are more familiar. For example, programs in statistics packages like MATLAB, R and MATHEMATICA will have to be redone in C which is a lot of work – we will have to make a CPU-only version as well as a GPU hybrid version

    o Specifically, we will be targeting "**FOR NEXT**" loops inside code to make operations run in **parallel** on the GPU. This means operations must be **independent** – no **recursive** functions or array indexing of [**i-1**]. If we need to "look back", then we must use other functions to offset arrays by [i-1] in order to "trick" the computer in thinking that it is looking at today's value or element [i]. It's also good to use **single** versus **double** precision whenever possible since the CPU is optimized for double while the GPU is optimized for **single** (or float)

A Quick Primer in C: A Boxing Metaphor – Mike Tyson vs. Sugar Ray

Some of the younger generations can get anxious when they hear "C/C++". This might be due to the perception that C/C++ can be harder to learn due to "pointers". Pointers and more specifically, pointers-to-arrays, are very powerful tools that allow for rapid processing via vectorization. Let's focus on C here since this is what we'll predominantly use for this study. With its pointers, C is a minimalist and "muscle-bound" programming language. In boxing terms, C is a lot like the former boxer, Mike Tyson. Tyson did not have a "pretty" boxing style, but was a short, powerful man that wasted no energy in his punching technique – he had a brutally direct technique that was centered right at his opponents. The C language is a lot like this and is procedural rather than "OOP". There are not a lot of extras or "frills" in C or even pre-made libraries. When you use C, you have to manage the dynamic memory of pointers and code up most of your own functions

13

even for simple calculations such as variance and moving averages. This can seem like a pain in the neck, but this process ensures that you really understand what you are trying to do as well as the tools you are using to accomplish the task at hand. Many users use pre-built functions like "**variance**" in statistics programs without even knowing they are using "**sample variance**" (which has "N-1" in the denominator) – when you're forced to code this up yourself, you'll quickly find out something is wrong if you don't make this adjustment in the denominator. So, in the right hands, C is a lethal weapon. Let's also remember that many engineering programs in aerospace still run on C which is over 50 years old at this point in time. That's right – many missile and radar systems still run on C and it is still widely used in the defense industry. In fact, Boeing has its own version/extension of C.

On the other hand, .NET and scripting languages are probably more popular with younger folks. For example, let's consider C#/.NET in which various operations like memory management are taken care of "behind the scenes" via "garbage collection". There are also many libraries that can be easily referenced in C#. C# is an intermediate-level language with "JIT" compilation versus C. In C, very little is done 'behind the scenes' for us, so we must specify what we want and also pass by reference. But because of this as well as it being a "native" language, C can run very fast. So, C#/.NET is more like the graceful former boxer, Sugar Ray Leonard, who used to bob-and-weave and dancing around his opponents. OOP languages like C# also allow us to be very organized since we can create our own classes and group functions by classes. In C, we cannot create our own classes, but are bound by "built-in" classes like structures, instead. So, there is certainly a kind

of beauty or creativity which we cannot really capture in C. All these classes take up space and resources at some point, however, which is another reason why C runs quickly due to limited overhead. As with things, there are trade-offs between programming languages, but here C is really useful since we are concerned with **maximizing** speed in GPU computing via vectorization.

Let's work through a simple example using pointers so we can see the true power of C. When doing computation, we typically have variables of varying size which are like columns in an Excel spreadsheet. Thinking back to matrices, think of these variables as "vectors" – a matrix is nothing more than a bunch of columns or vectors (e.g., a "grid" in Excel). Here is typically what happens in C computation:

- Declare a **pointer** to an array for an unknown variable (of varying size) or '**vector**' in the **main** program – it is proper syntax to use the "indirection" operator or asterisk "*" here like "**double *var**" – by doing this, we are allowing for "**vectorization**" of code so that memory and math operations can be done in blocks rather than element-by-element
- Allocate space in memory for the variable via MALLOC or CALLOC like "**var=(double*) calloc(length,sizeof(double))**"
- Pass this variable by reference to the function call like "**function(var)**" – there is **no** "*" here since we are passing by reference
  - In the formal declaration and definition of the function, we must use "*" like "**void function(double *var)**"
  - Once **within** the function definition, we can then refer to an instance of the variable as **var[i]** using the typical index notation since the C-compiler makes no distinction between a pointer pointing to the first element of an array versus an array, itself[3] –

---

[3] Stephen G. Kochan, "Programming in C, Third Edition," Sams Publishing July 2005: 264.

that is, the C-compiler will always convert an array to a pointer of the same type for a function [array] variable (without you knowing about it)

- Since C does not check "**bounds**" before running (like in C#/.NET), we must be careful to not to loop past the end of the length of the variable

• Finally, after calling the function and getting the results back, we **must** "free up" the memory we allocated using "**free var**" – there is **no** "*" here since we are referring to the **address** like in all memory operations - if we forget to do this, our computer will start running out of memory as it eats into more-and-more virtual "soft" memory and so other programs/applications will slow down in performance (e.g., a memory "**leak**") – it's best to run a C-program in "**debug**" mode so that we eventually get a warning here, instead

• That's it – we just keep repeating this process over-and-over for each variable – not so bad as long as we can remember **when** to use "*" before the pointer (e.g., **dereference**)

So, we can intuit the powerful **2-way communication** here between the main calling routine/program and the specific function – this happens very **quickly** in C since we are using pointers and passing by reference. Here is a specific example:

Exhibit 1 – C-Code Pointer & Function Example *Author, 2013.*

```c
#include <stdio.h>
#include <math.h>
#include <io.h>
#include <stdlib.h>
#include <string.h>
#include <float.h>
```

Must include or refer to standard C libraries here to use some base functionality via "include" statement

```c
//forward declaration of function up here before main program
void s_tr(double *tr,double *hi, double *lo, double *cl, int n);
```

Declare function before MAIN program

```c
//start main program
void main (int argc, char *argv[]){
        int i=0;
        long numlin;

        //declare pointer array variables
        double *hi,*lo,*cl,*tr;
```

*Must use * here for declaration of pointers and memory allocation*

```c
        char source[60],line[100];
        FILE *fin;

        //open file of market data
        sprintf(source,"c:\\data\\%s.dat",argv[1]);
        fin=f_openr(source);
        numlin=f_line(fin);

        //allocate memory for pointer array variables
        hi=(double*)calloc(numlin+1,sizeof(double));
```

```c
        lo=(double*) calloc(numlin+1,sizeof(double));
        cl=(double*) calloc(numlin+1,sizeof(double));
        tr=(double*) calloc(numlin+1,sizeof(double));

        //read in data from file
        while(fgets(line,100,fin)>0){
                sscanf(line,"%*s %*s %lf %lf %lf",&hi[i],&lo[i],&cl[i++]);}

        //close file
        f_close(source,fin);

        //call function - pass by reference
        s_tr(tr,hi,lo,cl,i);          Do not need * when freeing memory
                                      or passing by reference to function

        //free memory of pointer array variables
        free (hi); free (lo); free (cl); free (tr); }
                                      Must use * here as input
                                      to function
//define function down here
void s_tr(double *tr,double *hi, double *lo, double *cl, int nday){
        double m1,m2,m3,m4; int i;
                                      Can use array indexing here
                                      inside function definition
        for(i=1;i<nday;i++){
                m1=abs(hi[i]-lo[i]);
                m2=abs(hi[i]-cl[i-1]);
                m3=abs(cl[i-1]-lo[i]);
                m4=max(m1,m2);
                tr[i]=max(m4,m3);}
        return; }
```

https://bitbucket.org/adrew/gpu/commits/527fc01520b6fb16c5a27053ccea6739

Description of GPU Server Operations & Architecture

A CPU Task versus a GPU Task: Pizza Delivery

CPU cores resemble large trucks compared to smaller GPU cores. Again, it is helpful to think this dichotomy as trucks versus motorcycles. The CPU is good for large, sequential operations. Most programs are not programmed to enable multi-threading on the CPU so it is fair to say that the CPU goes about its work in non-parallel fashion bouncing around from task-to-task. On the other hand, the GPU is designed inherently for parallelism. If we imagine pizza delivery in a neighborhood, the CPU would deliver a pizza to one house and then move on to the next. The GPU would send out many smaller messengers simultaneously to multiple houses via its smaller cores, however, which is why operations must be **independent**. The next

17

few slides show this pictorially:
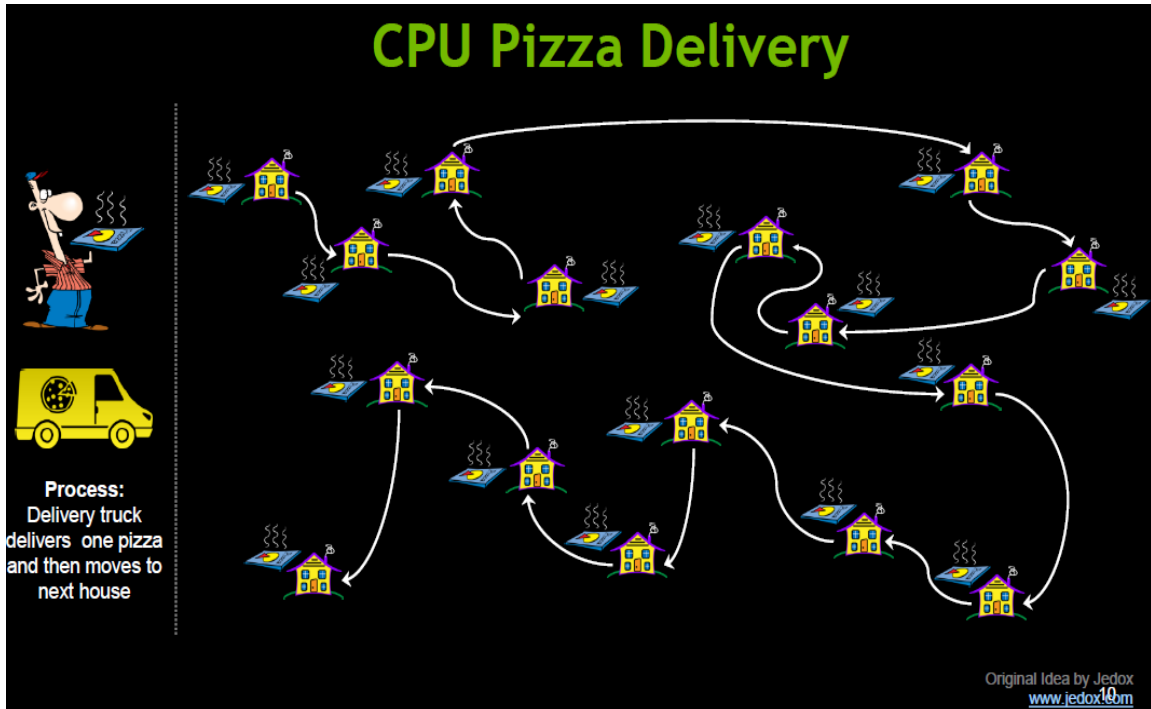
Figure 7 – A Typical CPU Operation Nvidia, 2012: 10.
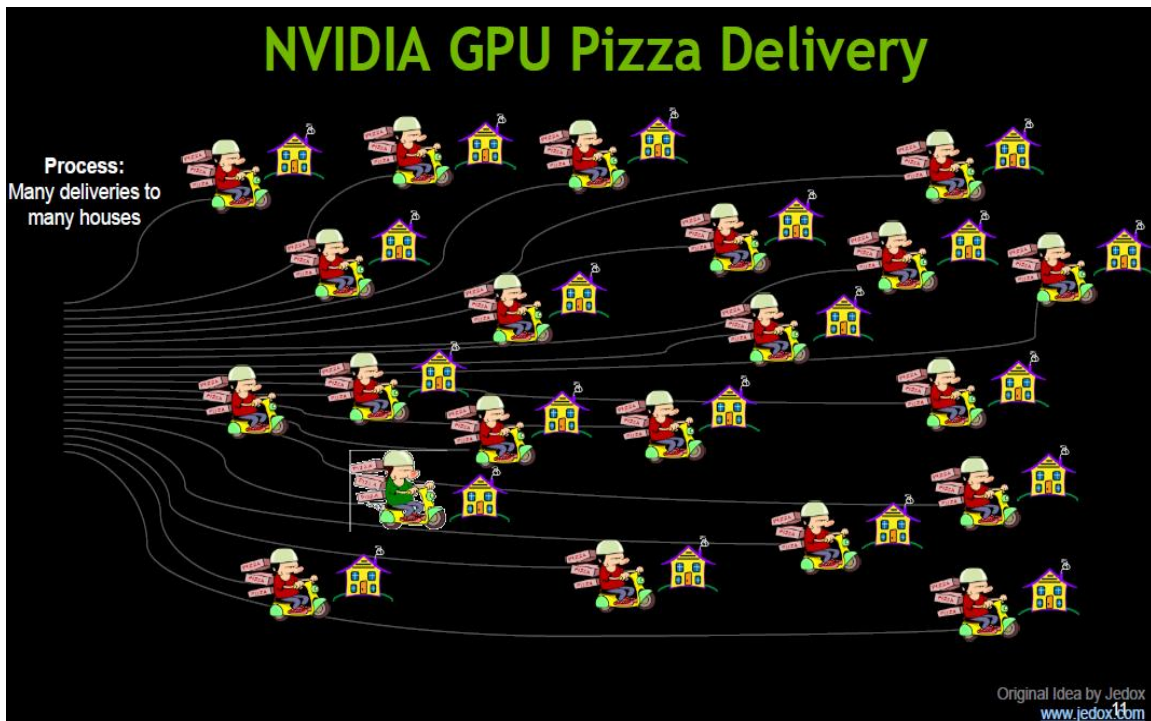
Figure 8 – A Typical GPU Operation Nvidia, 2012: 11.

What does a GPU chip look like? Cores, Caches & SM's

A GPU is largely made up of cores, caches and streaming multiprocessors (SM's). Namely, thousands of cores are divided up into blocks on SM's – an SM is essentially a mini "brain" on the chip – all the SM's added together make up the entire 'brain' or chip. Furthermore, each SM has memory caches and registers. A picture is necessary here – there are 32 cores per SM on this GPU:
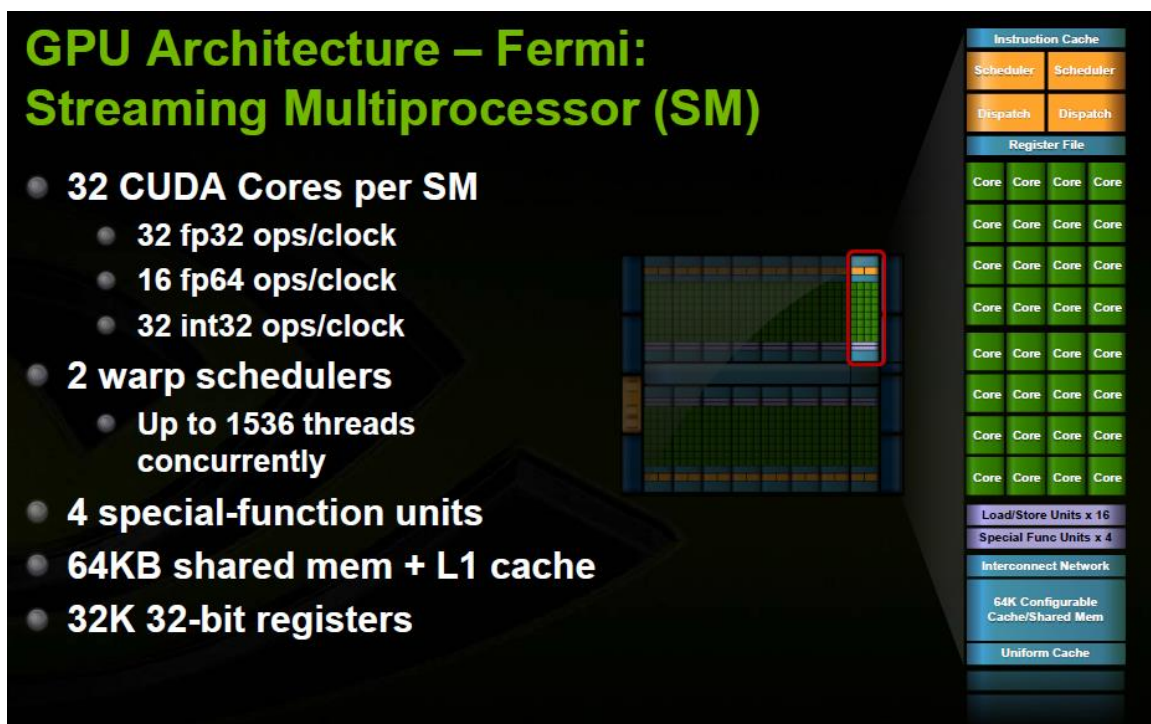
Figure 9 – GPU Architecture Nvidia, 2012: 29.

Hybrid CPU-GPU Operations: Processing Flow

The next 3 charts demonstrate how the CPU and GPU interact when processing hybrid functions or operations - CPU contents are copied to GPU DRAM, the GPU does calculations via SM's and then GPU contents are copied back to CPU:
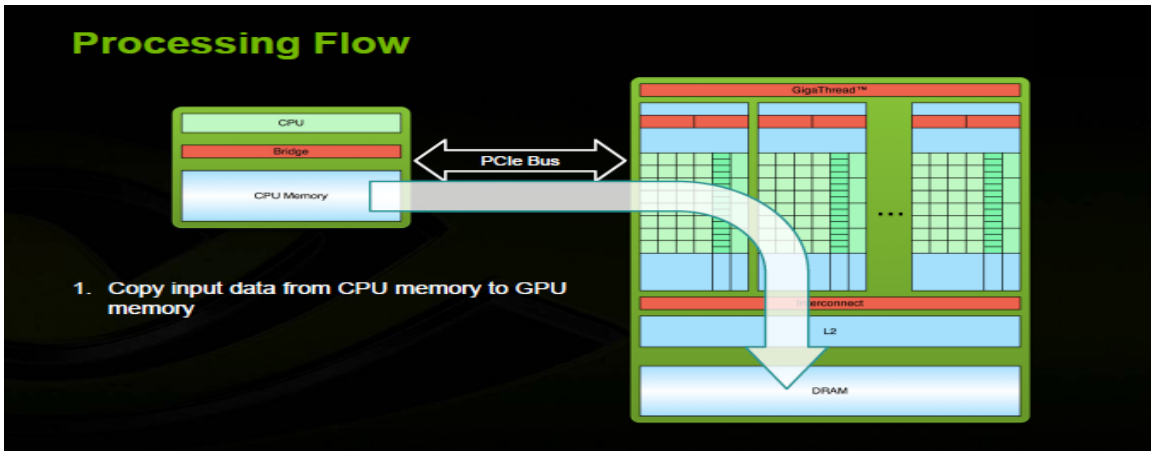
Figure 10 – A Typical Hybrid CPU-GPU Operation: CPU-to-GPU Nvidia, 2012: 24.
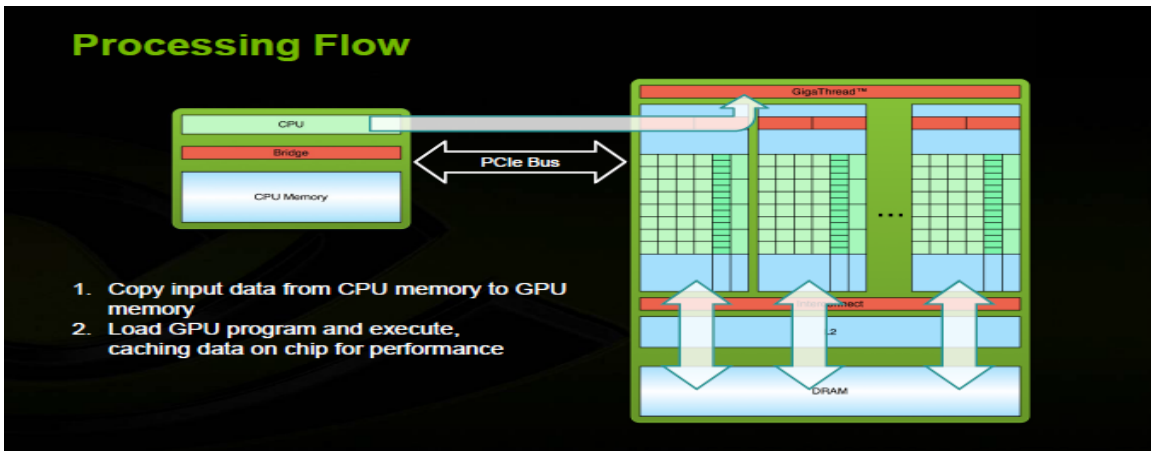
Figure 11 – A Typical Hybrid CPU-GPU Operation: GPU calculations via SM's Nvidia, 2012: 25.
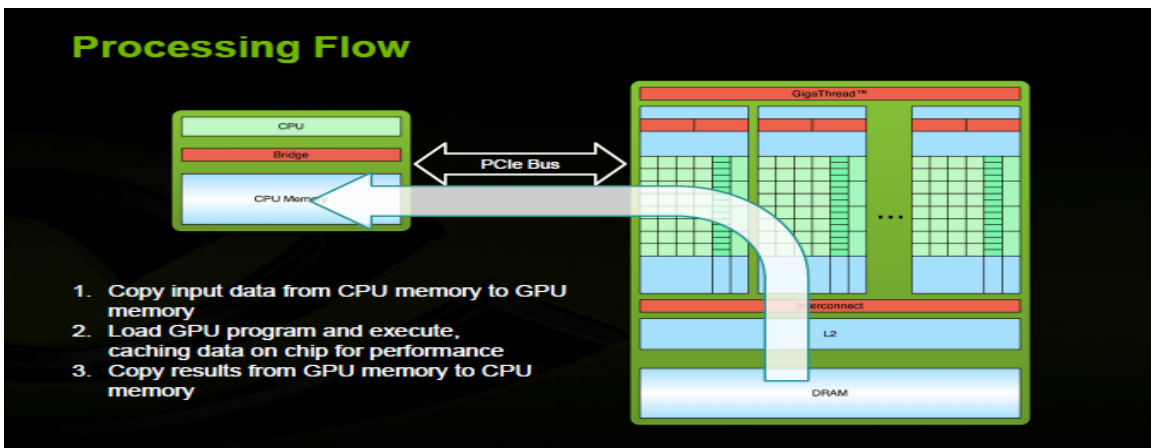
Figure 12 – A Typical Hybrid CPU-GPU Operation: GPU-to-CPU Nvidia, 2012: 26.

GPU Memory Hierarchy & Kernels: Threads, Blocks & Grids → Cores, SM's & GPU's

GPU memory breaks down into threads stored by registers in local memory. Groups of threads then make up blocks stored in shared memory. Blocks then make up grids which encompass global memory. In terms of kernel or function execution, threads map to cores while blocks map to SM's. Finally, grids can take up the entirety of the GPU or even multiple GPU's via multi-streaming, concurrent execution – these 2 slides show it best:
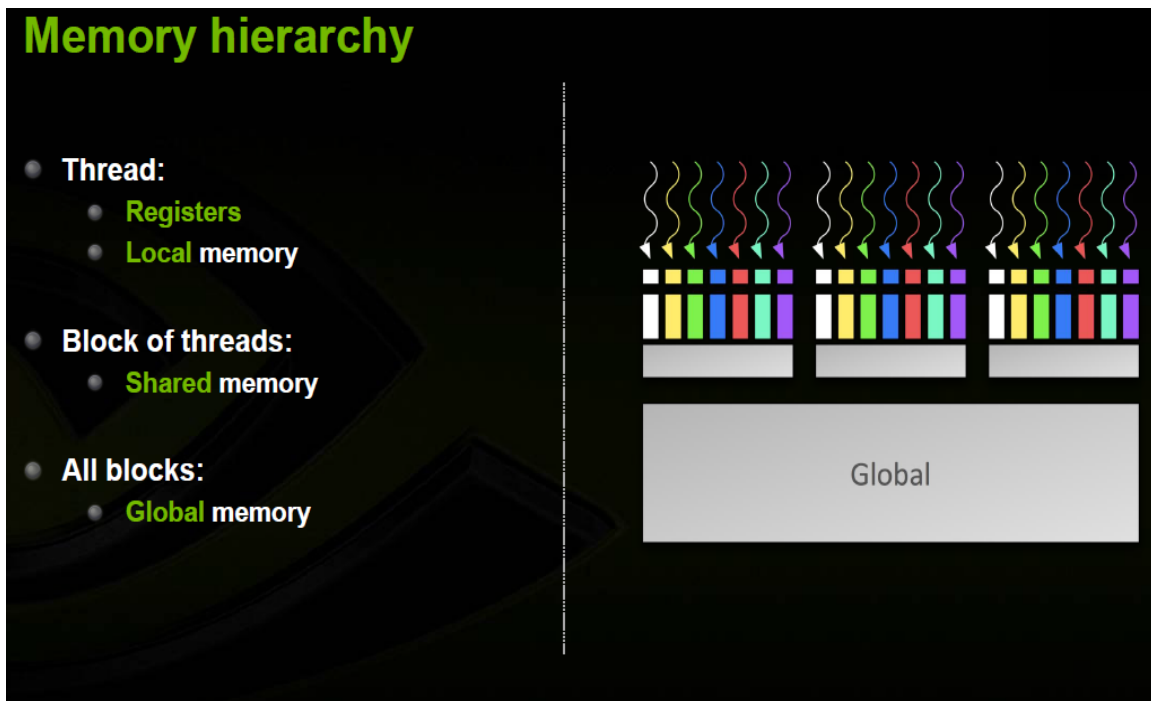
Figure 13 – GPU Memory Hierarchy Nvidia, 2012: 27.
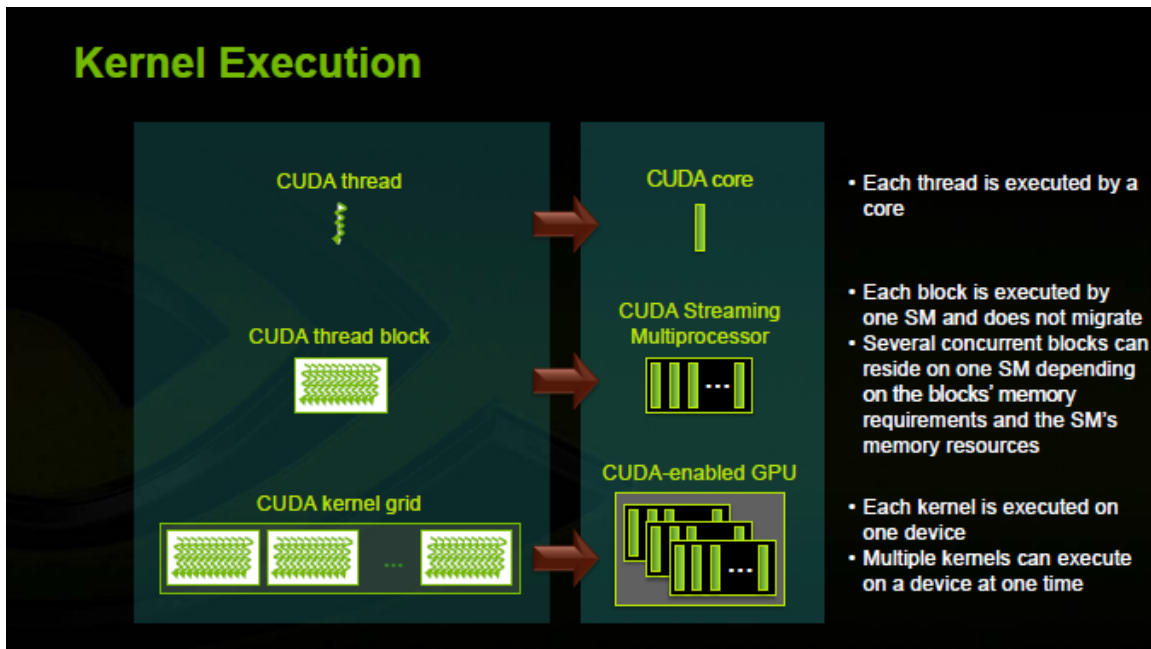
Figure 14 – GPU Kernel Execution Nvidia, 2012: 43.

A Simple CUDA C Function

The simple function from Nvidia shows the difference between C code and CUDA C – CUDA C is just an extension of C with some additional syntax – the CUDA C compiler is called "NVCC" and is very similar to the standard C compiler. They are similar except the way they declare and call functions as well as memory allocation. In CUDA C, you typically use "**_global_ void**" rather than just "**void**" to declare a function. Also, you must use the triple chevron **"<<< blocks, threads >>>"** to call a CUDA function. Lastly, you cannot use "**calloc**" to allocate memory to GPU-related variables – you must use "**malloc**" instead:

Figure 15 – C Function versus CUDA C Function Nvidia, 2012: 34.

A Simple CUDA C Program

In the following CUDA C program, we're simply going to make a CUDA

function that adds the number 10 to each subsequent iteration of a variable. We'll

then call that function in the **main** program and print the results to the console.

There are two versions here – one without error trapping as well as the same

version showing how to trap CUDA errors – it is the same basic program, overall:

Exhibit 2 – Simple CUDA C Program *Author, 2013.*

```c
#include <stdio.h>

// For the CUDA runtime library/routines (prefixed with "cuda_") - must include this file
#include <cuda_runtime.h>

/* CUDA Kernel Device code
 * Computes the vector addition of 10 to each iteration i */
__global__ void kernelTest(int* i, int length){

    unsigned int tid = blockIdx.x*blockDim.x + threadIdx.x;

    if(tid < length)
        i[tid] = i[tid] + 10;}

/* This is the main routine which declares and initializes the integer vector, moves it to the
device, launches kernel
 * brings the result vector back to host and dumps it on the console. */
int main(){

//declare pointer and allocate memory for host CPU variable - must use MALLOC or CudaHostAlloc
here
    int length  = 100;
    int* i = (int*)malloc(length*sizeof(int));

//fill CPU variable with values from 1 to 100 via loop
    for(int x=0;x<length;x++)
        i[x] = x;

//declare pointer and allocate memory for device GPU variable denoted with "_d" – must use
cudaMalloc here
    int* i_d;
    cudaMalloc((void**)&i_d,length*sizeof(int));

//copy contents of host CPU variable over to GPU variable on GPU device
    cudaMemcpy(i_d, i, length*sizeof(int), cudaMemcpyHostToDevice);

//designate how many threads and blocks to use on GPU for CUDA function call/calculation - this
depends on each device
    dim3 threads; threads.x = 256;
    dim3 blocks; blocks.x = (length/threads.x) + 1;

//call CUDA C function - note triple chevron syntax
    kernelTest<<<threads,blocks>>>(i_d,length);

//wait for CUDA C function to finish and then copy results from GPU variable on device back to CPU
variable on host – this is a blocking operation and will wait until GPU has finished calc process
    cudaMemcpy(i, i_d, length*sizeof(int), cudaMemcpyDeviceToHost);

//print results of CPU variable to console
    for(int x=0;x<length;x++)
        printf("%d\t",i[x]);

//free memory for both CPU and GPU variables/pointers – must use cudaFree here for GPU variable
        free (i); cudaFree (i_d);

//reset GPU device
        cudaDeviceReset();  }
```

Annotations:

- This one file must be included here for base CUDA functionality, but can also include others
- Thread id # (tid) is function of BOTH thread and BLOCK since thread count resets to 0 on next contiguous block
- Must use MALLOC or CudaHostAlloc for host CPU variable – cannot use CALLOC like in normal C
- Thread & block sizes can vary and depend on size of data and GPU device – run CUDA tests later on for optimality, but **MUST** assign an initial value for the number of threads which is **256** here and is a fairly standard size
- This is a BLOCKING operation since transfer will WAIT until GPU is done processing CUDA function calcs (see CUDA call above)
- This resets GPU device and all local contexts including the thread counter index – this is important for profiling CUDA application later on

https://bitbucket.org/adrew/gpu/commits/7e8154cf89bfc312adbf899187d89622

Exhibit 3 – Simple CUDA C Program with Error Trapping Author, 2013.

```c
#include <stdio.h>
// For the CUDA runtime routines (prefixed with "cuda_")
#include <cuda_runtime.h>
#include <cuda_runtime_api.h>

//CUDA Kernel Device code - Computes the vector addition of 10 to each iteration i
__global__ void kernelTest(int* i, int length){

    unsigned int tid = blockIdx.x*blockDim.x + threadIdx.x;

    if(tid < length)
        i[tid] = i[tid] + 10; }

 /* This is the main routine which declares and initializes the integer vector, moves it to the
device, launches kernel and brings the result vector back to host and dumps it on the console. */
int main(void){

        // Error code to check return values for CUDA calls
    cudaError_t err = cudaSuccess;

    int cumsum[200]={0},x=0;
    int length  = 100;
        printf("[Vector multiplication of %d elements]\n", length);

        // Allocate the host input vector A
    int* i = (int*)malloc(length*sizeof(int));

    for(int x=0;x<length;x++)
        i[x] = x;

        // Allocate the device input vector
    int* i_d;
    err=cudaMalloc((void**)&i_d,length*sizeof(int));

        if (err != cudaSuccess)
    {
        fprintf(stderr, "Failed to allocate device matrix  (error code %s)!\n",
        cudaGetErrorString(err));
        exit(EXIT_FAILURE); }

        // Copy the host input vector A in host memory to the device input vectors on GPU
        printf("Copy input data from the host memory to the CUDA device\n");
    err = cudaMemcpy(i_d, i, length*sizeof(int), cudaMemcpyHostToDevice);

        if (err != cudaSuccess)
    {
        fprintf(stderr, "Failed to copy matrix from host to device (error code %s)!\n",
        cudaGetErrorString(err));
        exit(EXIT_FAILURE); }

        // Launch the Vector Add CUDA Kernel
    dim3 threads; threads.x = 256;
    dim3 blocks; blocks.x = (length/threads.x) + 1;
    kernelTest<<<threads,blocks>>>(i_d,length);

        err = cudaGetLastError();

    if (err != cudaSuccess)
    {
        fprintf(stderr, "Failed to launch vectorMultiply kernel (error code %s)!\n",
        cudaGetErrorString(err));
        exit(EXIT_FAILURE); }

         // Copy the device result vector in device memory to the host result vector
        printf("Copy output data from the CUDA device to the host memory\n");
        err = cudaMemcpy(i, i_d, length*sizeof(int), cudaMemcpyDeviceToHost);
```

> We can also use "error trapping" in the same program to show us where there are problems in CUDA functionality as we learn – "stderr" is the basic error flag or stream in C denoting an error in a particular operation upon which it will print to the console and exit

```
    if (err != cudaSuccess)
{
    fprintf(stderr, "Failed to copy matrix from device to host (error code %s)!\n",
    cudaGetErrorString(err));
    exit(EXIT_FAILURE); }

for(int x=0;x<length;x++)
    printf("%d\t",i[x]);

    // Verify that the result vector is correct
for (int x = 1; x <= length; ++x)
{
    cumsum[x] = cumsum[x-1]+i[x]; }

    if (cumsum[length-1]+i[0] != 5950)
    {
        fprintf(stderr,"Result verification failed at element %i!\n", cumsum[length-1]);
        exit(EXIT_FAILURE); }

    // Free host and device memory
free(i); cudaFree(i_d);

// Reset the device and exit
err = cudaDeviceReset();

if (err != cudaSuccess)
{
    fprintf(stderr, "Failed to deinitialize the device! error=%s\n", cudaGetErrorString(err));
    exit(EXIT_FAILURE); }

printf("Done\n");
    return; }
```

Anatomy of a CUDA C Program Flow & Compilation

These slides simply depict what occurred in the previous CUDA C program:
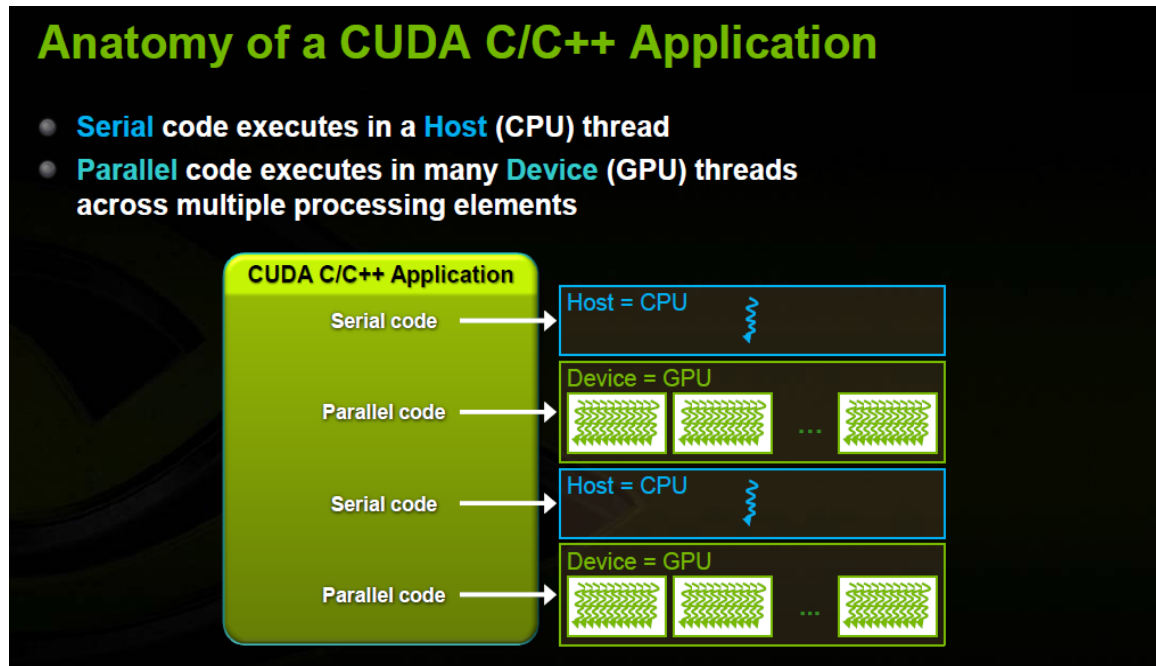
Figure 16 – Anatomy of CUDA C Program Nvidia, 2012: 32.
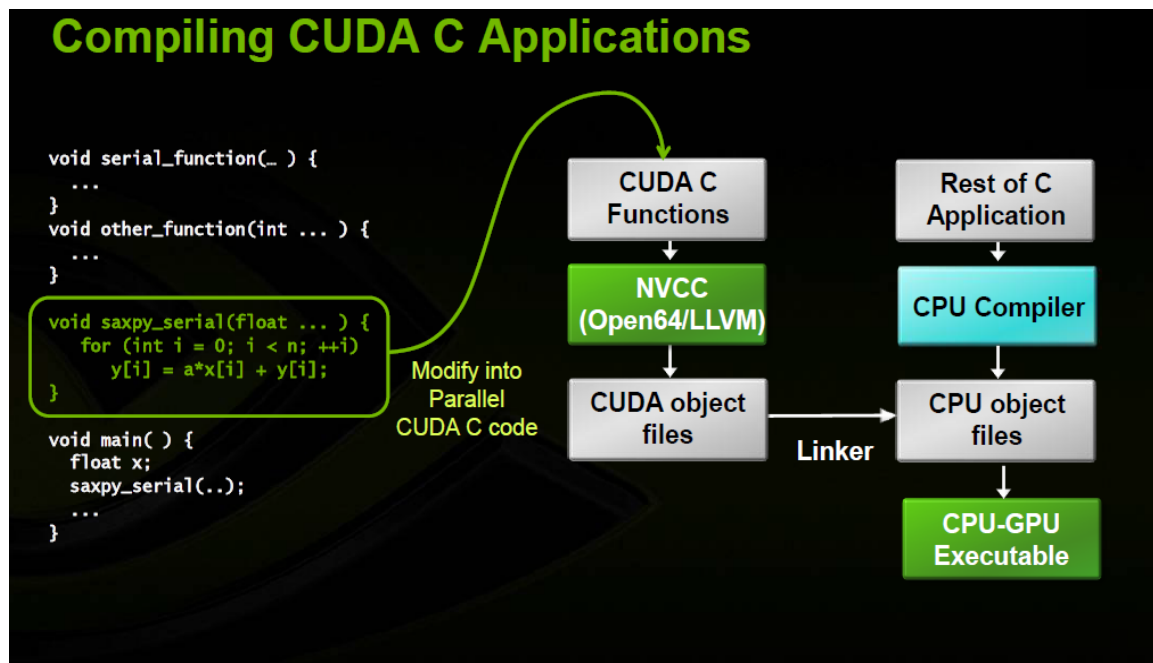
Figure 17 – Compilation of CUDA C Program Nvidia, 2012: 33.

A More Complex CUDA C Program: Model Optimization in Finance

A Primer in Finance & Building Models – The "Martians" Have Landed

The CUDA functions and programs demonstrated so far are simple in nature. This is well and good, but we really want to test how CUDA can speed up a more complex program. In science and finance, we are often dealing with models of the world and trying to speed up intensive computations. To do this, we should build a simple model that has many iterations and calculations and run it on the CPU (via C) as well as on the GPU (via CUDA C). We want to determine the benefits of using the CPU, alone, as well as using the CPU along with the GPU in hybrid fashion. We will build a trading model in finance in C as well as CUDA C, so, there will be two versions here. Finance may not be socially useful, but it is still intellectually challenging since "beating" the market on a risk-adjusted, consistent basis is very difficult to do without cheating – we are simply using our brains and probability theory here instead of relying on "insider" information or a legalized version of "front-running" (e.g., high-frequency trading via "flash" quotes). Some enjoy cross-word puzzles while others enjoy finding patterns in the market.

Let's take a step back and think about what a 'model' is. It is also helpful to give a brief primer in finance so the reader has some context as to what we're striving to accomplish in this demonstration. A model is nothing more than a simplistic view of the world that describes some event. The main components of a model are typically referred to as "**factors**" or variables. That is, we want to try and find factors that are helpful in describing some state of the world. We could use the computer to sift through thousands of potential factors and specify the relevant

variables – this is referred to loosely as "machine learning" (or data mining) and techniques like neural nets (ANN) and regressions (PLS) are of this type. However, we could also observe the world and then specify the factors, ourselves, via experience and then test their usefulness. That is what we'll do here.

Models are often described in mathematical form. Thinking back to a regression equation from grammar school, we could have: **y = X * w +** ....so, '**X**' is the factor or variable here describing 'y' while 'w' is the weight or probability of 'X'. However, we don't have use math to describe our model – we could also use logic or conditional language such as "***IF X THEN Y...***" – this type of language is intuitive and well-suited to programming since computer code is also written in pseudo-language. In finance, we typically mean: ***IF CONDITION X THEN EXCESS RETURNS ON Y*** (e.g., the implicit modeling or prediction of RETURNS on a stock or security). So, we are "relaxing" math constraints here and will describe (and program) our model in conditional language. This "less" precise approach affords us more flexibility to test "fuzzier" factors we might not have initially envisioned via the strict math method. In fact, this type of approach is often referred to as "fuzzy logic" in that it can be "loose" or imprecise.

Fuzzy logic and set theory was pioneered in the last century by a brilliant thinker named Kurt Gödel[4]. Many brilliant thinkers emerged out of the collapsed Austro-Hungarian after World War I including Gödel and it is important to mention them since most of the tools we use today were created by them – we owe a collective "tip of the hat" to this group. These thinkers laid the foundation for

---

[4] James Gleick, The Information: A History, a Theory, a Flood (New York: Pantheon, 2011) 136-186.

modern math and physics as well as computers. Humorously, this group made up of the likes of John Von Neumann, Edward Teller and Gödel were often referred to as the "Martians" since they were considered so smart that it was if they came from another planet[5]. It could be argued that modern generations have not come as far since we have not developed new fields in math or science though we have more to go on via computing power. Remember also that the last large space operation occurred in the 1960's when von Braun and his rocket scientists sent us to the moon – a remarkable feat considering the limited computing power at the time compared to nowadays.

In the 1931, Gödel published a set of logical theories including the "**incompleteness theorem**". At the time, it was trendy for both mathematicians and physicists to tinker in philosophy. In fact, thinkers like Bertrand Russell at Cambridge University were actually trying to make philosophy a science by applying various rigorous disciplines. Namely, Russell and several others were trying to create a "perfect" human language with no ambiguities based on Boolean logic and mathematics[6]. The idea was that one could represent linguistic expressions as mathematical conditions with binary (e.g., true-or-false) outcomes and, therefore, switch back-and-forth between the two modes of expression. Today, this endeavor might seem trite or foolish, but the hope was that a clear language could rid the world of misunderstanding and suffering. A mathematician, himself, Gödel came along and "wrecked" this logical quest though Wittgenstein also made some contributions to the cause. With the "incompleteness theorem", Gödel

[5] Istran Hargitta, Martians of Science (New York: Oxford Press, 2006) 11.
[6] Gleick 136-186.

demonstrated how a mathematical function can be logically constructed in language and vice-versa, but still remain improvable though simultaneously consistent or rational. So, he demonstrated how something logical could still end up being circular or incomplete in some sense (e.g., always lacking or imperfect). The long, formal proof of this theorem is beyond the scope of this study, but let it suffice that the two main points are that if a system is consistent or logical, then it cannot be complete and its axioms cannot be proven within the system. With one master stroke, Gödel basically pushed "back" math, physics and philosophy into "grey" space despite the best efforts of thinkers like Russell to make these disciplines binary or "black-and-white". So, Gödel brought back ambiguity and made it "okay", so to speak, since he showed that statements that are improvable can still have meaning or consistency. Since he used both math and language to illustrate his proof, his findings could not be refuted by mathematicians and philosophers, alike, which is why they had such far-reaching implications.

This theme of "greyness" or fuzziness is important because it relates back to our model in finance. Loosely speaking, the word "fuzzy" basically implies that something has meaning or usefulness though it is ambiguous. Mapping this theme into logic or probability theory, "fuzzy" means that something is not **true** or **false** in a strict sense, but rather has a degree of truth or consistency between **0** and **1**. So, a statement has a probability or outcome expressed as an interval or percentage between the binary levels of 0 and 1. Essentially, fuzzy logic implies a lack of precision and is a matter of degree or range, instead, which is very relevant to human language via conditional statements. It is as if to admit that there is only a

"sense" of the truth in human affairs which are often complex. Imagine someone making a stock market prediction and framing it as a matter of probability rather than certainty which seems prudent, since, it is so difficult to make prognostications about the future. A typical fuzzy statement is the following: "***Unexpected results of government reports cause big moves in the stock market***". Intuitively, this statement seems somewhat true or meaningful though it is not exactly quantified and, hence, ambiguous. This is the type of expression or outcome Gödel was trying to describe in his proof. Now imagine a computer trying to process the statements "cold, colder, coldest". Though these words have some "rough" meaning to humans, computers cannot quantify these expressions unless a range of temperatures (e.g., a "fuzzy" interval) describing each subset or word is also supplied. More formally in 1965, Lotfi Zadeh mathematically described a "fuzzy" set as a pair (**A,m**) where **A** is a set and **m** : **A** $\rightarrow$ **[0,1]**[7]. A fuzzy set or interval of a continuous function can also be written: $\tilde{A} \subseteq \mathbb{R}$. In terms of modeling, it's more useful to think of a fuzzy set in the form **A ● R = B** where A and B are fuzzy sets and **R** is a fuzzy "relation" – A ● R stands for the composition A with R[8]. So, what does this mean? It means that though sets A and B are originally independent, they might be probabilistically connected through **R**, the fuzzy relation, by one element in both sets. It is like saying A and B are independent, but that they could also be probabilistically related to each other via R – there is a sense of "fuzziness" or ambiguity here.

---

[7] Feb. 2013 < http://en.wikipedia.org/wiki/Fuzzy_set >.
[8] Feb. 2013 < http://en.wikipedia.org/wiki/Fuzzy_set >.

Now we have fuzzy logic under our belts, let's go ahead and specify our fuzzy model. Note that we will not formally or explicitly define our model and its factors via math, but do this implicitly, instead, via language. From observations and experience, we know that financial markets get interesting at the extremes. What does "extreme" mean? In terms of price action, this idea pertains to when markets get **overbought** (e.g., price has risen very high) or **oversold** (e.g., price has dropped very low). So, we want to build a simple model that can give us some insight into what happens when prices reach extremes. Should we buy on strength or "momentum" when prices are very high and overbought? Or, should we sell short in a kind of counter-trend or **mean-reversion** trade? Note that this relates to human behavior to some degree since people often engage in "crowding" actions akin to "fear-and-greed" cycles. So, this model will also have a behavioral edge as well as underpinnings to fuzzy logic.

To discover more, we must first find a way to measure extremes. From experience, we will use a statistical "trick" or tool called "**Z-scores**". A z-score is simply defined as: $z = (a - ā)/σ$. So, take today's price for a security and subtract from it a moving-average price and then divide by the standard deviation (or volatility) of the time series. At any given time, this measure will tell us how many standard deviations "**rich**" or "**cheap**" the current price is when compared to the average price. Sounds simple, but remember that we already have two parameters of this function – a parameter is simply an input into a function. Namely, we have **length** and a suitable cutoff level or fuzzy **range** that defines 'rich' or 'cheap'. So, our trading rules becomes: **IF Z-SCORE TODAY IS >= Z-SCORE RANGE THEN BUY**

(and vice-versa for SELL trades). From testing and experience, we happen to know that "going with the market" (instead of fighting it) yields better results – so, this type of model will be a **momentum** model since we are simply reacting to and 'going with' the recent trend in prices. Notice also that we only care about price here – we don't care about **exogenous**, fundamental variables like GDP (Gross Domestic Product). So, we are **keying** off "price action" to make a formulation about trading on the price of a security – there are no abstract levels of determination here since we are studying price to take action on price, itself. This type of model is often referred to as a technical or price-based model and is **endogenous**.

So, we have a relatively simple model here with one implicit factor called z-score. However, we'll need to test various values to determine the optimal parameters of the model – length and z-score range. This will involve combinatorial math. Let's say we have 3 possible inputs for these 2 parameters: {**1, 2, 3**} for z-score ranges and {**21, 34, 55**} for moving-average lengths – that is 3^2 combinations or 9 in total. Models can often have 2, 3 or even 4 factors – so we could easily have 3^4 or 81 combinations for testing. This means for each security, we must test **all** combinations over each time period to find the best one. This is called "**exhaustive optimization**" since we are testing all combinations – **genetic** optimization is faster, but there is no guarantee of finding the best result. What is the '**best**' one? In finance, this is often called lambda, **λ**, or the Sharpe ratio: **μ/σ** (the average return over the time period divided by the volatility over the same time period). This is not to be confused with lambda or "half-life" from physics. There is an old joke that if your math is not good enough, you leave science and go into finance. All jokes aside,

finance does indeed borrow a lot from physics – delta, gamma, lambda and omega are all used in option trading besides the fact that the Black-Scholes pricing equation (for options) was derived from the heat-transfer equation in physics (e.g., Ito's Lemma). So, we'll have to run all these combinations and then store the best combination for use later in the model for each security. So, 30 securities (though we could easily have more) multiplied by 9 combinations is a total of 270 optimizations. We can begin to see that the amount of computation is piling up here.

To add more realism to this model testing, we'll also have to repeat this optimization over-and-over for each time period on each security. So, as we move through time, our optimization process becomes a "rolling" optimization. We must look "back" and test the best possible parameter combination and then use that one for the next period forward to derive unbiased model returns (e.g., "profit-and-loss") for each security. The "look back" period is often referred to as an "**in-sample**" test while the next one forward is called an "**out-of-sample**" test (e.g., "ex-ante" versus "ex-post"). Remember that once we've found the best parameter combination on a security, we cannot go back in a time machine and trade off these results – so, we'll simply hold these parameters constant and use them for the next period forward in terms of trading and results. Since this model will be re-optimizing and **updating** with the market, it is also a kind of learning model or **DLM** (Dynamic Learning Model). The world of finance has been waiting patiently for the physicists to find the "time machine" model, but nothing has been found as of yet though quantum theory suggests this might be possible. This concept can be

confusing and is a common mistake amongst modelers so let's draw a timeline depicting this **rolling** optimization – here we are fitting parameters (for z-score) on the last 5 years and then repeating this process every year (or re-optimizing in **steps** of 1):

	Year 1 (Time 1) Year 2 (Time 2) Year 3 (Time 3) Year 4 (Time 4) Year 5 (Time 5) Year 6 (Time 6)
In-Sample Period 1 (2 & 21 best params over 5 years) Out-of-Sample Period 1 (2 & 21 params for next year forward)	
	Year 2 (Time 2) Year 3 (Time 3) Year 4 (Time 4) Year 5 (Time 5) Year 6 (Time 6) Year 7 (Time 7)
In-Sample Period 2 (1 & 34 best params over 5 years) Out-of-Sample Period 2 (1 & 34 params for next year forward)	
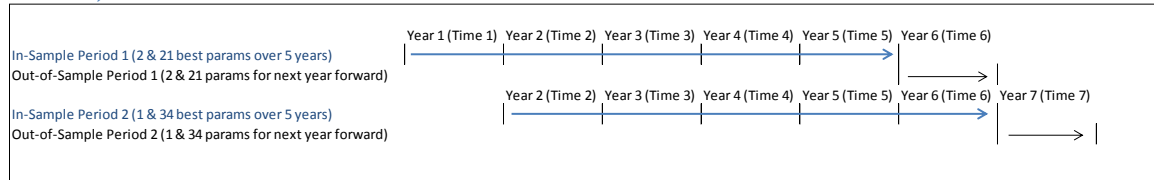
Considering many financial time series (in the futures markets) have an average length of at least 39 years, than mean 35 in-sample optimizations in total. So, now we have [**35 periods * 9 combinations * 30 securities**] equals **9,450** optimizations/combinations in total. Even though we have a simple 1-factor model, the amount of calculations increases exponentially. A more complex model could have [**35 periods * 81 combinations * 30 securities**] equals **85,050** optimizations/combinations in total. This amount of computation will easily overwhelm any CPU (including quad-cores) on a workstation or desktop which is really why we need GPU's here as well as multiple workstations (and GPU's). In the real world, no model is perfect since it is merely a simple representation of the world. Therefore, it is common in finance to run multiple models **simultaneously** in order to diversify the portfolio in an extreme sense. Instead of diversifying a portfolio of stocks according to finance theory, take this theme a level **higher** and imagine becoming a portfolio manager of models (instead of just securities). Now envision this: [**35 periods * 81 combinations * 30 securities * 30 models**] =

2,551,500 optimizations/combinations in total. Typically, re-optimization is done every week instead of every year like the example above (e.g., **smaller steps**) – *Houston we have a problem*! Hopefully, everyone gets the gist of the computational challenges in the world of finance and science.

An Overview of the Model Program Flow, Memory & Process

Since there are thousands of line of code between the CPU and GPU versions of the model program, it is impractical to highlight all the code here. However, here are some highlights as well as some general comments on the process:

- It's a good habit to have all functions in a utility header or source file (utility.h or utility.c) and then include this in the **MAIN** PROGRAM **(#include "utility.h")** – put any needed files inside the project folder so there is no problem with "paths" – be sure to include the **cuda_runtime.h** as well since this is most important and is a requirement.

  - Instead of making a new CUDA project from scratch, it can be easier to take an example from Nvidia and then customize it further. To start one from scratch, we must be sure to change the **BUILD CUSTOMIZATION** in the project by selecting the CUDA/NVCC compiler. Also, the following library paths must be added in under the project properties: **cudart.lib** under LINKER/INPUT and **$(CUDA_PATH)** under LINKER/GENERAL

- As for code, in order to get the combination of parameters, we'll need to use a series of outer/inner **for-next** loops. In the example here, there are **4** "**for-next**" loops: **LOOP BY MARKET >>> LOOP BY TIME PERIOD >>> LOOP BY FIRST PARAMETER >>> LOOP BY SECOND PARAMETER...**

  - We'll run an optimization based on the previous rolling **6** years and then "re-fit" every year or in steps of **1** (though this is variable by the user via #define)

  - We'll store the best parameter combination and then use it for the NEXT PERIOD FORWARD so that our results are NOT biased – we want to collect OUT-OF-SAMPLE

results here – remember, we cannot go back in a time machine and use the parameters we discovered today for yesterday's trading – biasing returns with IN-SAMPLE information is a common mistake. If we do so, any model can look good given the amount of computing power we have nowadays (e.g., the computer can always find some setting or parameter that looks like it "works")…

- o For CUDA C, try to use "page-locked" or "**pinned**" memory as much as possible for speed gains – store CPU variables or pointers using **CudaAllocHost** (instead of **MALLOC**) so that the CPU and GPU can more quickly transfer data back-and-forth. Remember, transferring data between the CPU and GPU is the **SLOWEST** operation (or biggest "bottle-neck") so try to do so sparingly by running as many calculations on the GPU before sending the results back to the CPU. Remember that the MAX MEMORY (DRAM) on ANY GPU is **6 GIGS** so be mindful of large memory operations. Here is a visualization of **pinned** versus **non-pinned/pageable** memory:
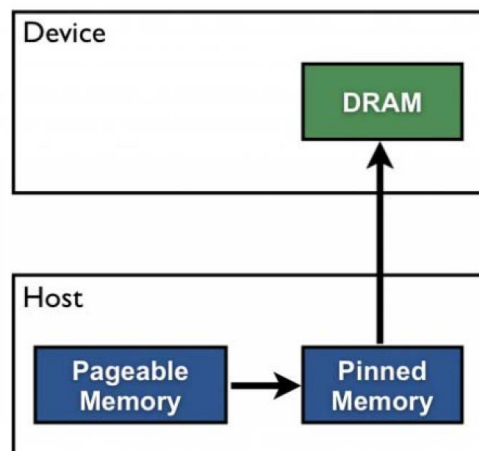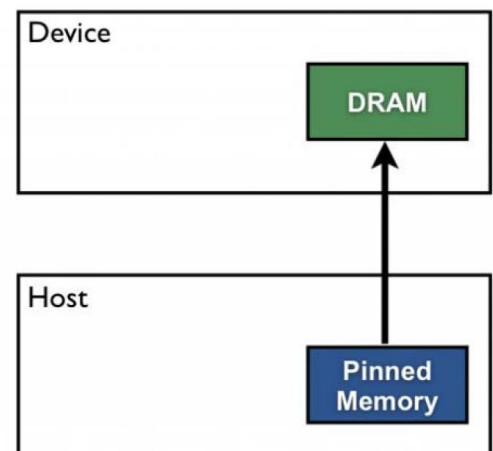
- o Also, depending on the capability of the video card, try to use **ASYNCHRONOUS** (overlapping) operations as much as possible – use **cudaMemcpyAsync** instead of **cudaMemcpy**. This means that the CPU launches an operation on the GPU and does NOT have to wait for the GPU to finish and so the CPU simply moves on to the next operation (e.g., this type of memory/data transfer is not a "blocking" operation)

- Try using single-precision or **FLOAT** as much possible – don't declare variables as DOUBLE – or, **CAST** them to FLOAT as you feed them to the GPU – the GPU is OPTIMIZED for SINGLE-PRECISION (whereas the CPU is geared for DOUBLE-PRECISION) so you'll see up to **6X** the speed gains if you do this – after all, do we really need up to 16 decimals? Be careful switching back-and-forth between FLOAT and DOUBLE since there can be problems with **precision** due to **PROMOTION** – use **FLOAT LITERALS** whenever possible to avoid this (e.g., **constant = 1.0f** instead of **constant = 1.0**) – also, be sure to use **CudaMemset** to initialize GPU variables to **ZERO**

- Try to use BLOCK sizes that are multiples of **WARPS** (e.g., 32 threads on most GPU devices) or half-warps for more efficient memory alignment and coalescing

- If you have multiple GPU's or video cards, launch several GPU operations simultaneously (or **concurrently**) on various cards – this is called MULTI-STREAMING – this is not possible on most home setups due to hardware constraints

A Closer Look at the Model Program – Computer Code

Here we can see more complexity as opposed to the earlier examples in

CUDA C:

Exhibit 5 – A Complex CUDA C Program: Trading Model Author, 2013.

```
#include <stdio.h>
#include <math.h>
#include <io.h>
#include <stdlib.h>
#include <string.h>
#include <float.h>
#include "utility.h"
#include "utility.c"
#include <time.h>
#include <malloc.h>
#include <cuda_runtime.h>
#include <cuda_runtime_api.h>

//Declaring macros and constants in pre-processor - STEP A//INSERT NEW STUFF DOWN HERE EACH TIME
START SUB-FUNCTION******PRE-PROCESSOR AREA*******//
#define ABS(X) (X>=0?X:-X)
#define MAX(X,Y) (X>=Y?X:Y)
#define MIN(X,Y) (X<=Y?X:Y)
```

Additional references here: utility.h and utility.c refer to other custom functions, time.h for timer – must have malloc.h, stdio.h and cuda_runtime.h here

Instead of formally making functions, we can make some simple ones here in the pre-processor – this is a nice trick in C

```
#define SIGN(X) (X>=0?(X==0?0:1):-1)
#define ROUND(X,Y) ((X>=0?(X<<1)+Y:(X<<1)-Y)/(Y<<1))*Y

//Change path below for UNIX "c://usr"
#define PATH "C:\\"
#define LOOKBACK 1597     // 1597-987-610-377-144-89 fibos rolling optimization historical period
#define STEP 377          // or 89 fibos step forward in time period for next rolling optimization
#define NUMI 27           //up to 27 number of markets

__global__ void kernelSim(float *zscores_d,float *rets_d,float *pnl_d,float *pos_d,int start,int
stop,float zcut,int lens){

        const float buy =1.00f;
        const float sell=-1.00f;
        const float flat=0.00f;

    //Thread index
        unsigned int tid = blockDim.x * blockIdx.x + threadIdx.x;
    //Total number of threads in execution grid
        unsigned int THREAD_N = blockDim.x * gridDim.x;

    //No matter how small is execution grid or how large OptN is,
    //exactly OptN indices will be processed with perfect memory coalescing
        for(int opt = tid+start; opt < stop; opt += THREAD_N){

            if(zscores_d[opt] > zcut && opt >=lens) pos_d[opt] = buy;

            if(zscores_d[opt] < -zcut && opt >=lens) pos_d[opt] = sell;
            if(opt >=lens && (pos_d[opt]==buy || pos_d[opt]==sell)) pnl_d[opt] =
            (pos_d[opt] * rets_d[opt]);
            else {pnl_d[opt] = flat; pos_d[opt] = flat; } } }

//INT MAIN//INSERT NEW STUFF HERE EACH TIME START SUB-FUNCTION*******MAIN AREA****//Declare each
new variable here - initializing and declaring space/memory for return arrays of variables or
output we want****STEP B//
int main(int argc, char **argv){
        //array holder for parameter combinations later on aka "parameter sweeps" which GPU can
        greatly speed up// a[] is # standard deviations
        double a[] = { 1.25, 1.50 };
        double b[] = { 21.00, 34.00 };
        //** ALLOCATE SPACE FOR MEMORY FOR CPU VARIABLES
        cudaHostAlloc(&rets, (int)(end)*sizeof(float), cudaHostAllocDefault);
        cudaHostAlloc(&pos, (int)(end)*sizeof(float), cudaHostAllocDefault);
        cudaHostAlloc(&pnl, (int)(end)*sizeof(float), cudaHostAllocDefault);
        cudaHostAlloc(&zscores, (int)(end)*sizeof(float), cudaHostAllocDefault);

        sharpp=(double*) calloc(end+1,sizeof(double));
        sumip=(double*) calloc(end+1,sizeof(double));

        //** ALLOCATE SPACE FOR MEMORY FOR CUDA-RELATED DEVICE VARIABLES**
        cudaMalloc((void**)&zscores_d, (int)(end)*sizeof(float));
        cudaMalloc((void**)&pos_d, (int)(end)*sizeof(float));
        cudaMemset(pos_d, 0, (int)(end)*sizeof(float));
        cudaMalloc((void**)&pnl_d, (int)(end)*sizeof(float));
        cudaMemset(pnl_d, 0, (int)(end)*sizeof(float));
        cudaMalloc((void**)&rets_d, (int)(end)*sizeof(float));

        ret(p, end, rets);
        zscore(lens, p, sumv, varv, end, zscores, stdevv, m_avev);
        m = (lensa*z)+j;

        //** COPY CUDA VARIABLES FROM CPU (HOST) TO GPU (DEVICE) - USE ASYNC TRANSFER FOR MORE
        SPEED SO CPU DOES NOT HAVE TO WAIT FOR GPU TO FINISH OPERATION AND CAN PROCEED FURTHER IN
        THE MAIN PROGRAM**
        cudaMemcpyAsync(zscores_d, zscores, (int)(end)*sizeof(float), cudaMemcpyHostToDevice,0);
        cudaMemcpyAsync(rets_d, rets, (int)(end)*sizeof(float), cudaMemcpyHostToDevice,0);
```

Put constants here in pre-processor as well

Here is CUDA function to get model returns as well as positions (1/0/-1) – use float **literals** here for constants for precision and loop by threads/**grids** vs. threads/blocks for memory coalescing

Start of MAIN PROGRAM

Arrays for param 1 & 2 values for sweeping via 4 for-next loop optimizations

Use **cudaHostAlloc** for pinning of CPU vars in memory for more speed – can use CALLOC for non-GPU vars

Must use **cudaMalloc** here for GPU vars – should use **cudaMemset** to initialize to ZERO to avoid spurious results

Call CPU functions (from utility.c) so can pass on results to GPU vars later

Use ASYNC memory transfer **cudaMemcpyAsync** vs. cudaMemcpy to pass CPU function results to GPU vars- spec threads/block #'s by device

```
lenny=stop-start;
dim3 threads; threads.x = 896;//use 896 threads as per specific GPU device for higher ↙
OCCUPANCY/USE OF CARD - trial-and-error via PROFILING - max blocks is 112 on GTX 670 GPU

//** CALL GPU FUNCTION/KERNEL HERE FOR MODEL PARAMETER SWEEP TO GENERATE IS RESULTS
kernelSim<<<threads,112>>>(zscores_d,rets_d,pnl_d,pos_d,start,stop,(float)(a[z]),lens);
```

┌───┐
│ Call CUDA function specifying threads/blocks │
└───┘

```
//** COPY CUDA VARIABLES/RESULTS FROM GPU (DEVICE) BACK TO CPU (HOST) - MUST WAIT FOR GPU
OPERATION/FUNCTION TO FINISH HERE SINCE LOW ASYNC/CONCURRENCY ON NON_TESLA GPU DEVICES**
cudaMemcpy(pos, pos_d, (int)(end)*sizeof(float)/*stop-start*/, cudaMemcpyDeviceToHost);
cudaMemcpy(pnl, pnl_d, (int)(end)*sizeof(float), cudaMemcpyDeviceToHost);
```

┌──┐
│ Must use NON-ASYNC transfer here to push GPU results to CPU │
└──┘

```
for (i = 0; i < lensc; i++)      {
            //find best sharpe ratio from table and store it for next period
            if (table[i][3] > maxi) maxi = table[i][3];
            if (maxi == table[i][3]) high=i; }

                    sharplist[gg][0][ii] = high;
                    //row of max sharpe recap
                    sharplist[gg][6][ii] = table[high][3];
                    //max sharpe
                    sharplist[gg][1][ii] = table[high][1];
                    //param 1 recap
                    sharplist[gg][2][ii] = table[high][2];
                    //param 2 recap
                    sharplist[gg][3][ii] = table[high][4];
                    //cum ret recap
                    sharplist[gg][4][ii] = table[high][0];
                    //test number recap
                    sharplist[gg][5][ii] = gg;
```

┌────────────────────────────┐
│ Store optimization results │
│ in 3D array table so can   │
│ keep track of best param   │
│ combination and use later   │
│ on for Out-of-sample runs   │
└────────────────────────────┘

┌────────────────────────────┐
│ Free memory for GPU   │
│ variables as well as CPU   │
│ variables – cudaFree vs.   │
│ cudaFreeHost…MUST FREE or   │
│ will run out of memory!   │
└────────────────────────────┘

```
cudaFree(zscores_d);cudaFree(pnl_d);cudaFree(pos_d);cudaFree(rets_d);cudaFreeHost(pos);cudaFreeHos
t(pnl); }
```

https://bitbucket.org/adrew/gpu/commits/56e6d2665c693fce6af5bd4f3d7e325f

Profiling the Model Program for Further Speed Gains

When you install the CUDA SDK (link on previous page), NVIDIA also gives

you a built-in tool called **PROFILER**. This will allow you to find "bottle-necks" and

increase code efficiency. The main goal here is to use as **MUCH** of the GPU as

possible by utilizing **ALL** of the CORES. You can do this by iteratively changing the

THREAD and **BLOCK SIZE** when you call the CUDA C FUNCTION – this depends on

each specific video card so it's more of an art here than science (e.g., heuristic). For

this application, we got about ~ **63% OCCUPANCY** utilization (by specifying **896**

threads x **112** blocks) which is pretty good though **multi-streaming** was non-

existent due to the limitation of the NON-TESLA GPU (as well as only having one

GPU) – here is what it looks like:
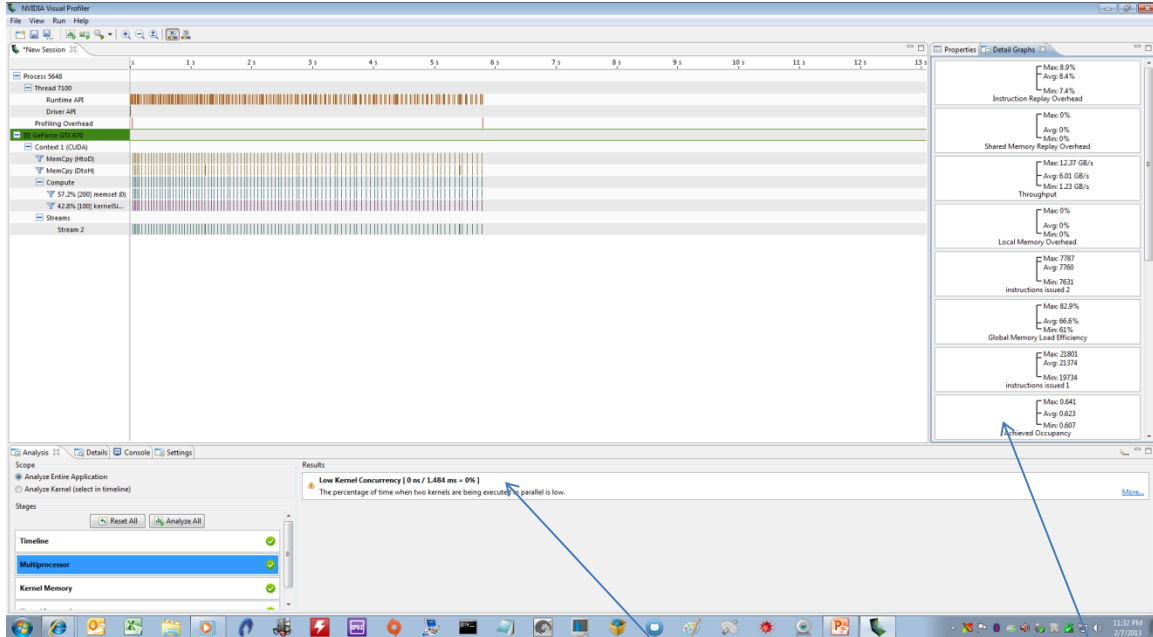
Figure 18 – Using Nvidia PROFILER to Optimize Occupancy on GPU Cores Author, 2013.

Average Occupancy at 63% - concurrency is non-existent due to lack of multiple GPU's and multi-streaming

Using Non-Nvidia/Non-Cuda Open Source Libraries for Model Program – A Look at OpenACC

Note that we don't have to use Nvidia's CUDA libraries. Instead, we could use an open-source library called "OpenACC", "OpenMP" or "OpenCL". This will allow us to use non-Nvidia GPU's like AMD/ATI for GPU programming. Since OpenCL is similar to CUDA syntax and partly based on it, let's take a look at OpenACC. OpenACC was created by the Portland Group and is made up of developers who used to work at CRAY computers. Note that this library has its own compiler, so, we'll have to get used to new syntax once again. The compiler can be downloaded from here:

http://www.pgroup.com/support/download_pgi2013.php?view=current

With OpenACC, we must remember 3 **important** rules:

- You **MUST** use **array indexing** and notation within loops – NO pointer notation or pointer arithmetic: **A[i] = B[i]** versus ***ptr A = *ptr B**

- Any OpenACC statement always begins with **#pragma acc**

- OpenACC parallel operations are essentially loops within loops – they are not fragmented into various parts like CUDA C – they tend to be more holistic

Here is the main function of the same model program written in OpenACC – we see a tiny hit to performance here versus CUDA C, but it's still a very powerful package:

Exhibit 6 – A Complex OpenACC Program vs CUDA C: Trading Model Author, 2013.

```
//Start outer loop and declare DATA REGION - list pointers/arrays for copying to and from GPU
        #pragma acc data copyin(zscores,rets,start,stop) copyout(pos,pnl)
        {
                //Declare KERNEL for ACCELERATION - sub-loop
                #pragma acc kernels
                {

                        const float buy =1.00f;
                        const float sell=-1.00f;
                        const float flat=0.00f;
                        float zcut = 0.00f;
                        zcut = (float)(a[z]);

                        //Start innermost loop to step thru values and rules to get positions and
                        returns
                        #pragma acc loop independent
                        for(int opt = start; opt < stop; opt++){
                        if(zscores[opt] > zcut && opt >=lens) pos[opt] = buy;

                        if(zscores[opt] < -zcut && opt >=lens) pos[opt] = sell;
                        if(opt >=lens && (pos[opt]==buy || pos[opt]==sell)) pnl[opt] = (pos[opt]
                        * rets[opt]);
                        else {pnl[opt] = flat; pos[opt] = flat;

                        }
                }
        }
}
```

```
#pragma acc statements
starts of a series of
nested loops – notice array
indexing as well
```

https://bitbucket.org/adrew/gpu/commits/6bebcf48b100dde9eb1f9876cf6b42a68c4c616b

To compare and contrast, let's look at the main function in the original CUDA C

program (from earlier):

```
 //** COPY CUDA VARIABLES FROM CPU (HOST) TO GPU (DEVICE) - USE ASYNC TRANSFER FOR MORE
SPEED SO CPU DOES NOT HAVE TO WAIT FOR GPU TO FINISH OPERATION AND CAN PROCEED FURTHER IN
THE MAIN PROGRAM**
cudaMemcpyAsync(zscores_d, zscores, (int)(end)*sizeof(float), cudaMemcpyHostToDevice,0);
cudaMemcpyAsync(rets_d, rets, (int)(end)*sizeof(float), cudaMemcpyHostToDevice,0);

lenny=stop-start;
dim3 threads; threads.x = 896;//use 896 threads as per specific GPU device for higher
OCCUPANCY/USE OF CARD - trial-and-error via PROFILING - max blocks is 112 on GTX 670 GPU

//** CALL GPU FUNCTION/KERNEL HERE FOR MODEL PARAMETER SWEEP TO GENERATE IS RESULTS
kernelSim<<<threads,112>>>(zscores_d,rets_d,pnl_d,pos_d,start,stop,(float)(a[z]),lens);

//** COPY CUDA VARIABLES/RESULTS FROM GPU (DEVICE) BACK TO CPU (HOST) - MUST WAIT FOR GPU
OPERATION/FUNCTION TO FINISH HERE SINCE LOW ASYNC/CONCURRENCY ON NON_TESLA GPU DEVICES**
cudaMemcpy(pos, pos_d, (int)(end)*sizeof(float)/*stop-start*/, cudaMemcpyDeviceToHost);
cudaMemcpy(pnl, pnl_d, (int)(end)*sizeof(float), cudaMemcpyDeviceToHost);
```

Use ASYNC memory transfer **cudaMemcpyAsync** vs. cudaMemcpy to pass CPU function results to GPU vars- spec threads/block #'s by device

Call CUDA function specifying threads/blocks

Must use NON-ASYNC transfer here to push GPU results to CPU

Using Wrappers and Other "Non-native" Packages for Modelling: What are the tradeoffs of using these alternatives?

The MATLAB Stats package

Instead of writing CUDA on the lowest, most pure level (e.g., **kernel** level),

we can use "**wrappers**" in other programming packages, instead. Wrappers are

overlays or translators that call more complex functions beneath them allowing for

easier use and operability. Statistics packages like MATLAB give us this option and

are forgiving in that they are "high-level" packages in which a few keystrokes can

result in powerful results. Essentially, packages like MATLAB do a lot of the heavy

lifting for us and so have become very popular within science and finance. Cleve

Moler originally created MATLAB in the 1970's based on FORTAN and then

morphed into C later on. MATLAB stands for "MATRIX LABORATORY" and it is a

very powerful stats program that is optimized for large data sets and vectorization

of code. MATLAB is not a native language like C (though there is a MATLAB C

compiler that can transform "M-code" into native C), but it is still very fast –

unfortunately, it is not open source and is expensive (unlike packages like R). More

precisely, MATLAB is more like a functional language and is fourth generation type

in nature[9]. In terms of CUDA, we will take a slight hit to performance here since we

are one level removed from purity, so to speak. There is also less functionality

compared to the standard CUDA library. Here is what the M-code looks like in terms

of the main part of the original model program:

Exhibit 7 – A Complex MATLAB Program w/Wrappers: Trading Model *Author, 2013.*

```
%start clock timer (wall clock)
tic

%Check/set GPU device
g = gpuDevice(1);

%Number of optims - variable
Cut = 1810;

%Get market data in one shot from flat files and store in matrix
setdbprefs({'DataReturnFormat','ErrorHandling','NullNumberRead','NullNumberWrite','NullStringRead'
,'NullStringWrite','JDBCDataSourceFile'},{'numeric','empty','NaN','NaN','null','null',''});
conn = database('BLP32','','');
e = exec(conn,'SELECT ALL "pcrhc$"."pcr date","pcrhc$"."pcr price","spxhc$"."spx price" FROM
"pcrhc$","spxhc$" WHERE "pcrhc$"."pcr date" = "spxhc$"."spx date"');
e = fetch(e);
BLP = e.Data;
close(e)
close(conn)

%Get Z-scores from market data
Blotter = BLP;
Blotter(:,4:5) = single(zscore(BLP(:,2:3)));
Zpcr =single(zscore(Blotter(:,4)));
Zspx = single(zscore(Blotter(:,5)));
dates = Blotter(:,1);
lens = length(dates);

%Set parameters for z-score cutoff, moving-average length and rolling period - variable
assignin('base', 'zii', '1');
assignin('base', 'fii', '21');
assignin('base', 'pii', '377');

%start outer loop for optimizations
for fors=1:Cut;

%assign space in memory for CPU data and set initial values to zero for matrices - speed gains
ret =single(zeros(lens,1));
fp= single(zeros(lens,2));
fproll=single(zeros(lens,5));
```

Get z-scores by calling built-in function – no need to write code for generating this metric like in C – MATLAB has many such math functions and makes us lazy

---

[9] Feb. 2013 < http://en.wikipedia.org/wiki/Matlab >.

```matlab
ZpcrRoll=single(zeros(lens,2));
ZspxRoll=single(zeros(lens,2));
lookback=str2double(pii);
period =str2double(fii);
zcut=str2double(zii);
dates1=zeros(lens,2);
dates1f=zeros(lens,2);
dirf=zeros(lens,2);
dir=zeros(lens,2);
dirup=zeros(lens,1);
dirupf=zeros(lens,1);
dirdn=zeros(lens,1);
dirdnf=zeros(lens,1);

%copy data from CPU to GPU using WRAPPER function
Bl7   = gpuArray( single(Blotter(:,7)) );
Bl6   = gpuArray( single(Blotter(:,6) ));
Fpr   = gpuArray( single(fproll) );
Fpr2  = gpuArray( single(fproll) );
Fpr4  = gpuArray( single(fproll) );

%run main calculations on GPU – since arrays have been moved to GPU, function will run on GPU
for lp=1:lens-period+1;
    if Bl7(lp) > zcut && zcut > 0
            Fpr(lp)=Bl6(lp);
            for lp2=0:period-1;
                Fpr2(lp+lp2)=Bl6(lp+lp2);
            end;
            Fpr4(lp)=sum(Bl6(lp:lp+period-1));
    elseif Bl7(lp) < zcut && zcut < 0
             Fpr(lp)=Bl6(lp);
            for lp2=0:period-1;
                Fpr2(lp+lp2)=Bl6(lp+lp2);
            end;
            Fpr4(lp)=sum(Bl6(lp:lp+period-1));
    else Fpr(lp)=0;
    end;
end;

%Gather results on GPU and send back to CPU
f1=gather(Fpr);
f2=gather(Fpr2);
f4=gather(Fpr4);

%assign results to main data matrix
fproll(:,1)= (f1(:,1));
fproll(:,2)= (f2(:,1));
fproll(:,4)= (f4(:,1));

%specify and format output data for flat files
DATE = [datef];
PCR = [Blotter(:,2)];
SPX = [Blotter(:,3)];
PCRstaticZscore = [Blotter(:,4)];
PCRrollingZscore = [Blotter(:,7)];
StaticFwdRet = [Blotter(:,10)*100];
RollFwdRet = [Blotter(:,11)*100];
format bank;
Pres = dataset(DATE,PCR,SPX,PCRstaticZscore,PCRrollingZscore,StaticFwdRet,RollFwdRet);

%export fomratted data to .dat file in local directory
export(Pres,'file','PCRdata.dat');
end;
toc


%reset GPU device
reset(g);
```

Copy CPU data into GPU arrays via wrapper function "gpuArray" – switch to single or floating precision for speed gains – no asynchronous or overlapping memory transfers allowed like in CUDA C

Run core loops on GPU for positions and returns – do not need to specify GPU device here since we are looping through GPU arrays (previous block) and so operations automatically run on GPU

Note that advanced CUDA tools like multi-streaming are not available in MATLAB like in CUDA C – this is trade-off for ease of use!

Must collate results from GPU before transferring back to CPU via "gather" wrapper function

Finally, reset the GPU device at the end

Again, the performance hit will be notable running GPU operations in MATLAB versus CUDA C – it is roughly **2X** to **4X** slower in MATLAB. Also, note that open source statistics packages also exist such as R and SPYDER. Each package handles GPU functionality in its own way (if available). For PYTHON lovers, a great package is called SPYDER that comes with an extension called PYCUDA. This combination is the closest thing to MATLAB: https://code.google.com/p/spyderlib/ .

JCUDA

Because JAVA is still so popular and is often used in HPC open-source computing, it is an interesting challenge see how well it can execute the same complex test program by wrapping CUDA functions (hence "JCUDA"). Since we've already installed the drivers for earlier programs, we simply need to add the binary packages from the JCUDA website and place these in our JAVA PATH folder: http://www.jcuda.org/tutorial/TutorialIndex.html . If all goes well, we should then be able to compile the same program after changing some pointer references and see something like this:

```
Microsoft Windows [Version 6.1.7601]
Copyright (c) 2009 Microsoft Corporation.  All rights reserved.

C:\Users\Xtreme>cd C:\Program Files\Java\jre7\JCuda-All-0.6.5-bin-windows-x86_64


C:\Program Files\Java\jre7\JCuda-All-0.6.5-bin-windows-x86_64>dir
 Volume in drive C is SSD
 Volume Serial Number is C8F2-D949

 Directory of C:\Program Files\Java\jre7\JCuda-All-0.6.5-bin-windows-x86_64
```

```
10/26/2014  07:53 PM    <DIR>          .
10/26/2014  07:53 PM    <DIR>          ..
10/26/2014  12:20 PM           183,545 jcublas-0.6.5.jar
10/26/2014  12:20 PM           145,408 JCublas-windows-x86_64.dll
10/26/2014  12:20 PM           274,944 JCublas2-windows-x86_64.dll
10/26/2014  12:20 PM           978,273 jcuda-0.6.5.jar
10/26/2014  12:20 PM           128,512 JCudaDriver-windows-x86_64.dll
10/26/2014  12:20 PM           109,568 JCudaRuntime-windows-x86_64.dll
10/26/2014  01:16 PM               302 JCudaRuntimeTest.java
10/26/2014  12:20 PM            77,073 jcufft-0.6.5.jar
10/26/2014  12:20 PM            29,696 JCufft-windows-x86_64.dll
10/26/2014  12:20 PM            80,297 jcurand-0.6.5.jar
10/26/2014  12:20 PM            36,864 JCurand-windows-x86_64.dll
10/26/2014  12:20 PM           259,370 jcusparse-0.6.5.jar
10/26/2014  12:20 PM           487,424 JCusparse-windows-x86_64.dll
              13 File(s)      2,791,276 bytes
               2 Dir(s)  100,684,779,520 bytes free

C:\Program Files\Java\jre7\JCuda-All-0.6.5-bin-windows-x86_64>javac -cp ".;jcuda
-0.6.5.jar" JCudaRuntimeTest.java

C:\Program Files\Java\jre7\JCuda-All-0.6.5-bin-windows-x86_64>
```

[TODO]

Extreme GPU Computing: Master Level Complexity

Using Multiple GPU's (Server/Cluster) & Streams for Further Speed Gains in Model Program – Multi-GPU's & Multi-Streams (Plus Advanced Math Functions)

We've been building up in complexity in terms of GPU programs. Finally, we've arrived at the most complex iteration possible and it involved CUDA C (which should come as no surprise). Previously, we only ran one CUDA kernel at a time on one GPU device – recall that NVIDIA's PROFILER tool suggested that CONCURRENCY was non-existent (see page 39). So, even though we were multi-threading in earlier programs, **there is more to true parallelism than threading**. For maximum speed gains, we're now going to fix that as well as other adjustments (e.g., more advanced math functions plus multiple GPU devices).

When we have access to a GPU cluster or **server**, we can get further speed gains by re-writing our code (in LINUX typically since most servers run on this OS). Recall in the earlier example of the model program, we were only looping through **1** market (or security) at a time. Though the CUDA function was running in parallel by using multiple cores (for multiple threads) simultaneously to generate trade positions and returns, there is still a bit of a serial or sequential "feel" here in terms of process. But, if we have a large enough GPU (e.g., a TESLA GPU) or even multiple GPU's, we can re-write our code to process several markets or securities at once. For example, NVIDIA[10] has 2 TESLA (K20) GPU's on one of its server nodes. So, theoretically, we could at least process 2 markets simultaneously instead of stepping through **market-by-market**. Theoretically, this should further reduce computational time by a factor of 2, but it does not exactly work out this way due to additional overhead regarding the coordination of multiple GPU's.

Amdhal's Law

Let's remember **AMDHAL**'s law: **S = 1 / 1−P**. Now, if a program is further parallelized, the maximum speed-up over serial code is **1 / (1 − 0.50) = 2**. So, we should still see additional speed gains overall though not by a factor of 2 exactly. Here is what the main part of the code looks like in terms of multi-streaming and multiple GPU's – remember that we had 4 nested "**for-next**" loops earlier (see pages 34-37) – now we will need a total of **5** loops: LOOP BY GPU

---

[10] The numerical simulations needed for this work were performed on [NVIDIA Partner] Microway's Tesla GPU accelerated compute cluster.

Exhibit 8 – The Most Complex CUDA C Program: Trading Model Author, 2013.

```
//Change path below for UNIX "c://usr"
#define PATH "C:\\"
#define LOOKBACK 1597    // 1597-987-610-377-144-89 fibos rolling optimization historical period
#define STEP 377          // or 89 fibos step forward in time period for next rolling optimization
#define NUMI 3   //up to 27 number of markets - this is not FREE downloadable data from internet
#define STR 2    //number of streams - use 2 to be safe since most GPU's can handle 2 streams
#define THR 256 //initial threads - must specify this PARAM - can affect OCCUPANCY or %USE OF GPU

//INT MAIN//INSERT NEW STUFF HERE EACH TIME START SUB-FUNCTION*******MAIN AREA****//Declare each
new variable here - initializing and declaring space/memory for return arrays of variables or
output we want****STEP B//
int main(int argc, char **argv){
```
> MAIN program entry point
> Must specify initial number of kernel streams (GPU function launches) per GPU – 2 is a safe number

```
//CHECK for USER INPUT larger than MAX NUMI of Markets
       if (NUMI > 27)
   {
       fprintf(stderr, "You entered too many markets! MAX number is 27! Please try again!\n");
       exit(EXIT_FAILURE); }

       // Error code to check return values for CUDA calls
       cudaError_t err = cudaSuccess;
       err=cudaGetDeviceCount(&GPUn);

       if (err != cudaSuccess)
   {
       fprintf(stderr, "Failed to find GPU devices  (error code %s)!\n",
       cudaGetErrorString(err));
       exit(EXIT_FAILURE); }
```
> Must check for number of GPU's on server – use error-trapping here to exit program upon error

```
//LOOP thru as many GPU devices as there are available - so now we are MULTI-STREAMING and using
MULTI-GPU's - this is FULL POWER of GPU computing
for (dd = 0; dd < GPUn; dd++) {
```
> Outer-most for-next loop – looping thru by number of GPU's detected on server – 1 of 5 for-next loops

```
       clock_t ff, ss, t3; float diff=0.00f;
       ff=dd;
       ff = clock();
       strcpy(destr, PATH"recon");//output RECON directory for each GPU
       strcpy(foldr, ".dat");
       sprintf(fnumss, "%d", dd);
       strcat(destr,fnumss);
       strcat(destr,foldr);
       recon=f_openw(destr);

       err=cudaSetDevice(dd);
       if (err != cudaSuccess){
       fprintf(stderr, "Failed to find GPU device  (error code %s)!\n", cudaGetErrorString(err));
       exit(EXIT_FAILURE); }

       //divide market data array up into blocks of markets to run on EACH GPU
       gapn = NUMI/GPUn;
       gapo = NUMI/GPUn;
       startg = 1 + (dd*gapn);
```
> Divide market data up into blocks to run on each GPU – check for odd number of markets

```
       //check for odd number of markets divided by GPUs
       if (dd == GPUn-1 && NUMI % GPUn != 0) gapn = ((int)((NUMI % GPUn)*GPUn)+gapo)-1;
       if (gapn == 0) gapn = 1;
```

```
for (gg = 1; gg <= gapn; gg++) {          // top loop for number of market data files passed thru
dfiles[] // must change NUMI in #def as add number of markets

        sprintf(sources,dfiles[(gg-1)+(dd*gapo)]);     //find and open price data files to get
        lengths for periodicities//

        fins=f_openr(sources);
        endf=f_line(fins);
        endf--;
        peri[gg] = (int)(((endf-LOOKBACK)/STEP)+1); //number of rolling periods in each data set
        for rolling optimization (aka moving average)//

        f_close(sources,fins);
        sprintf(fnums, "%d", gg);
        strcpy(desta, PATH"OSrunALL");              //output directory for out-of-sample tests for
        all combined tests per market

        strcpy(foldr, ".dat");
        strcat(desta,"-");
        strcat(desta,fnums);
        strcat(desta,"-");
        strcat(desta,marks[(gg-1)+(dd*gapo)]);
        strcat(desta,foldr);
        ferri=f_openw(desta);

 for (ii = 1; ii <= peri[gg] ; ii++) {   // loop is for periodicity - so 30yrs of price data
divided into sub-units for rolling optimization (aka parameter sweeps)

   for (z = 0; z < lensa; z++) {          // 2 nested for loops for parameter sweep or combination
of arrays a[] and b[]//

    for (j = 0; j < lensb; j++) {

        lens = (int)(b[j]);

//** COPY CUDA VARIABLES FROM CPU (HOST) TO GPU (DEVICE) - USE ASYNC TRANSFER FOR MORE SPEED SO
CPU DOES NOT HAVE TO WAIT FOR GPU TO FINISH OPERATION AND CAN PROCEED FURTHER IN THE MAIN
PROGRAM**
        cudaMemcpyAsync(zscores_d, zscores, (int)(end)*sizeof(float), cudaMemcpyHostToDevice,0);
        cudaMemcpyAsync(rets_d, rets, (int)(end)*sizeof(float), cudaMemcpyHostToDevice,0);

        gap = (stop - start)/STR;
        lenny=stop-start;
        dim3 threads; threads.x = THR;   //use threads as per specific GPU device for higher
        OCCUPANCY/USE OF CARD - trial-and-error via PROFILING
        dim3 blocks; blocks.x = (lenny/threads.x) + 1;  //max blocks is 112 on GTX 670 device

        // allocate and initialize an array of stream handles
        cudaStream_t *streams = (cudaStream_t *) malloc(STR * sizeof(cudaStream_t));

        //** Create Streams for Concurrency or Multi-Streaming - now we will call several KERNELS
        simultaneously**
        for(int i = 0; i < STR; i++) cudaStreamCreate(&(streams[i]));

        //** CALL GPU FUNCTION/KERNEL HERE FOR MODEL PARAMETER SWEEP TO GENERATE IN_SAMPLE
        RESULTS**THIS IS THREAD REDUCTION DUE TO CONCURRENCY!
        for (i = 0; i < STR; i++){
        kernelSim<<<32,threads,0,streams[i]>>>(zscores_d,rets_d,pnl_d,pos_d,start+(i*gap),start+((
        i+1)*gap),(float)(a[z]),lens);

        if (i == STR-1)
        kernelSim<<<32,threads,0,streams[i]>>>(zscores_d,rets_d,pnl_d,pos_d,start+(i*gap),stop,(fl
        oat)(a[z]),lens);}

        //sync streams before copying back to CPU
        cudaStreamSynchronize(streams[STR-1]);
```

2 of 5 for-next loops by number of markets on each GPU

3 of 5 for-next loops by rolling time period of each market (for period optimization)

4th & 5th for-next loops by model parameters 1 & 2 (for period optimization)

Allocate & create N kernel simultaneous streams for each GPU for concurrency

Use FMA fast math GPU function in GPU kernel

Call main CUDA function kernel by looping thru N kernels created up above – note streams[i] here

Sync up concurrent kernel streams before copying data from GPU back to CPU

```
//** COPY CUDA VARIABLES/RESULTS FROM GPU (DEVICE) BACK TO CPU (HOST) - MUST WAIT FOR GPU
OPERATION/FUNCTION TO FINISH HERE SINCE LOW ASYNC/CONCURRENCY ON NON_TESLA GPU DEVICES**
cudaMemcpy(pos, pos_d, (int)(end)*sizeof(float)/*stop-start*/, cudaMemcpyDeviceToHost);
cudaMemcpy(pnl, pnl_d, (int)(end)*sizeof(float), cudaMemcpyDeviceToHost);

        //** Destroy Streams for Concurrency or Multi-Streaming - now we will RELEASE
resources back to GPU**
for(int i = 0; i < STR; i++) cudaStreamDestroy(streams[i]);
```

Destroy concurrent
kernel streams at end

So, how did we do? We now have the **fastest** computational times yet: 1810

optimization in 65 seconds (or **0.036** seconds per optimization). All that work

finally paid off since we were at 87 seconds before running on a single GPU without

multiple streams – so, roughly a +33% pickup in speed. Here is a fun picture of a

massive server room at a local business (name withheld):

Conclusion and Wrap-Up: Did we prove out speed gains?

We can see that GPU computing is a powerful tool just by looking at the following **performance grid** where we pick up speed gains of ~ **300% to 700%** depending on the benchmark. Some of these gains are due to the C language, itself, in spite of the GPU. It is interesting to note that the same model program ran considerably slower in C# - remember that C# is OOP and does not have **pointers**-to-arrays like in C. Again, C makes itself easily available to vectorization of math and memory operations. Remember also that data structures like arrays operate in linear time (**O(n)**) if not constant time (**O(1)**) for other operations. Other data structures like sorted lists and trees can hit log time (**O(log n)**) under certain circumstances.

However, most of these gains are indeed due to GPU operations in CUDA C where we see a +300% pickup in speed versus the same model program in standalone C. Again, the **fastest** program utilized multiple GPU's and kernel streams as well as tips like pinned memory, asynchronous memory transfers and floating operations. This study slowly built-up the complexity of the CUDA C approach as follows: SINGLE KERNEL ON GPU → MULTIPLE KERNELS ON MULTIPLE GPUs. Open source libraries like OpenACC also did well while wrapper programs like MATLAB fared worse. Lastly, this survey has demonstrated that GPU computing need not be all that complicated – truly, we don't have to be geniuses to take advantage of hybrid HPC and younger generations are encouraged to get involved, especially, as the world moves from finance back to "hard" science.

[TODO]

Exhibit 9 – Trading Model Performance Results Across Platforms *Author, 2013.*

Test	Language/Platform	GPU/CPU	#Combos/Optims	#Markets	#Years (Avg Length of test)	Total Time (sec)	Avg Time per Optim(sec)
1-f Model	C# - Visual Studio 2010 - PC	CPU	89	1	2	23	0.26
1-f Model	M-CODE + CUDA WRAPPER - MATLAB 2012 STATS PACKAGE - PC w/Single GPU	CPU & GPU	89	1	2	13	0.15
1-f Model	C - Visual Studio 2008 - PC	CPU	89	1	2	7	0.08
1-f Model	OpenACC C - Visual Studio 2008 (Open Source Non-Nvidia Library) - PC w/Single GPU	CPU & GPU	89	1	2	5	0.06
1-f Model	CUDA C - Visual Studio 2008 - PC w/Single GPU	CPU & GPU	89	1	2	4	0.05
1-f Model	M-CODE + CUDA WRAPPER - MATLAB 2012 STATS PACKAGE - PC w/Single GPU	CPU & GPU	1810	27	2	257	0.14
1-f Model	C - Visual Studio 2008 - PC	CPU	1810	27	2	213	0.12
1-f Model	OpenACC C - Visual Studio 2008 (Open Source Non-Nvidia Library) - PC w/Single GPU	CPU & GPU	1810	27	2	108	0.06
1-f Model	CUDA C - Visual Studio 2008 - PC w/Single GPU	CPU & GPU	1810	27	2	99	0.05
1-f Model	CUDA C - Eclipse on Linux - PC w/Single GPU	CPU & GPU	1810	27	2	87	0.05
1-f Model	CUDA C - NVIDIA LINUX GPU SERVER (KEPLER) - SINGLE-GPU On CLUSTER w/XEON CPU SERVER	CPU & GPU	1810	27	2	86	0.05
1-f Model	CUDA C - NVIDIA LINUX GPU SERVER (KEPLER) - MULTI-GPU STREAMS/SERVER w/XEON CPU SERVER	CPU & GPU	1810	27	2	65	0.036

PC Test Bed:
Intel DZ77GA-70K Motherboard
Intel Core i7-3770KCPU @ 4.2GHz OC'd
Corsair 32G RAM DDR3 @ 1600MHz OC'd
Corsair 240G Force 3 SSD SATA III @ 6G/s
Nvidia GTX 670 (GK104 w/1344 cores w/PCIe 3.0) GPU @ 1050MHz OC'd
64-Bit OS (Windows 7 or Ubuntu 11.10 (for Linux))

SERVER Test Bed:
Nvidia TESLA K20 (KEPLER) NODE - 2 GPU's**
Intel XEON SERVER- 8 CPU's

*The numerical simulations needed for this work were performed on [NVIDIA Partner] Microway's Tesla GPU accelerated compute cluster***

Here is a fun chart of other organizations using GPU computing to speed up work:
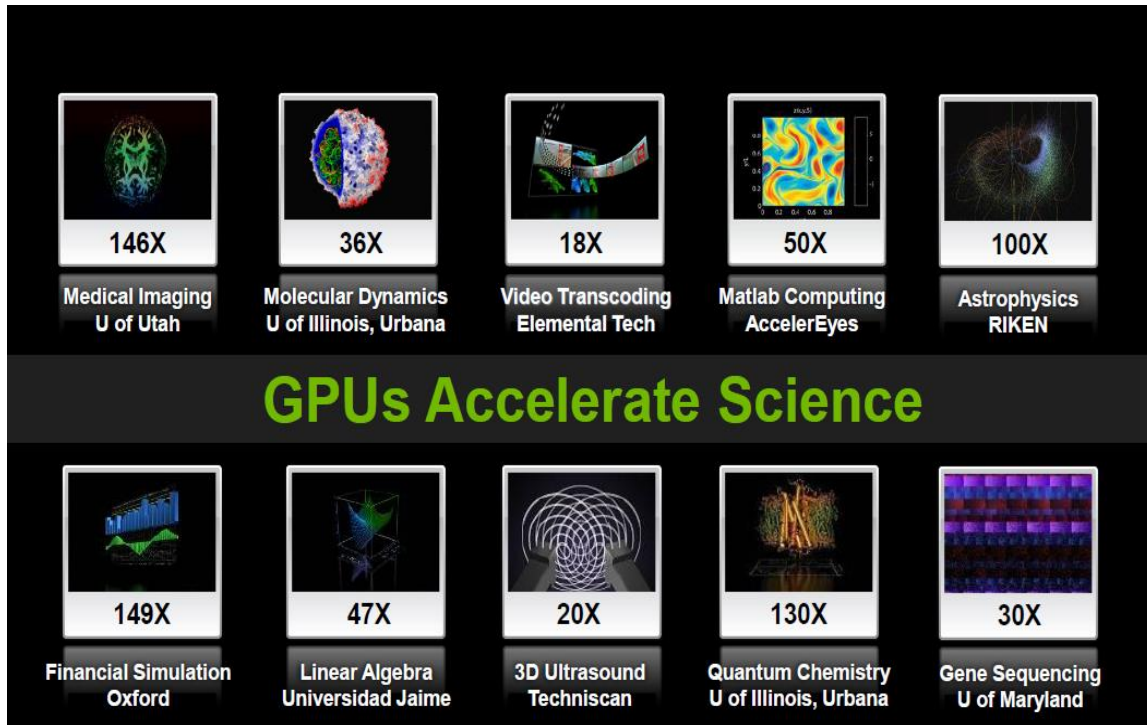

Figure 19 – Organizations Using GPU Accelerators Nvidia, 2012: 45.

Common Pitfalls of CUDA Programming: Precision & Promotion

When using casting or other techniques to convert variables and constants from DOUBLE to FLOAT (for further speed gains), we have to be very careful. This is because many languages like C have been formulated to "think" in DOUBLE precision (to optimize the CPU). So, if there is a statement like "constant = 1.0", this will be interpreted as a DOUBLE. The problem comes when we start mixing such a value with other values that are FLOATS. Remember that FLOATS have up to 8 significant digits while DOUBLE has 16. If we carelessly mix these 2 types, C will start to make transformations/conversions to various values without us knowing about it so that we might not get back the results we intended. So, it is always better to use FLOAT LITERALS with other FLOAT values to keep things straight - "constant = 1.0f" is better here if we indeed are using various values in a GPU function that is intended for FLOAT types. There is a nice blurb here from Nvidia warning about this[11]:

When comparing the results of computations of float variables between the host and device, make sure that promotions to double precision on the host do not account for different numerical results. For example, if the code segment:
float a;

...

a = a*1.02;

*were performed on a device of compute capability 1.2 or less, or on a device with compute capability 1.3 but compiled without enabling double precision (as mentioned above), then the multiplication would be performed in single precision. However, if the code were performed on the host, the literal 1.02 would be interpreted as a double precision quantity and a would be promoted to a double, the multiplication would be performed in double precision, and the result would be truncated to a float—thereby yielding a slightly different result. If, however, the literal **1.02** were replaced with **1.02f**, the result would be the same in all cases because no promotion to doubles would occur. To ensure that computations use single-precision arithmetic, always use float literals. In addition to accuracy, the conversion between doubles and floats (and vice versa) has a detrimental effect on performance.*

---

[11] Nvidia, "CUDA C Guide and Best Practices," January 2013, 12.

Bibliography

Cochrane, John, *"Time Series for Macroeconomics and Finance,"* University of Chicago, January 2005.

Coulouris, George, "*Distributed Systems and Concepts, Fifth Edition*," Addison-Wesley, 2012.

Fuzzy set. (n.d.). Retrieved February 2013, from Wiki:
http://en.wikipedia.org/wiki/Fuzzy_set

Gleick, James, "*The Information: A History, a Theory, a Flood*," Pantheon, 2011.

Hargitta, Istran. *Martians of Science.* New York, NY: Oxford Press, 2006.

Kochan, Stephen, "*Programming in C, Third Edition*," Sams Publishing, July 2005.

Matlab. (n.d.). Retrieved February 2013, from Wiki:
http://en.wikipedia.org/wiki/Matlab

Nvidia, "*CUDA C Guide and Best Practices*," Nvidia, 2013.

Nvidia Cuda. (n.d.). Retrieved February 2013, from Wiki:
http://en.wikipedia.org/wiki/CUDA

Thiruvathukal, George, "*High-Performance Java Platform Computing*," Pearson Education, 2000.