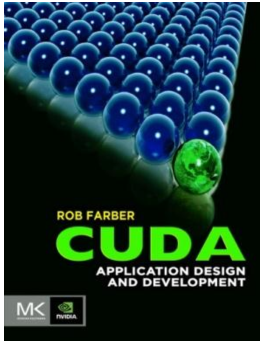
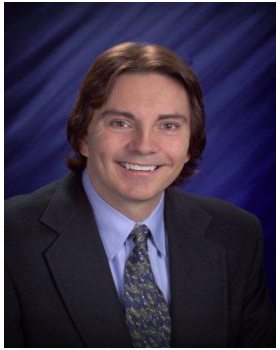
Designing Killer CUDA Applications for X86, multiGPU, and CPU+GPU

Rob Farber

Chief Scientist, BlackDog Endeavors, LLC

Author, “CUDA Application Design and Development”





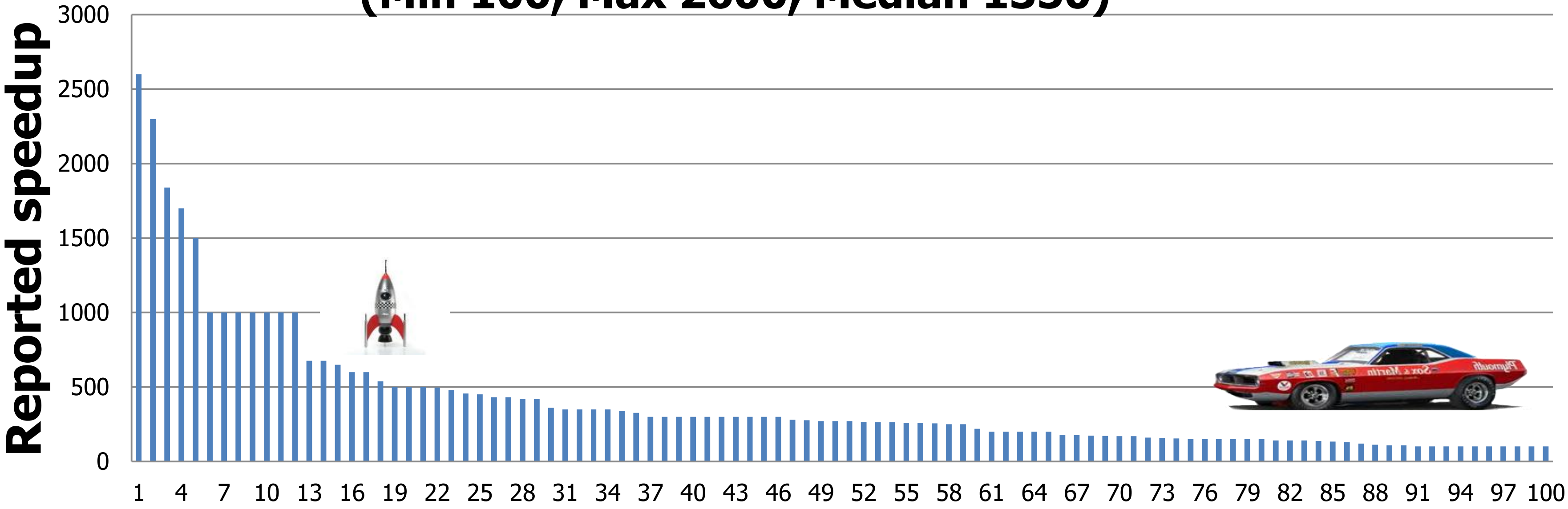
Doctor Dobb’s Journal CUDA tutorials

OpenCL “The Code Project” tutorials

Columnist

Performance is the reason for GPUs

Top 100 NVIDIA CUDA application showcase speedups as of July, 2011
(Min 100, Max 2600, Median 1350)



Ranked from highest to lowest speedup

http://developer.nvidia.com/cuda-action-research-apps

Supercomputing for the masses!

- Market forces evolved GPUs into massively parallel GPGPUs (General Purpose GPUs).
- **300+ million CUDA-enabled GPUs says it all!**
- CUDA: put supercomputing in the hands of the masses
  - December 1996, ASCI Red the first teraflop supercomputer
  - Today: kids buy GPUs with flop rates comparable to systems available to scientists with supercomputer access in the mid to late 1990s
    - GTX 560 $169 on newegg.com

Remember that Finnish kid who wrote some software to understand operating systems? Inexpensive commodity hardware enables:

- New thinking
- A large educated base of developers

<u>You can change the world!</u>

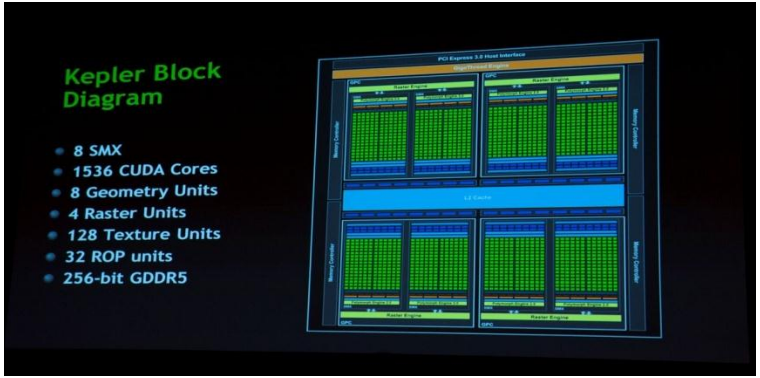
CUDA + GPUs are a game changer!

- CUDA enables orders of magnitude faster apps:
  - **10x** can make computational workflows more interactive (even *poorly* performing GPU apps are useful).
  - **100x** is disruptive and has the potential to fundamentally affect scientific research by removing time-to-discovery barriers.
  - **1000x** and greater achieved through the use of the NVIDIA SFU (Special Function Units) or multiple GPUs … Whooo Hoooo!

In this talk:

1. Two big ideas: SIMD, a strong scaling execution model
   - *A quick 12 slide trajectory from "Hello World" to approximately 400 teraflops of performance*
2. Another big idea: tying data to computation: multi-GPU and scalable workflows
3. Demonstrate simple real-time video processing on a mobile platform (an NIDIA GPU in a laptop)
   - *Example code is a foundation for augmented reality, smart sensors, and teaching*

Big idea 1: SIMD



<u>High-performance from the past</u>

- Space and power efficient
- Long life via a simple model

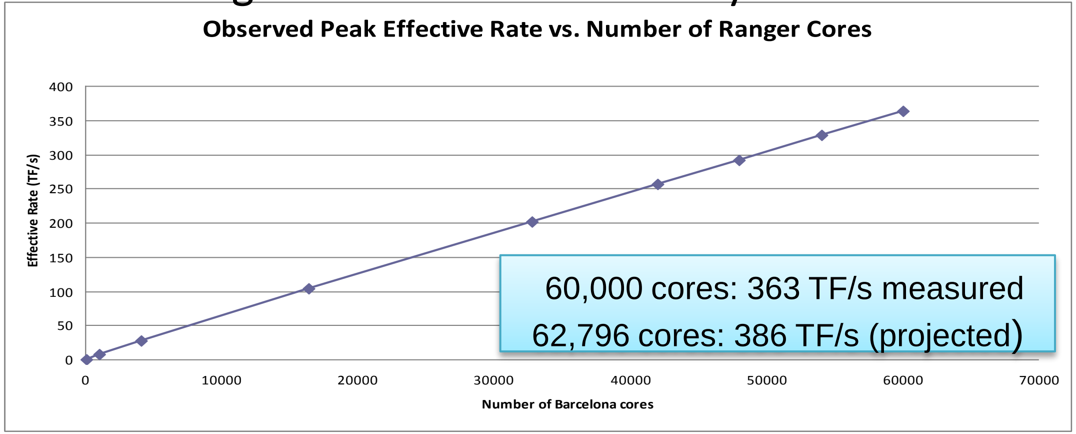
The Connection Machine

Farber: general SIMD mapping :

"Most efficient implementation to date"

(Singer 1990), (Thearling 1995)

Works great on multi-core MPI systems!

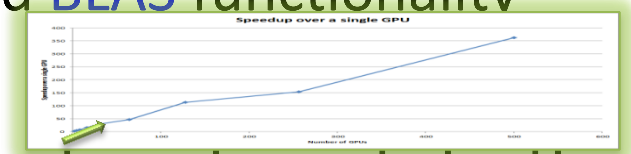


Results presented at SC09 (courtesy TACC)

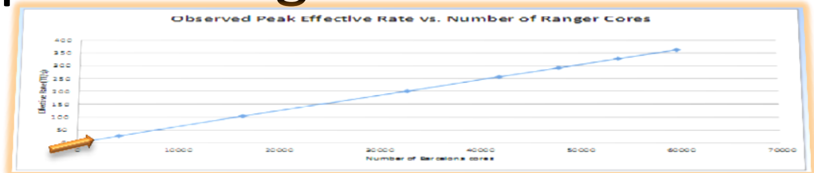
Big idea 2

The CUDA strong scaling execution model!

- Four basic types of programming models:
  - **Language platforms based on a strong-scaling execution model (CUDA and OpenCL™)**
  - Directive-based programming like OpenMP and OpenACC
  - Common libraries providing FFT and BLAS functionality
  - MPI (Message Passing Interface)

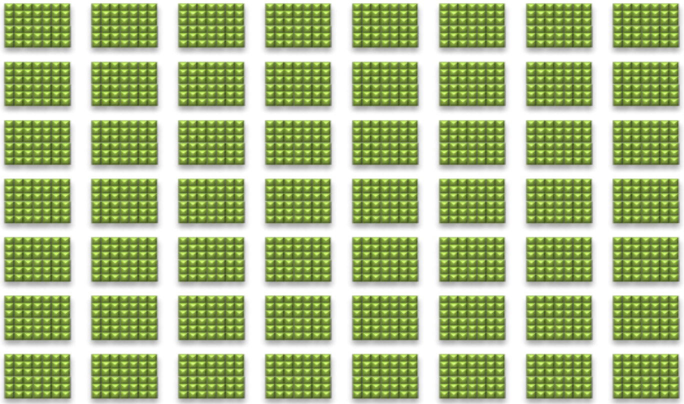


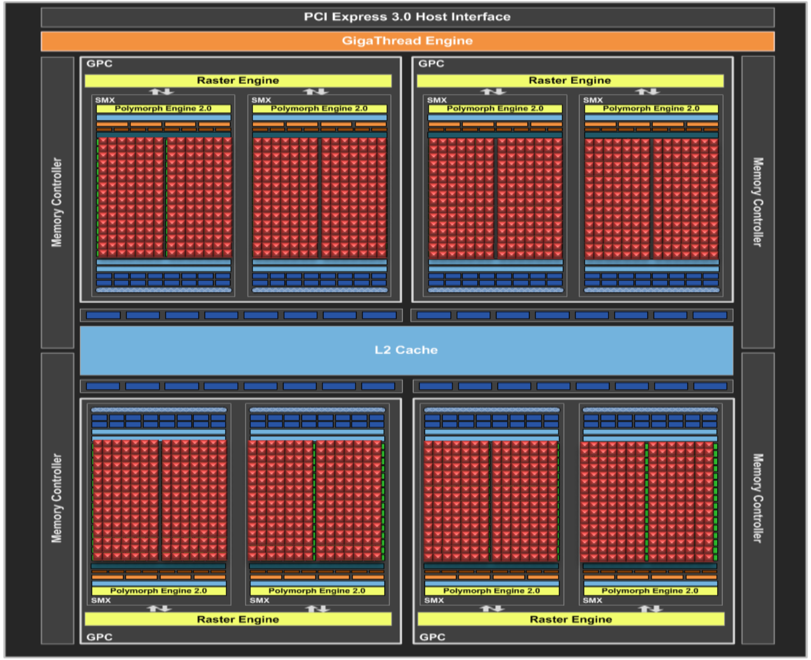
- Perfect strong scaling decreases runtime linearly by the number of processing elements



Scalability required to use all those cores (strong scaling execution model)

- Threads can only communicate within a thread block
  - (yes, there are atomic ops)
- Fast hardware scheduling
  - Both Grid and on SM/SMX



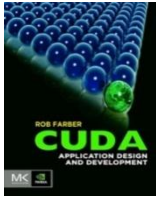


If you know C++, you are already programming GPUs!

```
//seqSerial.cpp
#include <iostream>
#include <vector>
using namespace std;

int main()
{
  const int N=50000;

  // task 1: create the array
  vector<int> a(N);

  // task 2: fill the array
  for(int i=0; i < N; i++) a[i]=i;

  // task 3: calculate the sum of the array
  int sumA=0;
  for(int i=0; i < N; i++) sumA += a[i];

  // task 4: calculate the sum of 0 .. N-1
  int sumCheck=0;
  for(int i=0; i < N; i++) sumCheck += i;

  // task 5: check the results agree
  if(sumA == sumCheck) cout << "Test Succeeded!" << endl;
  else {cerr << "Test FAILED!" << endl; return(1);}

  return(0);
}
```

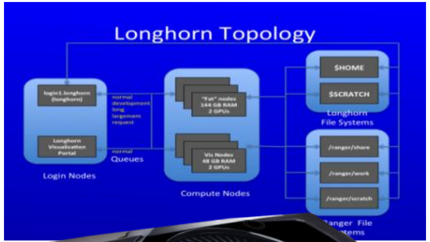
```
//seqCuda.cu
#include <iostream>
using namespace std;

#include <thrust/reduce.h>
#include <thrust/sequence.h>
#include <thrust/host_vector.h>
#include <thrust/device_vector.h>

int main()
{
  const int N=50000;

  // task 1: create the array
  thrust::device_vector<int> a(N);

  // task 2: fill the array
  thrust::sequence(a.begin(), a.end(), 0);

  // task 3: calculate the sum of the array
  int sumA= thrust::reduce(a.begin(),a.end(), 0);

  // task 4: calculate the sum of 0 .. N-1
  int sumCheck=0;
  for(int i=0; i < N; i++) sumCheck += i;

  // task 5: check the results agree
  if(sumA == sumCheck) cout << "Test Succeeded!" << endl;
  else { cerr << "Test FAILED!" << endl; return(1);}

  return(0);
}
```

First two examples in



Congrats on your first CUDA program!

- *Thrust::transform_reduce()*
  - Uses a functor to operate on (transform) data
  - Applies the reduction

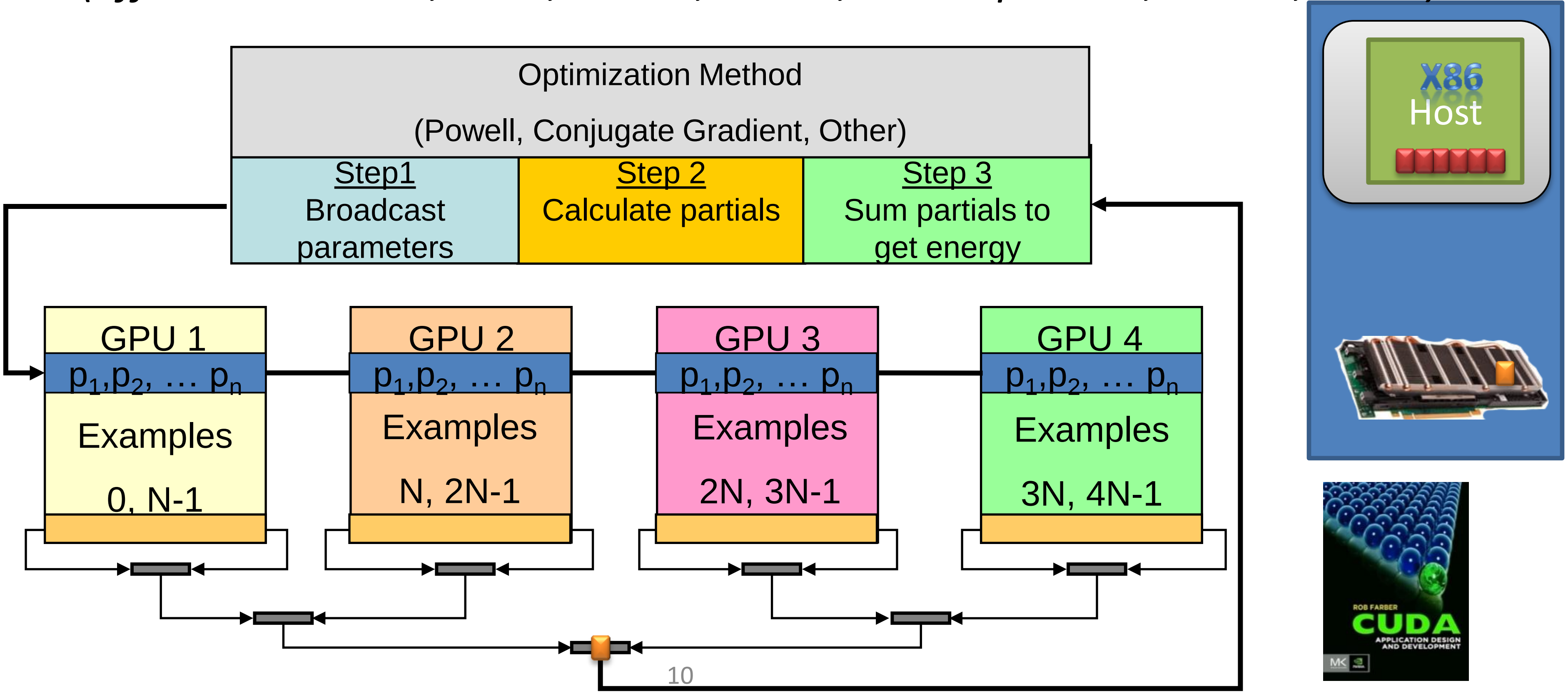


Surprise, you are now petascale to exascale capable!

A general mapping: use thrust::transform_reduce()

energy = objFunc(p_1, p_2, … p_n)

(efficient on SIMD, SIMT, MIMD, vector, vector parallel, cluster, cloud)

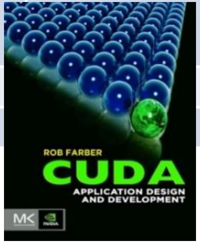


Speedup over a quad core when learning XOR

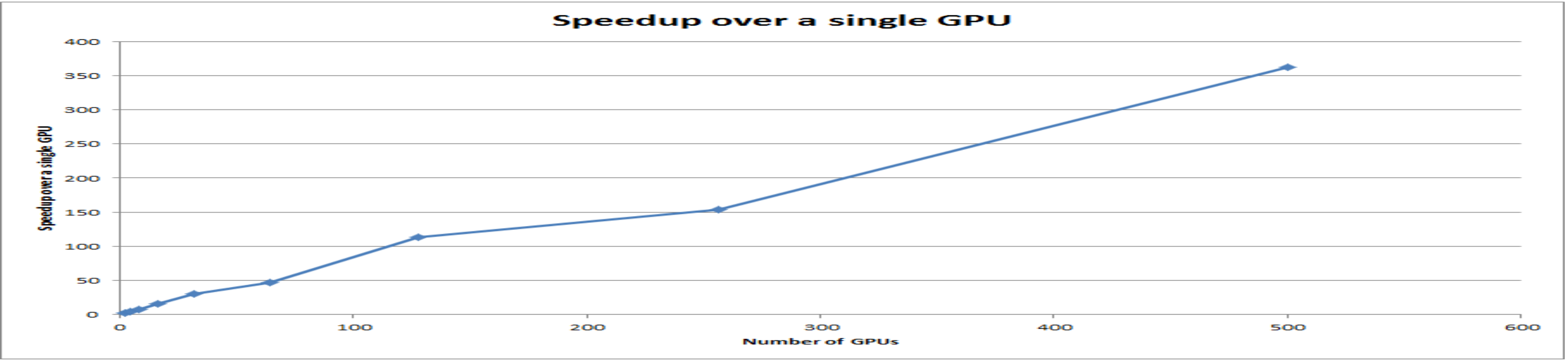
OS	Machine	Opt method	Precision	Ave obj func time	% func time	Speedup over quad-core	Speedup over single-core
Linux	NVIDIA C2070	Nelder-Mead	32	0.00532	100.0	85	341
Win7	NVIDIA C2070	Nelder-Mead	32	0.00566	100.0	81	323
Linux	NVIDIA GTX280	Nelder-Mead	32	0.01109	99.2	41	163
Linux	NVIDIA C2070	Nelder-Mead	64	0.01364	100.0	40	158
Win7	NVIDIA C2070	Nelder-Mead	64	0.01612	100.0	22	87
Linux	NVIDIA C2070	Levenberg-Marquardt	32	0.04313	2.7	10	38
Linux	NVIDIA C2070	Levenberg-Marquardt	64	0.08480	4.4	6	23
Linux	Intel e5630	Levenberg-M					
Linux	Intel e5630	Levenberg-M					
Linux	Intel e5630	Nelder-M					
Linux	Intel e5630	Nelder-M					

```
#pragma omp parallel for reduction(+ : sum)
  for(int i=0; i < nExamples; ++i)
  {
    Real d = getError(i);
    sum += d;
  }
```

Code for CPU generated by thrust

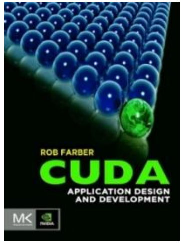


So simple it's the MPI example in Chapter 10



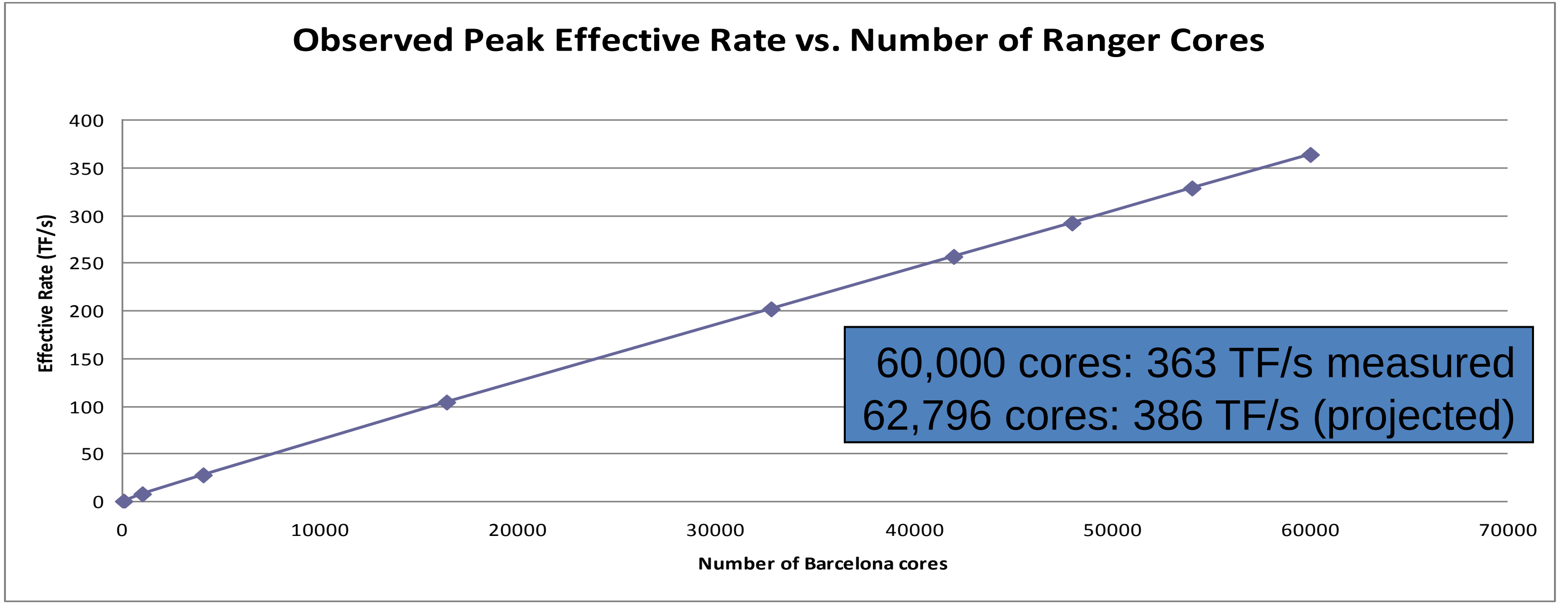
- Dominant runtime of code that scales to 500 GPUs

```
FcnOfInterest objFcn(input);

energy = thrust::transform_reduce(
            thrust::counting_iterator<int>(0),
            thrust::counting_iterator<int>(nExamples),
                                   objFcn, 0.0f, thrust::plus<Real>());
```



Exascale capable!

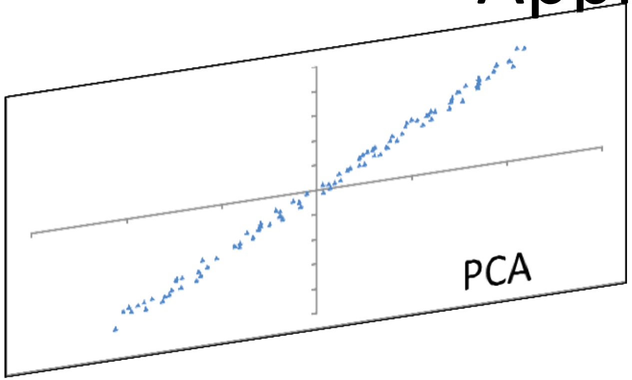
- Over 350TF/s of performance on Longhorn **(including communications!)**
- **Anybody willing to purchase 60,000 GPUs?** ☺



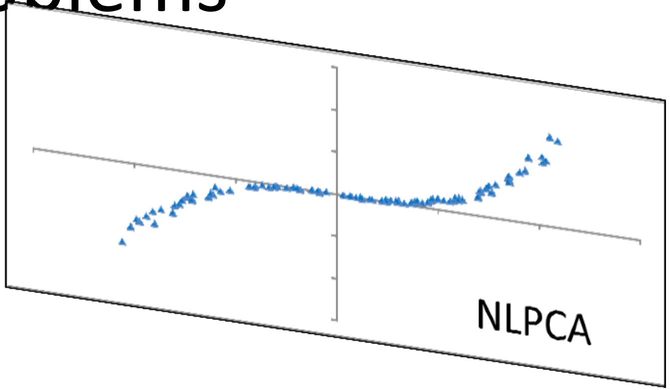
Results presented at SC09 (courtesy TACC)

From “first program” to petaflop capability in 7 slides!

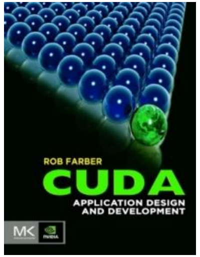
Applicable to real problems



- Locally Weighted Linear Regression
- Neural Networks
- Naive Bayes (NB)
- Gaussian Discriminative Analysis (GDA)
- k-means
- Logistic Regression (LR)
- Independent Component Analysis (ICA)
- Expectation Maximization (EM)
- Support Vector Machine (SVM)
- Others: (MDS, Ordinal MDS, etcetera)

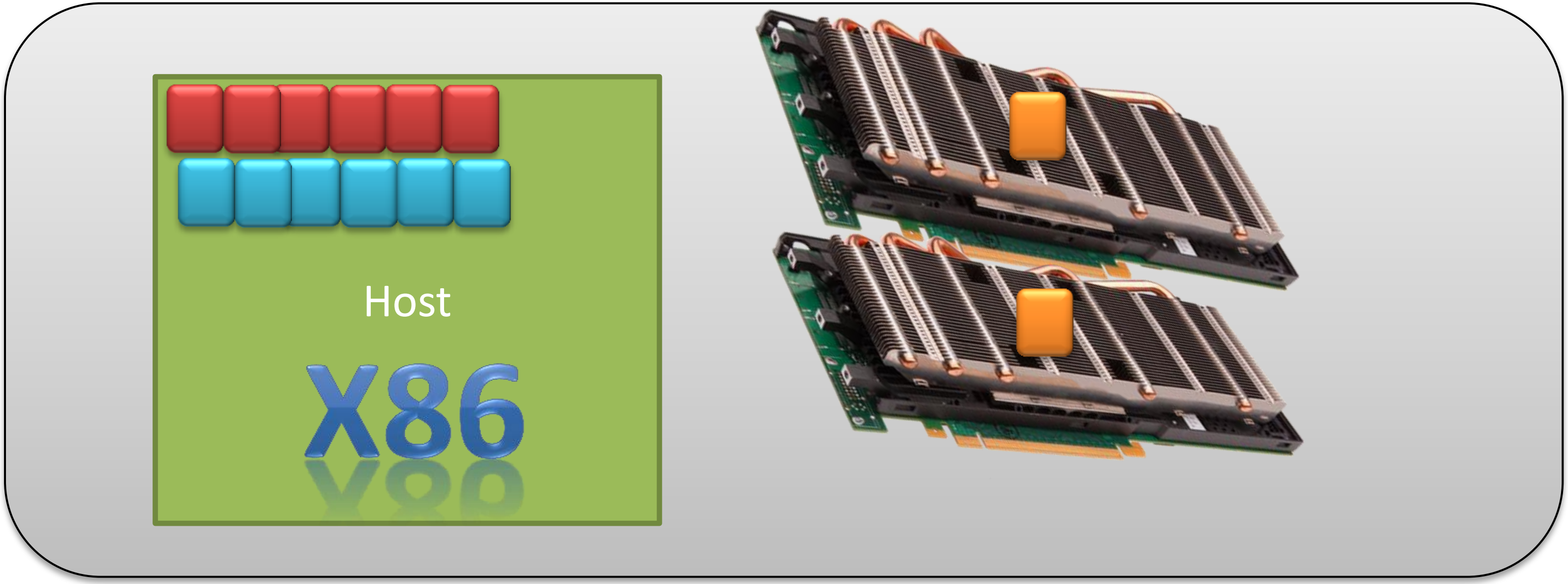


The book provides working code



CUDA 4.x makes multi-GPU much easier!

In-parallel, utilize GPUs and x86 capabilities!

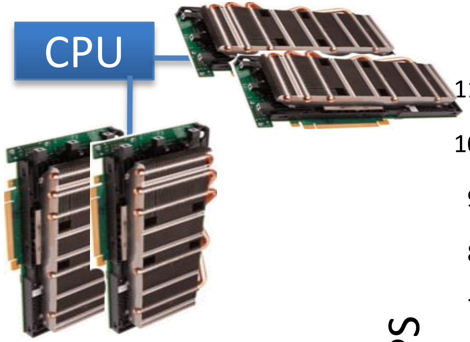


ICHEC
Irish Centre for High-End Computing

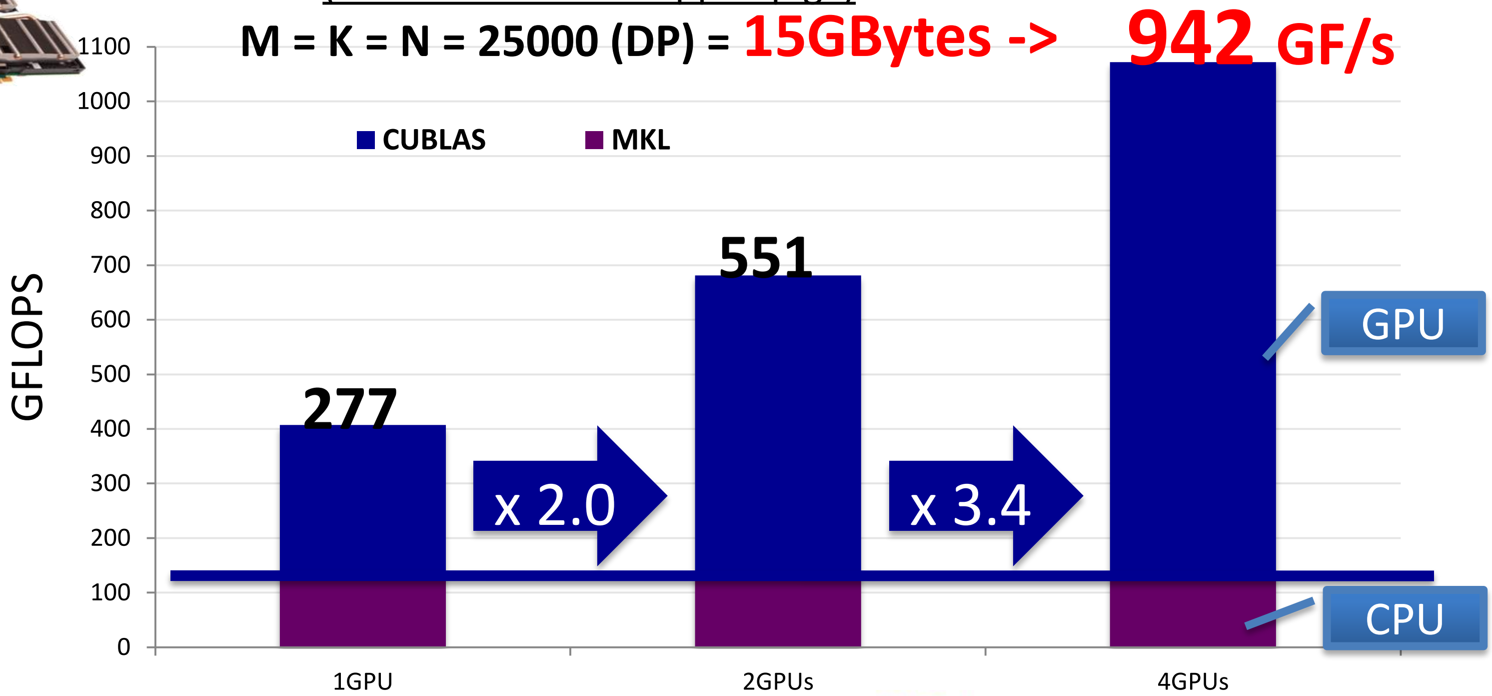
MultiGPU DGEMM
(matrix multiply)

http://qe-forge.org/projects/phigemm/
(Ivan Girotto and Filippo Spiga)

(2 GPUs per PCI Bus!!)



2 x Intel Xeon X5670 2.93GHz + 4 NVIDIA Tesla C2050

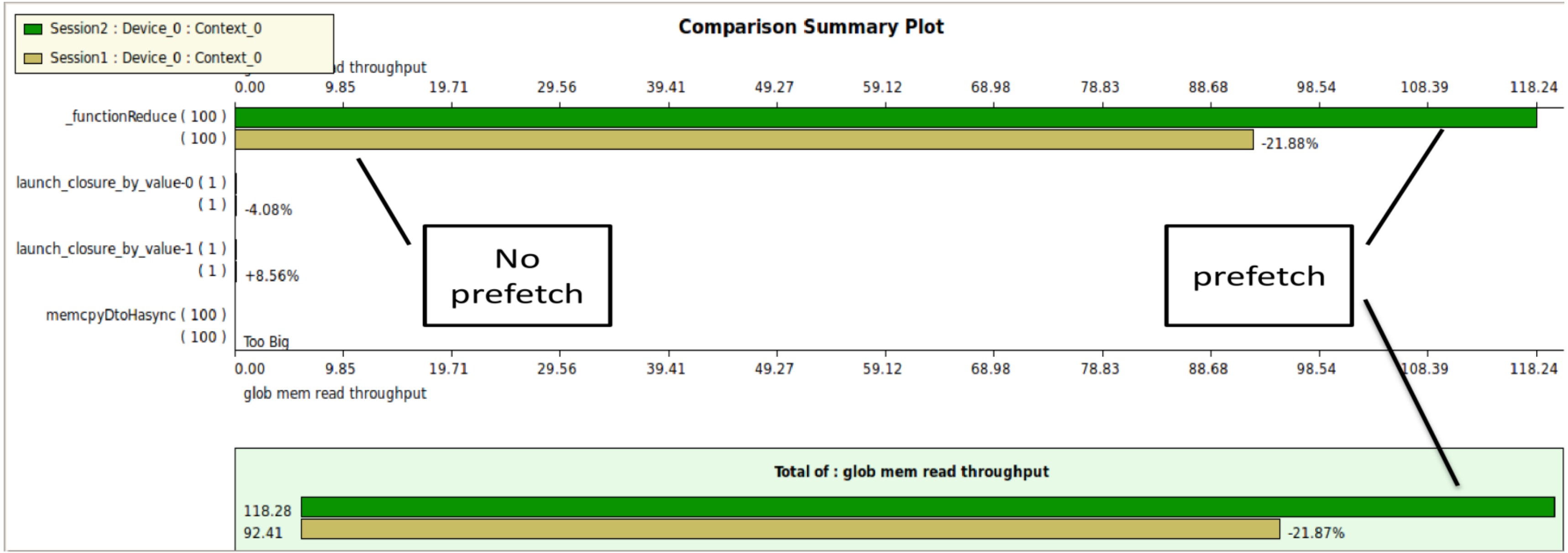


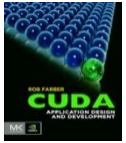
System provided by NVIDIA

Notice the size >> a single GPU

Use “PTX prefetch” to increase the effective memory bandwidth

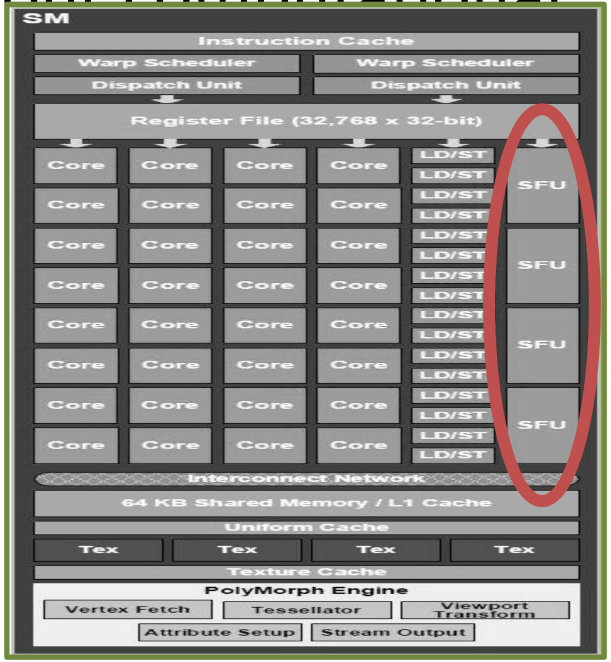
- asm volatile ("prefetch.global.L2 [%0];"::"l"(pt));
- Use prefetch in a vector reduction:





Love those SFUs! *(Special Function Units)*

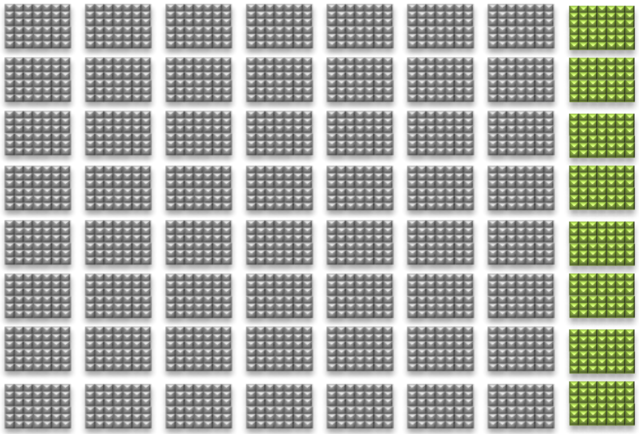
- Fast transcendental functions
  - The world is nonlinear … so are our computational models
  - Estimated 25x faster than x86

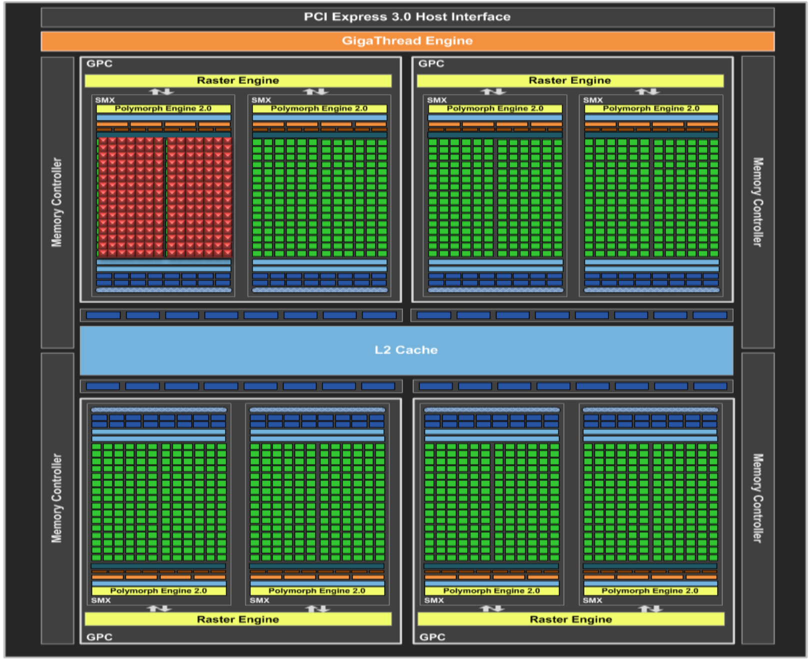


TLP (Thread Level Parallelism)

Bet that at least one thread will always be ready to run

– The more threads used, the better the odds are that high application performance will be achieved

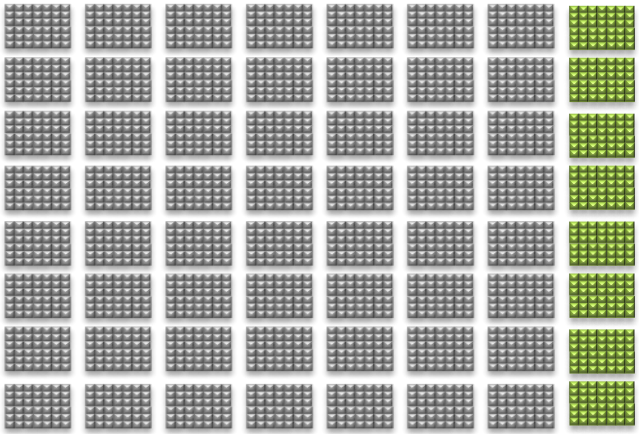


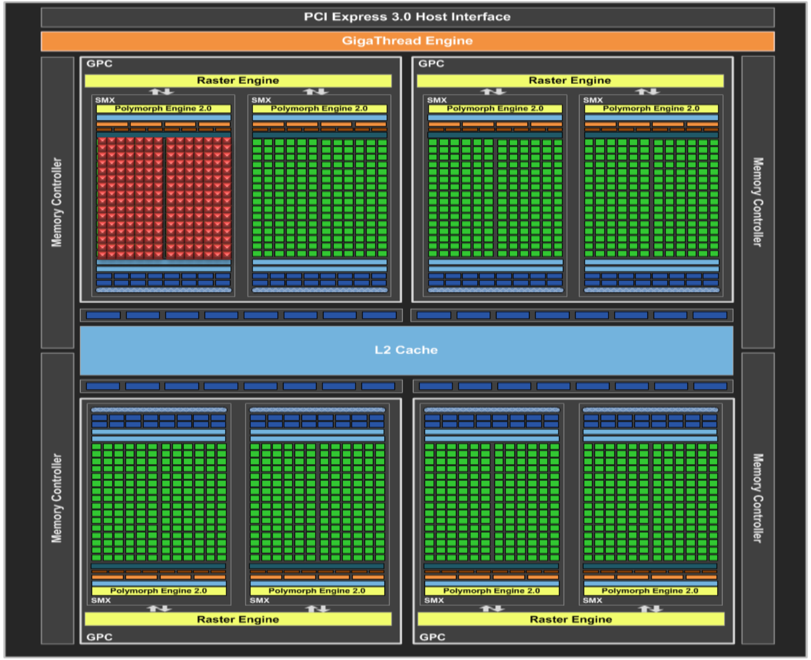


ILP (Instruction Level Parallelism)

Choreograph the flow of instructions for best parallelism

- Vasily Volkov has done some nice work in this area





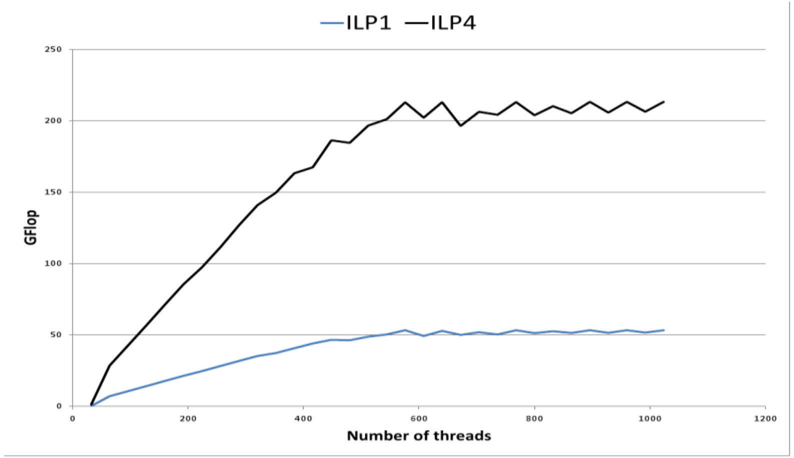
Use ILP to increase arithmetic performance

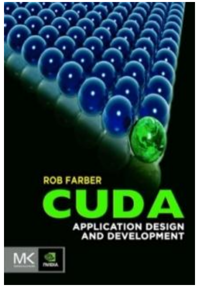
TLP

Thread 1	Thread 2	Thread 3	Thread 4
x = x + c	y = y + c	z = z + c	w = w + c
x = x + b	y = y + b	z = z + b	w = w + b
x = x + a	y = y + a	z = z + a	w = w + a

ILP

	Thread	
Instructions ->	w = w + b	Four independent operations
	z = z + b	
	y = y + b	
	x = x + b	
	w = w + a	Four independent operations
	z = z + a	
	y = y + a	
	x = x + a	





Kepler SMX with ILP

- Superscalar warp schedulers
  - Can transparently exploit some ILP for the programmer

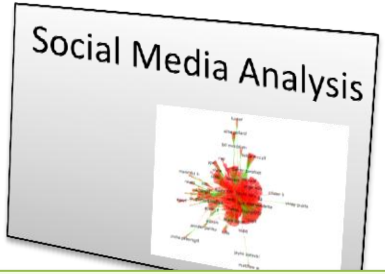
GTX 680

(Psst! GF104 like the Fermi GTX 460 has superscalar)

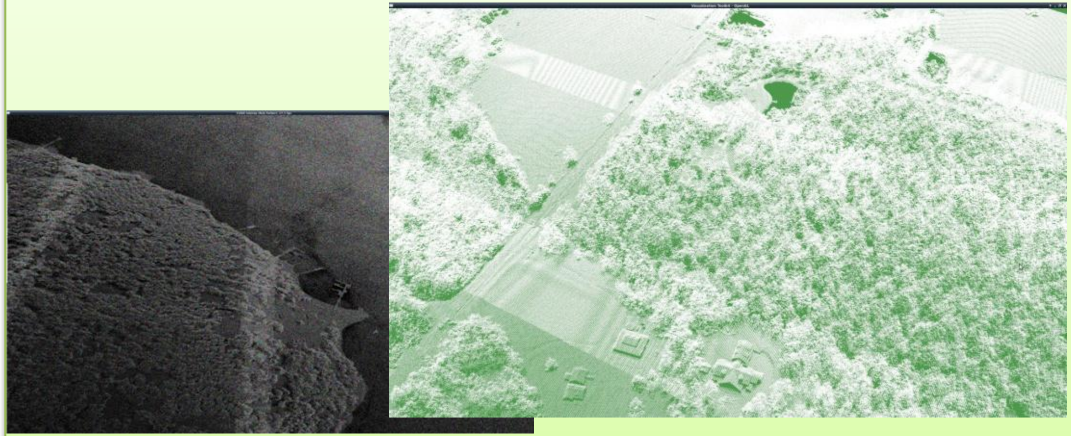
CUDA + Primitive Restart (a potent combination!)

Primitive restart: ***Looking forward to Kepler!***

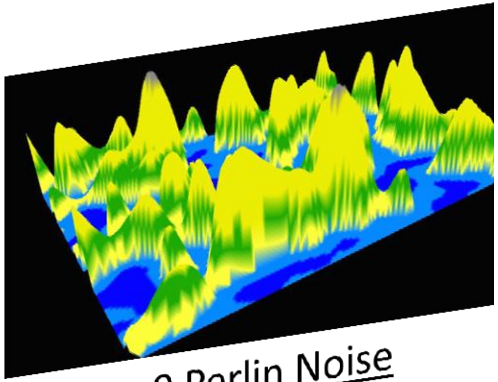
- A feature of OpenGL 3.1
- Roughly 60x faster than optimized OpenGL
- Avoids the PCIe bottleneck
- Variable length data works great!



LiDAR: 131M points 15 – 33 FPS (C2070)

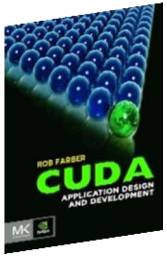


In collaboration with Global Navigation Sciences (http://http://globalnavigationsciences.com/



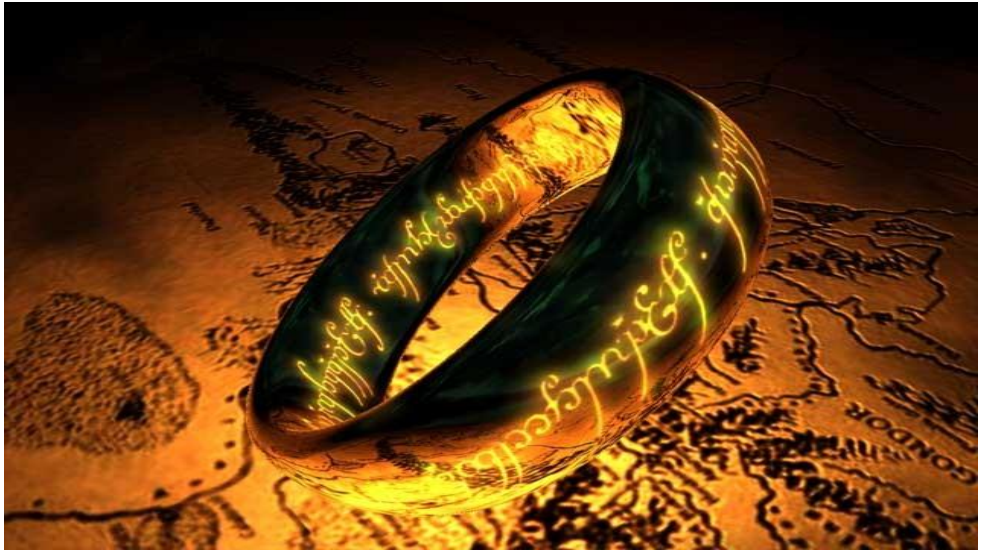
Chapter 9 Perlin Noise

Fly around in a 3D virtual world



“CUDA is for GPUs **and CPUs!** “

“One source tree to hold them all and on the GPU accelerate them!” (My parody of J.R.R. Tolkien)



Wait a minute!

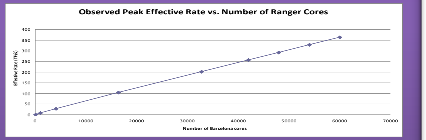
If CUDA and GPUs are so great ….

Why consider x86 at all?

1. Market accessibility
   - 1/3 Billion GPUs is a big market (Desktop, Mobile, …)
   - The number of customers who own x86 hardware is much bigger
     - *(The cellphone/tablet SOC competition may accentuate this)*

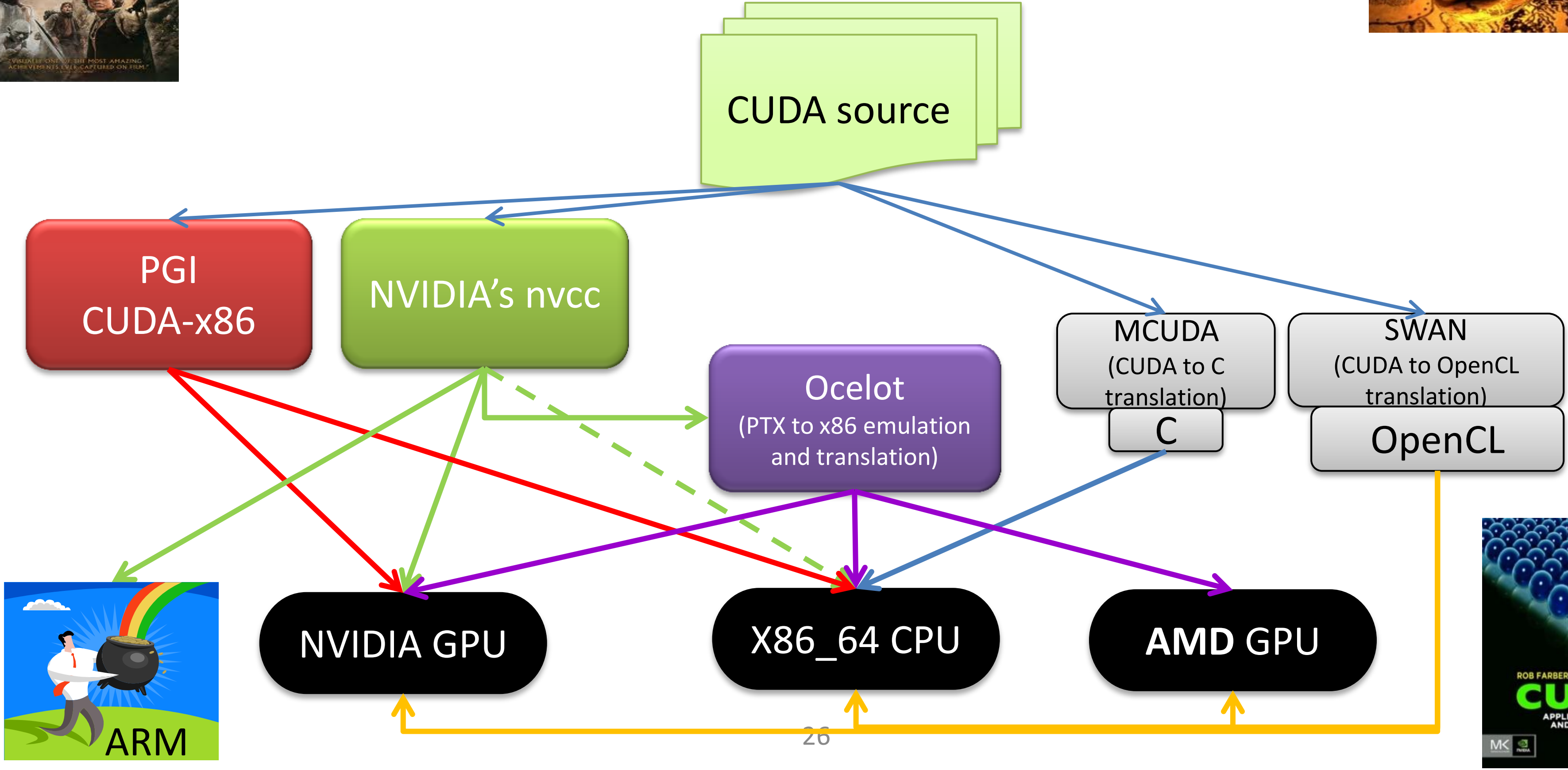
2. Achieve the biggest return on your software investment
   - One source tree saves money
   - GPU acceleration comes for free
   - CUDA is C/C++ based … not much of a change for many organizations

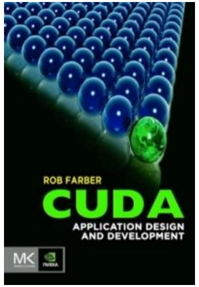
3. CUDA uses a “strong scaling” execution model
   - Very important for scalability – use a million threads … okay!
   - SIMD execution exploits x86 SIMD (e.g. SSE and AVX) instructions
   - CUDA was designed to expose parallelism to the programmer
     - Many legacy codes run faster after CUDA porting “experiments”
   - CUDA async queues (standard) -> execution graphs to control many devices



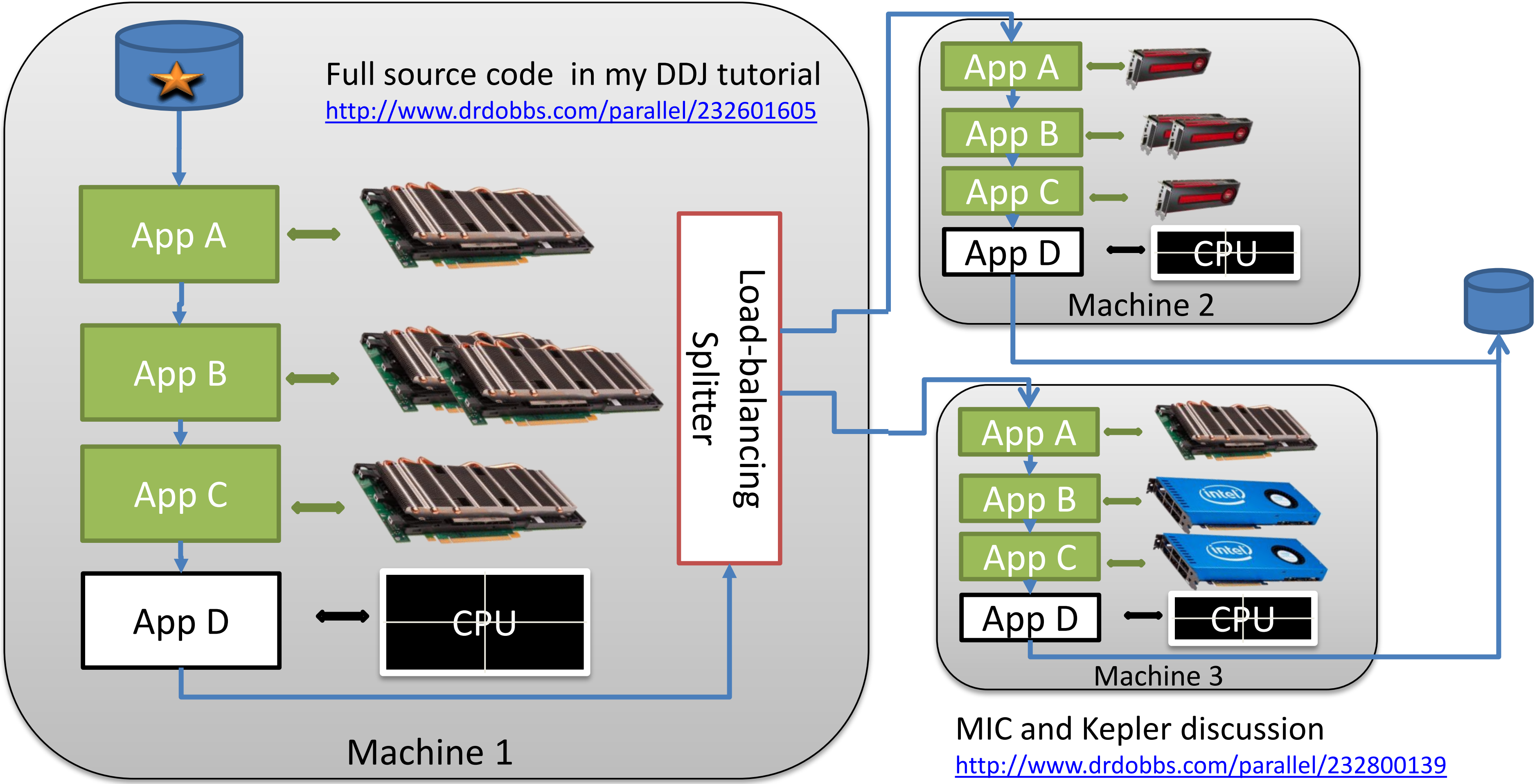
"CUDA is for GPUs **and CPUs!** "

One source tree to hold them all and on the GPU accelerate them





Fast and scalable heterogeneous workflows



Dynamically compile CUDA

(just like OpenCL)

Without dynamic plugins



No scalability

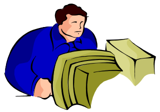
Collaborators need:

- All plugins
- For all machine types

Full source for Windows and Linux in Part 23:
http://www.drdobbs.com/parallel/232601605

With dynamic plugins

Scalable
Only need source for the plugins required



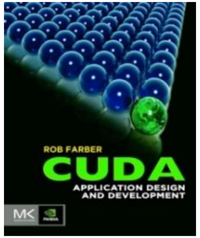
Scalable
Only need source for the plugins required

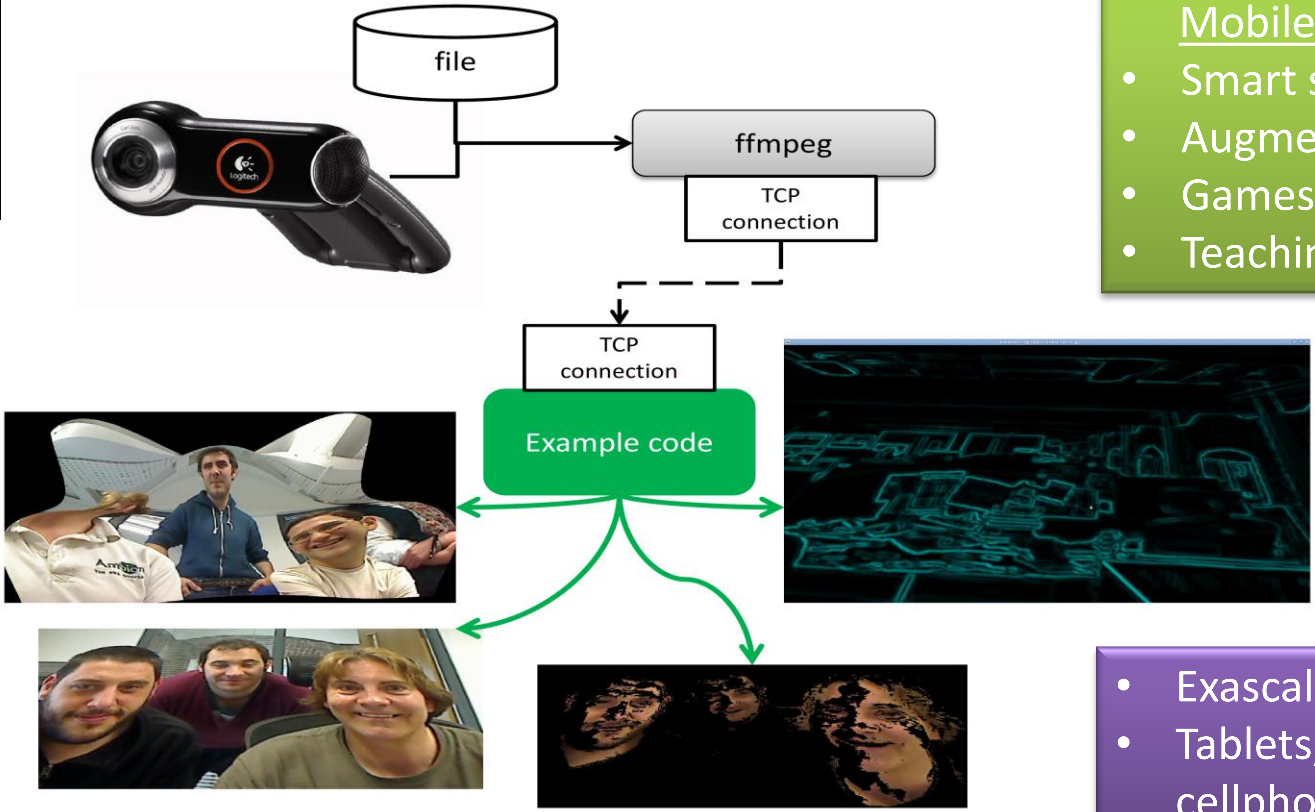
Scalable
Only need source for the plugins required

```
dynFunc vec2x < stream.dat | dynFunc reduction.cc
```

```
dynFunc vec2x < stream.dat \
      | ssh machine1 dynFunc app1 | dynFunc app2 \
      | ssh machine2 dynFunc reduction
```

A cool real-time video workflow





Mobile or desktop
- Smart sensors
- Augmented Reality
- Games
- Teaching

- Exascale video analysis
- Tablets, notebooks … cellphones?

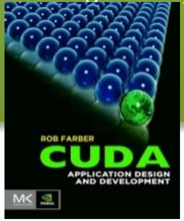
For the demo, think Kinect and 3D morphing for augmented reality

(identify flesh colored blobs for hands)

Artifacts caused by picking a colorspace rectangle rather than an ellipse

The entire segmentation method

```
__global__ void kernelSkin(float4* pos, uchar4 *colorPos,
                           unsigned int width, unsigned int height,
                           int lowPureG, int highPureG,
                           int lowPureR, int highPureR)
{
  un           blockIdx.x*blockDim.x + threadIdx.x;
               blockIdx.y*blockDim.y + threadIdx.y;
             s[y*width+x].x;
            Pos[y*width+x].y;
          orPos[y*width+x].z;
       reR = 255*( ((float)r)/(r+g+b));
      pureG = 255*( ((float)g)/(r+g+b));
  i  !( (p              ) && (pureG < highPureG)
                       PureR) && (pureR < highPureR)
                       _uchar4(0,0,0,0);
}
```



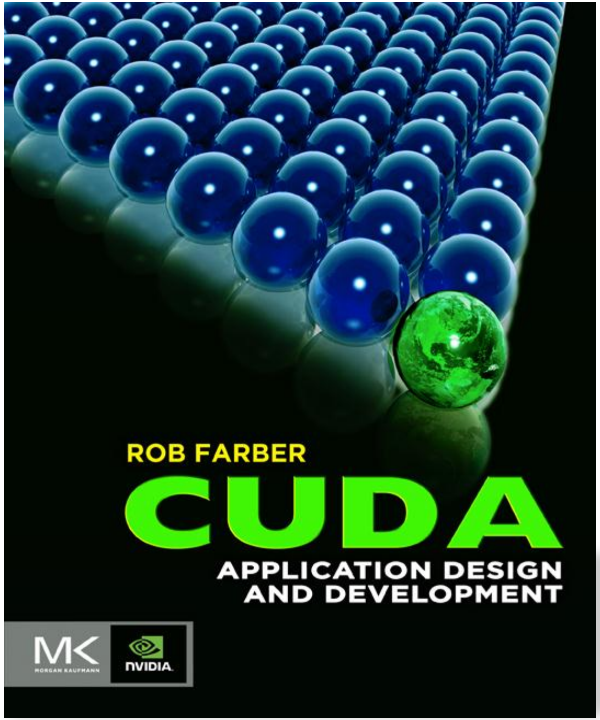
Full source code provided in *"CUDA Application Design and Development"* in print and on Kindle.

Available from many booksellers.

- Kindle version (color) is also available) http://www.amazon.com/CUDA-Application-Design-Development-Farber/dp/0123884268

The Chinese edition is coming! (interest in other translations?)

Teaching aids (PowerPoint slides, code) available on http://GPUcomputing.net/RobFarber



Chapter 12 real-time video example

- Note this demonstration is running on a battery powered laptop.
  - Think smart sensors
  - Augmented Reality
  - Many others!
- Laptop provided by NVIDIA
  - Thank you!