Adam Harries

CUDA, or how I stopped worrying and learned to love the GPU

Tuesday October 15th 2013

Overview

The talk

The talk

What this talk will be:

The talk

What this talk will be:

- Some history (of both GPUs and CUDA)

The talk

What this talk will be:

- Some history (of both GPUs and CUDA)
- A basic introduction to CUDA, how it works, and using it in Python

The talk

What this talk will be:

- Some history (of both GPUs and CUDA)
- A basic introduction to CUDA, how it works, and using it in Python
- Some basic examples of speeding up existing Python programs using CUDA

The talk

What this talk will be:

- Some history (of both GPUs and CUDA)
- A basic introduction to CUDA, how it works, and using it in Python
- Some basic examples of speeding up existing Python programs using CUDA

The talk

What this talk will be:

- Some history (of both GPUs and CUDA)
- A basic introduction to CUDA, how it works, and using it in Python
- Some basic examples of speeding up existing Python programs using CUDA

What this talk won't be

The talk

What this talk will be:

- Some history (of both GPUs and CUDA)
- A basic introduction to CUDA, how it works, and using it in Python
- Some basic examples of speeding up existing Python programs using CUDA

What this talk won't be

- A deep, massively complicated exploration of the CUDA programming model

The talk

What this talk will be:

- Some history (of both GPUs and CUDA)
- A basic introduction to CUDA, how it works, and using it in Python
- Some basic examples of speeding up existing Python programs using CUDA

What this talk won't be

- A deep, massively complicated exploration of the CUDA programming model
- An in-depth guide to optimizing your CUDA programs (see Aidan Chalk about that)

The talk

What this talk will be:

- Some history (of both GPUs and CUDA)
- A basic introduction to CUDA, how it works, and using it in Python
- Some basic examples of speeding up existing Python programs using CUDA

What this talk won't be

- A deep, massively complicated exploration of the CUDA programming model
- An in-depth guide to optimizing your CUDA programs (see Aidan Chalk about that)
- A lecture on parallel algorithms

The talk

What this talk will be:

- Some history (of both GPUs and CUDA)
- A basic introduction to CUDA, how it works, and using it in Python
- Some basic examples of speeding up existing Python programs using CUDA

What this talk won't be

- A deep, massively complicated exploration of the CUDA programming model
- An in-depth guide to optimizing your CUDA programs (see Aidan Chalk about that)
- A lecture on parallel algorithms
- Coherent

History

First, some history

GPU - Speeding up graphics computation since 1983

First, some history

GPU - Speeding up graphics computation since 1983

- Dedicated cards for handling graphical rendering

First, some history

GPU - Speeding up graphics computation since 1983

- Dedicated cards for handling graphical rendering
- Generally 3d acceleration today, but started off with 2d acceleration

First, some history

GPU - Speeding up graphics computation since 1983

- Dedicated cards for handling graphical rendering
- Generally 3d acceleration today, but started off with 2d acceleration
- Arguably first card: Intel iSBX 275 Video Graphics Controller Multimodule Board

Intel iSBX 275

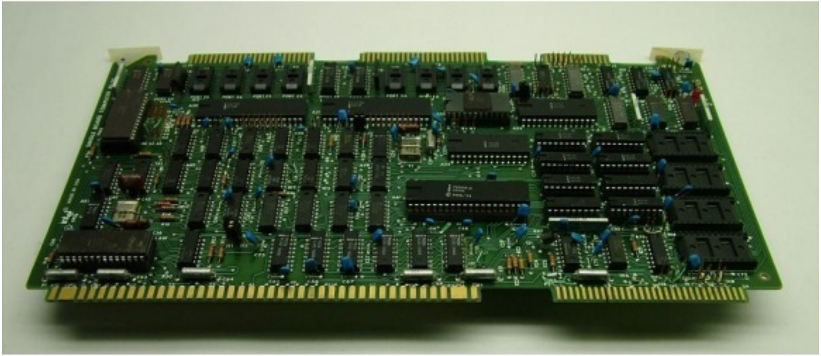


Figure : The Intel ISBX, because everyone loves dual in line packaging. . .

The arrival of 3d

Dedicated 3d graphics hardware speedup

The arrival of 3d

Dedicated 3d graphics hardware speedup

- 3dfx interactive's "Voodoo 1" card

The arrival of 3d

Dedicated 3d graphics hardware speedup

- 3dfx interactive's "Voodoo 1" card
- Dedicated 3d graphics card

The arrival of 3d

Dedicated 3d graphics hardware speedup

- 3dfx interactive's "Voodoo 1" card
- Dedicated 3d graphics card
- needed to be piggybacked with a 2d card

Voodoo 1 card

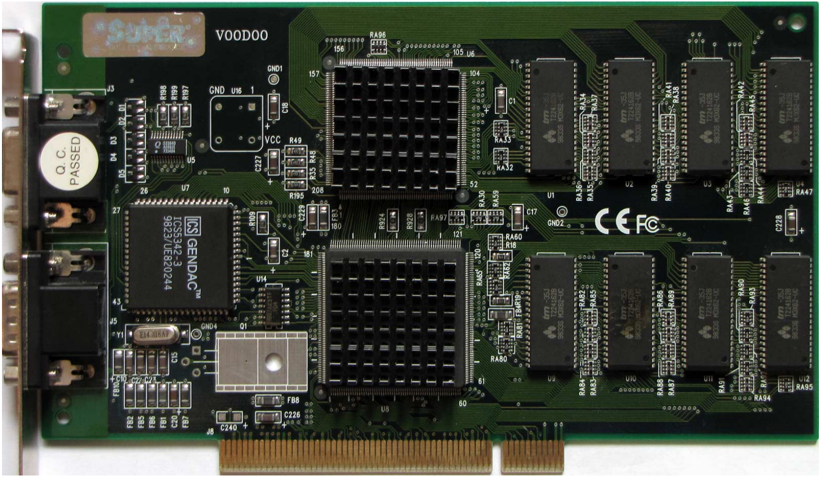


Figure : The 3dfx voodoo 1 card. Note the dual VGA ports, for piggybacking off a 2d card.

Programmable pipeline

The advent of programming on the graphics card

- Programmers could now define “shader programs”

Programmable pipeline

The advent of programming on the graphics card

- Programmers could now define “shader programs”
- First graphics card with support: nVidia GeForce 3

Programmable pipeline

The advent of programming on the graphics card

- Programmers could now define “shader programs”
- First graphics card with support: nVidia GeForce 3
- Only 12 years ago - in 2001

nVidia Geforce 3

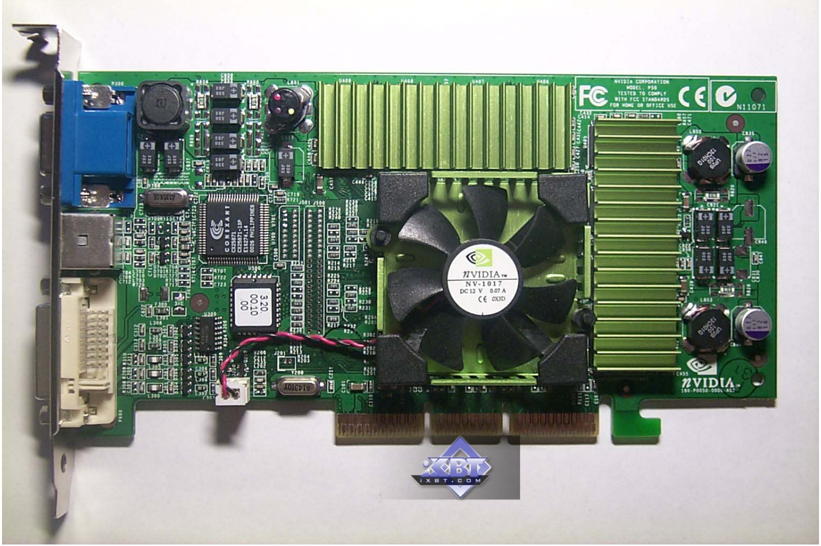


Figure : The first graphics card with a programmable pipeline

CUDA arrives!

- Only available on nVidia cards (obviously)

CUDA arrives!

- Only available on nVidia cards (obviously)
- Released to public in

CUDA arrives!

- Only available on nVidia cards (obviously)
- Released to public in
- First supporting card: 8800GTX

8800GTX

CUDA: an overview

Kernels

- CUDA is based upon the concept of kernels

Kernels

- CUDA is based upon the concept of kernels
- similar to functions

Kernels

- CUDA is based upon the concept of kernels
- similar to functions
- written in a dialect of C++

Kernels

- CUDA is based upon the concept of kernels
- similar to functions
- written in a dialect of C++
- run on the GPU

Kernels

- CUDA is based upon the concept of kernels
- similar to functions
- written in a dialect of C++
- run on the GPU
- one kernel per thread

Kernels

- CUDA is based upon the concept of kernels
- similar to functions
- written in a dialect of C++
- run on the GPU
- one kernel per thread

Kernels

- CUDA is based upon the concept of kernels
- similar to functions
- written in a dialect of C++
- run on the GPU
- one kernel per thread

Example:

```
__global__ void double(int *a)
{
    const int id = threadIdx.x;
    a[id] = 2*a[id];
}
```

Supplementary functions

- sometimes we can't fit everything in a single function

Supplementary functions

- sometimes we can't fit everything in a single function
- use `__device__` functions, callable from other function running on GPU

Supplementary functions

- sometimes we can't fit everything in a single function
- use `__device__` functions, callable from other function running on GPU
- `__global__` functions can't be called from anything, apart from invocation

Supplementary functions

- sometimes we can't fit everything in a single function
- use `__device__` functions, callable from other function running on GPU
- `__global__` functions can't be called from anything, apart from invocation
- `__device__` functions can't be invoked by the CPU, must be called by another function on GPU

Supplementary functions

- sometimes we can't fit everything in a single function
- use `__device__` functions, callable from other function running on GPU
- `__global__` functions can't be called from anything, apart from invocation
- `__device__` functions can't be invoked by the CPU, must be called by another function on GPU

Supplementary functions

- sometimes we can't fit everything in a single function
- use `__device__` functions, callable from other function running on GPU
- `__global__` functions can't be called from anything, apart from invocation
- `__device__` functions can't be invoked by the CPU, must be called by another function on GPU

```
__device__ int square(int x)
{
    return x*x;
}
__global__ void double_and_square(int *a)
{
    const int id = threadIdx.x;
    a[id] = 2*square(a[id]);
}
```

Running lots of kernels at once

- CUDA uses “thread blocks” to achieve parallelism

Running lots of kernels at once

- CUDA uses “thread blocks” to achieve parallelism
- define a dimension for the blocks running on

Running lots of kernels at once

- CUDA uses “thread blocks” to achieve parallelism
- define a dimension for the blocks running on
- define a dimension for threads within blocks

Running lots of kernels at once

- CUDA uses “thread blocks” to achieve parallelism
- define a dimension for the blocks running on
- define a dimension for threads within blocks
- limitations depending on card being used

Running lots of kernels at once

- CUDA uses “thread blocks” to achieve parallelism
- define a dimension for the blocks running on
- define a dimension for threads within blocks
- limitations depending on card being used
- use variables based on dimensions to tell kernel what ID it is

Some examples

An old lab

Visualising the interference of waves using a discrete grid of cells

An old lab

Visualising the interference of waves using a discrete grid of cells

- this is an actual exercise from last years Year 1 physics lab module

An old lab

Visualising the interference of waves using a discrete grid of cells

- this is an actual exercise from last years Year 1 physics lab module
- I haven't done it how it was recommended by the physics department

An old lab

Visualising the interference of waves using a discrete grid of cells

- this is an actual exercise from last years Year 1 physics lab module
- I haven't done it how it was recommended by the physics department
- nested for Loops over an array to calculate the value at each cell

An old lab

Visualising the interference of waves using a discrete grid of cells

- this is an actual exercise from last years Year 1 physics lab module
- I haven't done it how it was recommended by the physics department
- nested for Loops over an array to calculate the value at each cell
- both versions (cuda and traditional) use the same formula to calculate the value at each cell, from each wave (sin of hypotenuse):

An old lab

Visualising the interference of waves using a discrete grid of cells

- this is an actual exercise from last years Year 1 physics lab module
- I haven't done it how it was recommended by the physics department
- nested for Loops over an array to calculate the value at each cell
- both versions (cuda and traditional) use the same formula to calculate the value at each cell, from each wave (sin of hypotenuse):

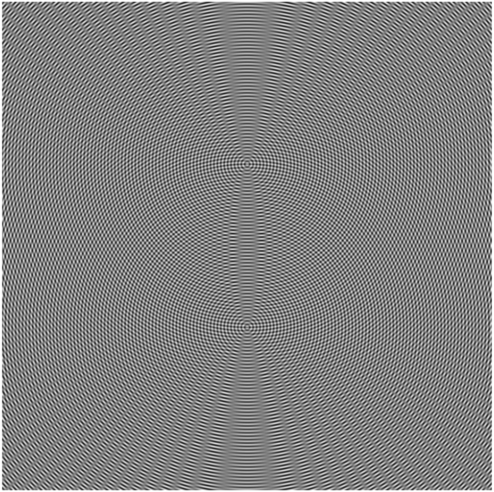
An old lab

Visualising the interference of waves using a discrete grid of cells

- this is an actual exercise from last years Year 1 physics lab module
- I haven't done it how it was recommended by the physics department
- nested for Loops over an array to calculate the value at each cell
- both versions (cuda and traditional) use the same formula to calculate the value at each cell, from each wave (sin of hypotenuse):

$$Wavevalue = \sin(\sqrt{(x - s_x)^2 + (y - s_y)^2})$$

The result



“Tradtional” loops version:

```
from numpy import *
import matplotlib.pyplot as plt
wav_array = zeros((300,300), dtype=float32)
pa = (300,450)
pb = (600,450)
for i in range(300):
    for j in range(300):
        adist = sin( sqrt( (i-pa[0])**2 + (j-pa[1])**2))
        bdist = sin( sqrt( (i-pb[0])**2 + (j-pb[1])**2))
        wav_array[i][j] = (adist+bdist)/2
plt.imshow(wav_array)
plt.clim(-1,1);
plt.set_cmap('gray')
plt.axis('off')
plt.show()
```

And some analysis

- not that fast - $O(n^2)$ in fact

And some analysis

- not that fast - $O(n^2)$ in fact
- still not as slow as it could be

And some analysis

- not that fast - $O(n^2)$ in fact
- still not as slow as it could be
- still not as fast either (eg, could be in C)

And some analysis

- not that fast - $O(n^2)$ in fact
- still not as slow as it could be
- still not as fast either (eg, could be in C)
- timings (from unix time utility, run on my laptop):

And some analysis

- not that fast - $O(n^2)$ in fact
- still not as slow as it could be
- still not as fast either (eg, could be in C)
- timings (from unix time utility, run on my laptop):

And some analysis

- not that fast - $O(n^2)$ in fact
- still not as slow as it could be
- still not as fast either (eg, could be in C)
- timings (from unix time utility, run on my laptop):

  real 0m12.579s

  user 0m10.220s

  sys 0m0.070s

- still faster than the way recommended by the physics department

The CUDA version

- the CUDA version is more difficult to display in a talk

The CUDA version

- the CUDA version is more difficult to display in a talk
- a little longer - but due to structuring, not difficulty

The CUDA version

- the CUDA version is more difficult to display in a talk
- a little longer - but due to structuring, not difficulty
- comprised of two main parts: kernel code, and calling/structure (CPU) code

The CUDA version

- the CUDA version is more difficult to display in a talk
- a little longer - but due to structuring, not difficulty
- comprised of two main parts: kernel code, and calling/structure (CPU) code
- all CPU code in Python

The CUDA version

- the CUDA version is more difficult to display in a talk
- a little longer - but due to structuring, not difficulty
- comprised of two main parts: kernel code, and calling/structure (CPU) code
- all CPU code in Python
- kernel written in CUDA (somewhat like C)

The structure/CPU code

```
from numpy import *
import matplotlib.pyplot as plt
import pycuda.compiler as comp
import pycuda.driver as drv
import pycuda.autoinit
mod = comp.SourceModule("""
#kernel declaration, of "wave_kernel"
""")
wav_array = zeros((900,900), dtype=float32)
wav_array_gpu = drv.mem_alloc(wav_array.nbytes)
make_waves = mod.get_function("wave_kernel")
make_waves(wav_array_gpu,block=(30,30,1), grid=(30,30))
drv.memcpy_dtoh(wav_array, wav_array_gpu)
```

Walking through the code

Generic imports, with the most interesting ones:

Walking through the code

Generic imports, with the most interesting ones:

```
import pycuda.compiler as comp
import pycuda.driver as drv
import pycuda.autoinit
```

Walking through the code

Generic imports, with the most interesting ones:

```
import pycuda.compiler as comp
import pycuda.driver as drv
import pycuda.autoinit
```

They respectively:

Walking through the code

Generic imports, with the most interesting ones:

```python
import pycuda.compiler as comp
import pycuda.driver as drv
import pycuda.autoinit
```

They respectively:

- give access to nvcc (nvidia cuda compiler)

Walking through the code

Generic imports, with the most interesting ones:

```
import pycuda.compiler as comp
import pycuda.driver as drv
import pycuda.autoinit
```

They respectively:

- give access to nvcc (nvidia cuda compiler)
- interface with CUDA, and provide a host of useful abstractions

Walking through the code

Generic imports, with the most interesting ones:

```
import pycuda.compiler as comp
import pycuda.driver as drv
import pycuda.autoinit
```

They respectively:

- give access to nvcc (nvidia cuda compiler)
- interface with CUDA, and provide a host of useful abstractions
- initialise much of what CUDA does, including getting kernels ready to launch

More walking

```
mod = comp.SourceModule("""
#kernel declaration, of "wave_kernel"
""")
```

More walking

```
mod = comp.SourceModule("""
#kernel declaration, of "wave_kernel"
""")

wav_array = zeros((900,900), dtype=float32)
wav_array_gpu = drv.mem_alloc(wav_array.nbytes)
```

More walking

```
mod = comp.SourceModule("""
#kernel declaration, of "wave_kernel"
""")

wav_array = zeros((900,900), dtype=float32)
wav_array_gpu = drv.mem_alloc(wav_array.nbytes)

make_waves = mod.get_function("wave_kernel")
make_waves(wav_array_gpu,block=(30,30,1), grid=(30,30))
```

More walking

```
mod = comp.SourceModule("""
#kernel declaration, of "wave_kernel"
""")

wav_array = zeros((900,900), dtype=float32)
wav_array_gpu = drv.mem_alloc(wav_array.nbytes)

make_waves = mod.get_function("wave_kernel")
make_waves(wav_array_gpu,block=(30,30,1), grid=(30,30))

drv.memcpy_dtoh(wav_array, wav_array_gpu)
```

The kernel code

```
__global__ void wave_kernel(float *wa)
{
    const int j = threadIdx.y+(blockIdx.y*gridDim.y);
    const int i = threadIdx.x+(blockIdx.x*gridDim.x);
    const int lID = i+(j*blockDim.y*gridDim.y);
    float pa[] = {300,450};
    float pb[] = {600,450};
    float a,b;
    a = sqrt(((i-pa[0])*(i-pa[0]))+((j-pa[1])*(j-pa[1])));
    b = sqrt(((i-pb[0])*(i-pb[0]))+((j-pb[1])*(j-pb[1])));
    wa[lID] = (sin(a)+sin(b))/2;
}
```

Strolling through the kernel

```
const int j = threadIdx.y+(blockIdx.y*gridDim.y);
const int i = threadIdx.x+(blockIdx.x*gridDim.x);
const int lID = i+(j*blockDim.y*gridDim.y);
```

Strolling through the kernel

```
const int j = threadIdx.y+(blockIdx.y*gridDim.y);
const int i = threadIdx.x+(blockIdx.x*gridDim.x);
const int lID = i+(j*blockDim.y*gridDim.y);
```

- Threads getting information about themselves

Strolling through the kernel

```
const int j = threadIdx.y+(blockIdx.y*gridDim.y);
const int i = threadIdx.x+(blockIdx.x*gridDim.x);
const int lID = i+(j*blockDim.y*gridDim.y);
```

- Threads getting information about themselves
- Allows threads to determine which parts of problems they should work on

Strolling through the kernel

```
const int j = threadIdx.y+(blockIdx.y*gridDim.y);
const int i = threadIdx.x+(blockIdx.x*gridDim.x);
const int lID = i+(j*blockDim.y*gridDim.y);
```

- Threads getting information about themselves
- Allows threads to determine which parts of problems they should work on
- in this case, (i,j) is the cell, while lID is the element of the final array

Some analysis

- Arguably an order of magnitude faster than the serial method

Some analysis

- Arguably an order of magnitude faster than the serial method
- Still some slowdowns in copying to/from GPU, but shouldn't be a problem if clever about it, and not constantly transferring data

Some analysis

- Arguably an order of magnitude faster than the serial method
- Still some slowdowns in copying to/from GPU, but shouldn't be a problem if clever about it, and not constantly transferring data
- Slow to invoke CUDA and kernels

Some analysis

- Arguably an order of magnitude faster than the serial method
- Still some slowdowns in copying to/from GPU, but shouldn't be a problem if clever about it, and not constantly transferring data
- Slow to invoke CUDA and kernels
- Timings (from unix time utility, on my laptop):

Some analysis

- Arguably an order of magnitude faster than the serial method
- Still some slowdowns in copying to/from GPU, but shouldn't be a problem if clever about it, and not constantly transferring data
- Slow to invoke CUDA and kernels
- Timings (from unix time utility, on my laptop):

Some analysis

- Arguably an order of magnitude faster than the serial method
- Still some slowdowns in copying to/from GPU, but shouldn't be a problem if clever about it, and not constantly transferring data
- Slow to invoke CUDA and kernels
- Timings (from unix time utility, on my laptop):

  real 0m5.028s

  user 0m0.337s

  sys 0m0.087s

Some analysis

- Arguably an order of magnitude faster than the serial method
- Still some slowdowns in copying to/from GPU, but shouldn't be a problem if clever about it, and not constantly transferring data
- Slow to invoke CUDA and kernels
- Timings (from unix time utility, on my laptop):

  real 0m5.028s

  user 0m0.337s

  sys 0m0.087s

- note disparity between real and user/sys: very little CPU time, lots of GPU

Some analysis

- Arguably an order of magnitude faster than the serial method
- Still some slowdowns in copying to/from GPU, but shouldn't be a problem if clever about it, and not constantly transferring data
- Slow to invoke CUDA and kernels
- Timings (from unix time utility, on my laptop):

  real 0m5.028s

  user 0m0.337s

  sys 0m0.087s

- note disparity between real and user/sys: very little CPU time, lots of GPU
- however, still faster than serial.

But still not faster than C

```c
#include <stdio.h>
#include <stdlib.h>
#include <math.h>
int main(int argc, char** argv){
 float *wav_array = malloc(sizeof(float)*900*900);
 float pa[] = {300,450};
 float pb[] = {600,450};
 int index = 0; float a,b;
 for(int i = 0;i<900;i++){
  for(int j = 0;j<900;j++){
   index = i+(j*900);
   a = sqrt(((i-pa[0])*(i-pa[0]))+((j-pa[1])*(j-pa[1])));
   b = sqrt(((i-pb[0])*(i-pb[0]))+((j-pb[1])*(j-pb[1])));
   wav_array[index] = (sin(a)+sin(b))/2;
  }
 }
 return 0;
}
```

C analysis

- Timings:

C analysis

- Timings:

C analysis

- Timings:

  real 0m0.110s

  user 0m0.107s

  sys 0m0.003s

C analysis

- Timings:

  real 0m0.110s

  user 0m0.107s

  sys 0m0.003s

- what???

C analysis

- Timings:

  real 0m0.110s

  user 0m0.107s

  sys 0m0.003s

- what???
- not the fault of CUDA, more the fault of python and poor programming

C analysis

- Timings:

  real 0m0.110s

  user 0m0.107s

  sys 0m0.003s

- what???
- not the fault of CUDA, more the fault of python and poor programming
- for larger problems, with more complex processing, we get a bigger speedup with CUDA

Rewrite the CUDA in C?

```
/*Same kernel definition */
int main(int argc, char** argv){
    float *wav_array = (float*)malloc(sizeof(float)*900*90
    float *gpu_wav_array;
    cudaMalloc(&gpu_wav_array, 900*900*sizeof(float));
    dim3 block_size;
    block_size.x = 30;
    block_size.y = 30;
    block_size.z = 1;
    dim3 grid_size;
    grid_size.x = 30;
    grid_size.y = 30;
    sin_dist<<<block_size, grid_size>>>(gpu_wav_array);
    cudaMemcpy(wav_array, gpu_wav_array, 900*900*sizeof(fl
        cudaMemcpyDeviceToHost);
    return 0;
}
```

Analysis:

- Timings:

  real 0m4.511s

  user 0m0.017s

  sys 0m0.047s

Analysis:

- Timings:

  real 0m4.511s

  user 0m0.017s

  sys 0m0.047s

- Again, slower than C??

Analysis:

- Timings:

  real 0m4.511s

  user 0m0.017s

  sys 0m0.047s

- Again, slower than C??
- yes, what we're doing in the graphics card isn't that hard in serial

Analysis:

- Timings:

  real 0m4.511s

  user 0m0.017s

  sys 0m0.047s

- Again, slower than C??
- yes, what we're doing in the graphics card isn't that hard in serial
- slowdown switching between CPU and GPU

Analysis:

- Timings:

  real 0m4.511s

  user 0m0.017s

  sys 0m0.047s

- Again, slower than C??
- yes, what we're doing in the graphics card isn't that hard in serial
- slowdown switching between CPU and GPU
- however, faster than PyCUDA - so some speedup gaineds

Analysis:

- Timings:

  real 0m4.511s

  user 0m0.017s

  sys 0m0.047s

- Again, slower than C??
- yes, what we're doing in the graphics card isn't that hard in serial
- slowdown switching between CPU and GPU
- however, faster than PyCUDA - so some speedup gaineds
- more importantly, we use the same CUDA code, so if we want to squeeze speed out, we can rewrite in C

Questions!

Any questions?