PyCUDA Crash course

Jonas Einarsson

8 February 2012

Agenda

- GPU computing – data parallelism
- General workflow
- What does PyCUDA do for you?
- Examples

- After class: Slides and source code available

Why I'm excited

- My workstation: Quadro 600 (96 cores)
- A cluster-grade Tesla card (448 cores)
- The test machine Soon(tm) available from C3SE
- SNIC GPU cluster in Lund (guesstimate)
- Top500 computer as of July 2011

Data parallelism

- When can we make use of the GPU’s power?

Data parallelism

- When can we make use of the GPU's power?
- Simplest case:

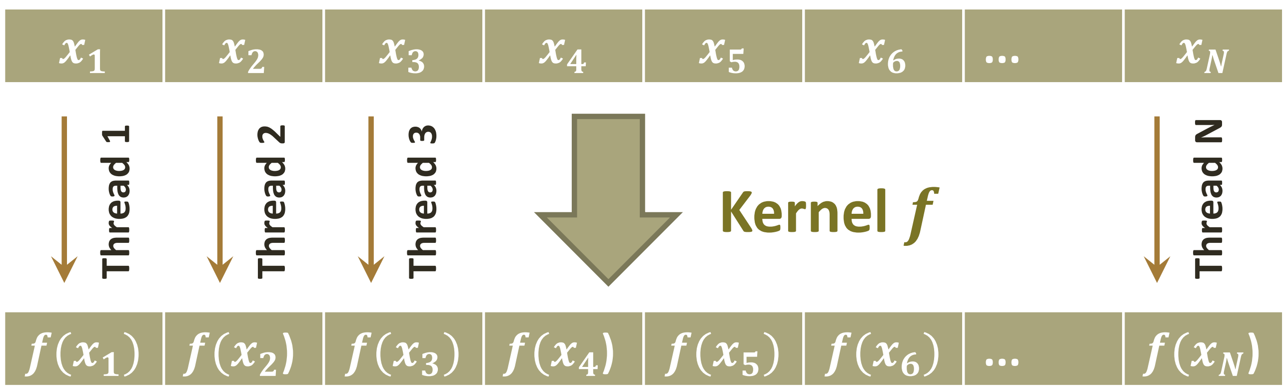
x_1	x_2	x_3	x_4	x_5	x_6	…	x_N

$\Downarrow$ **Kernel f**

$f(x_1)$	$f(x_2)$	$f(x_3)$	$f(x_4)$	$f(x_5)$	$f(x_6)$	…	$f(x_N)$

Data parallelism

- When can we make use of the GPU's power?
- Simplest case:



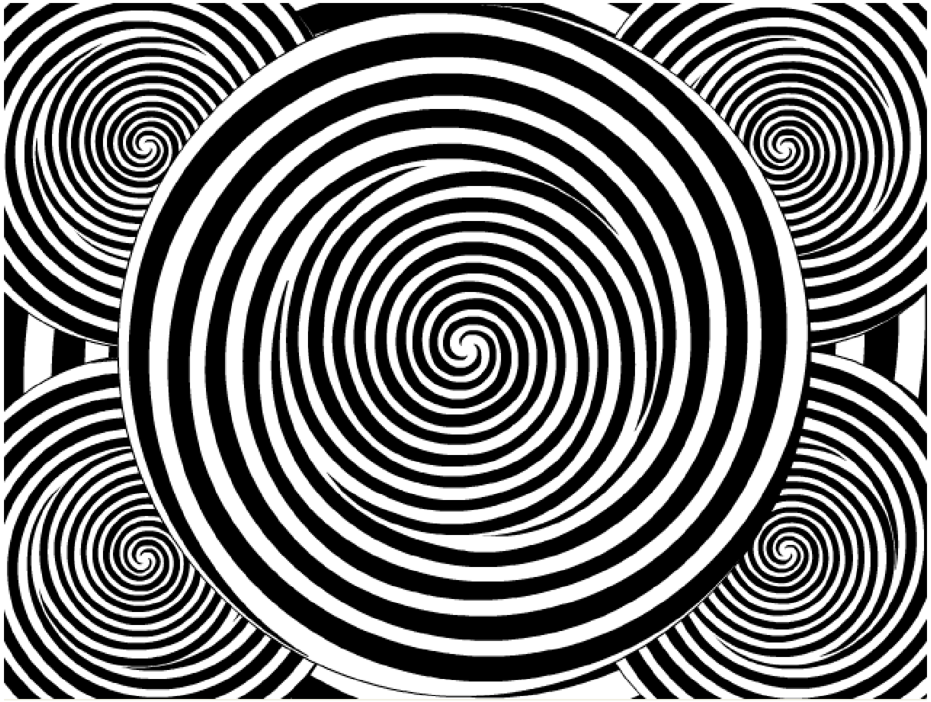
- Each element gets handles by one *thread*
- Compare to Map()

Words, words, words

- A *Kernel* is a function run by a thread
- A *Thread* is the abstraction of a function call on some data
- Threads are launched in groups called *Blocks*
- Blocks are in turn organised in a *Grid*

Words, words, words

- A *Kernel* is a function run by a thread
- A *Thread* is the abstraction of a function call on some data
- Threads are launched in groups called *Blocks*
- Blocks are in turn organised in a *Grid*



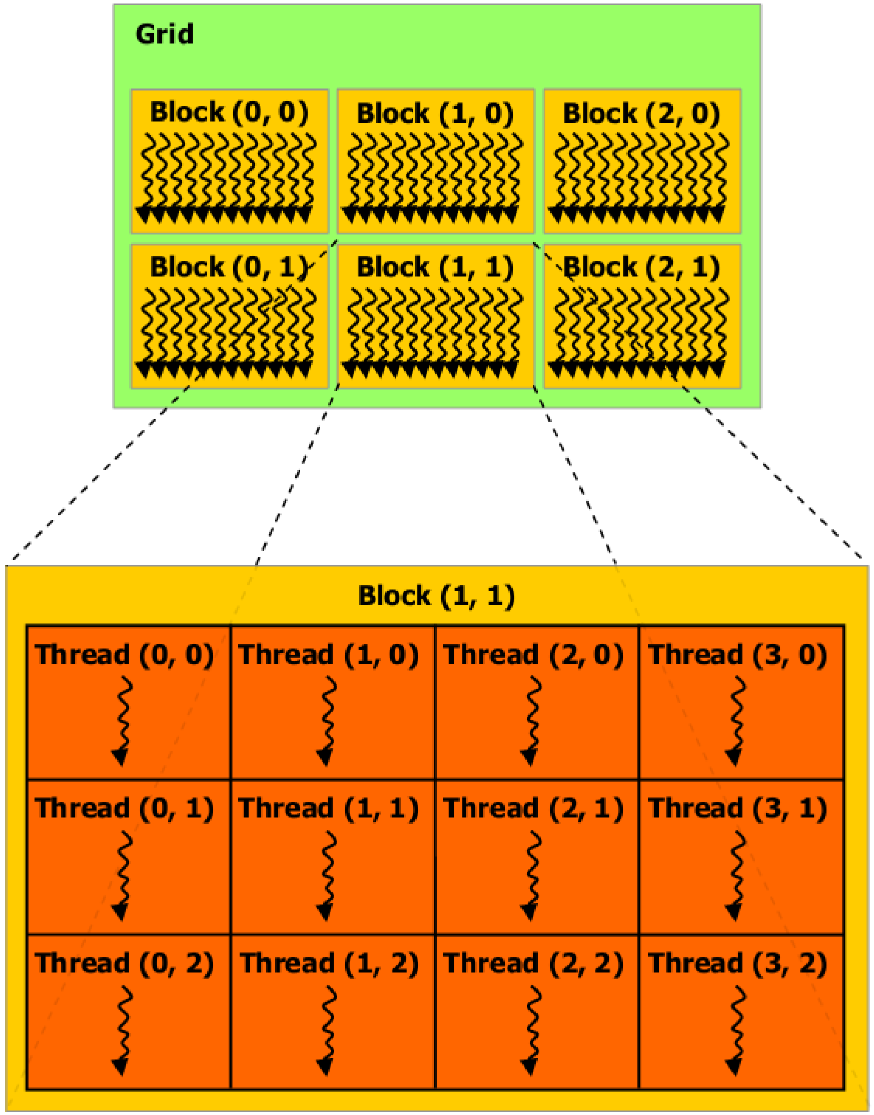
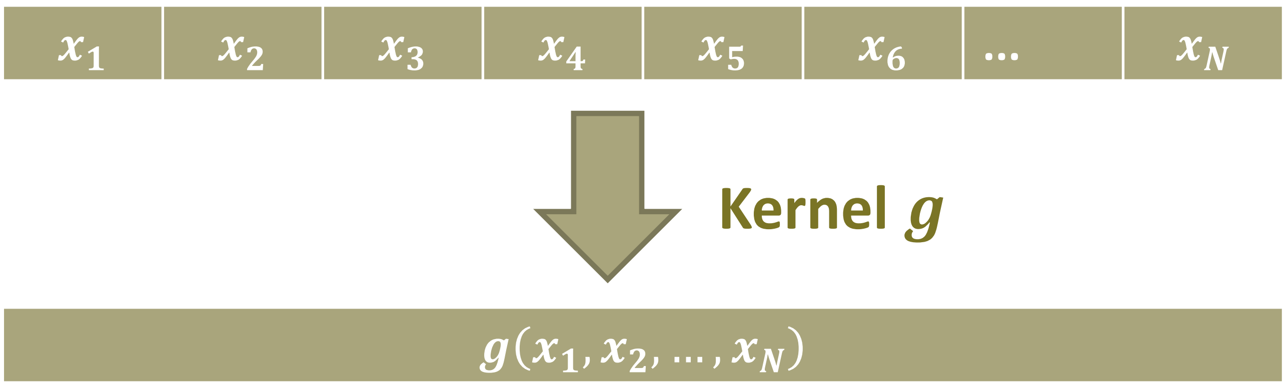


Figure 2-1. Grid of Thread Blocks

Figure from CUDA C Programming Guide page 9

More complicated cases

- Many problems require aggregating data



- Cannot easily be mapped one-to-one on *Threads*
- For example: Sum all elements in a list, or construct histogram of large arrays
- Compare Reduce()

Typical workflow

CPU

1. Initialise memory (in RAM) and transfer to GPU.

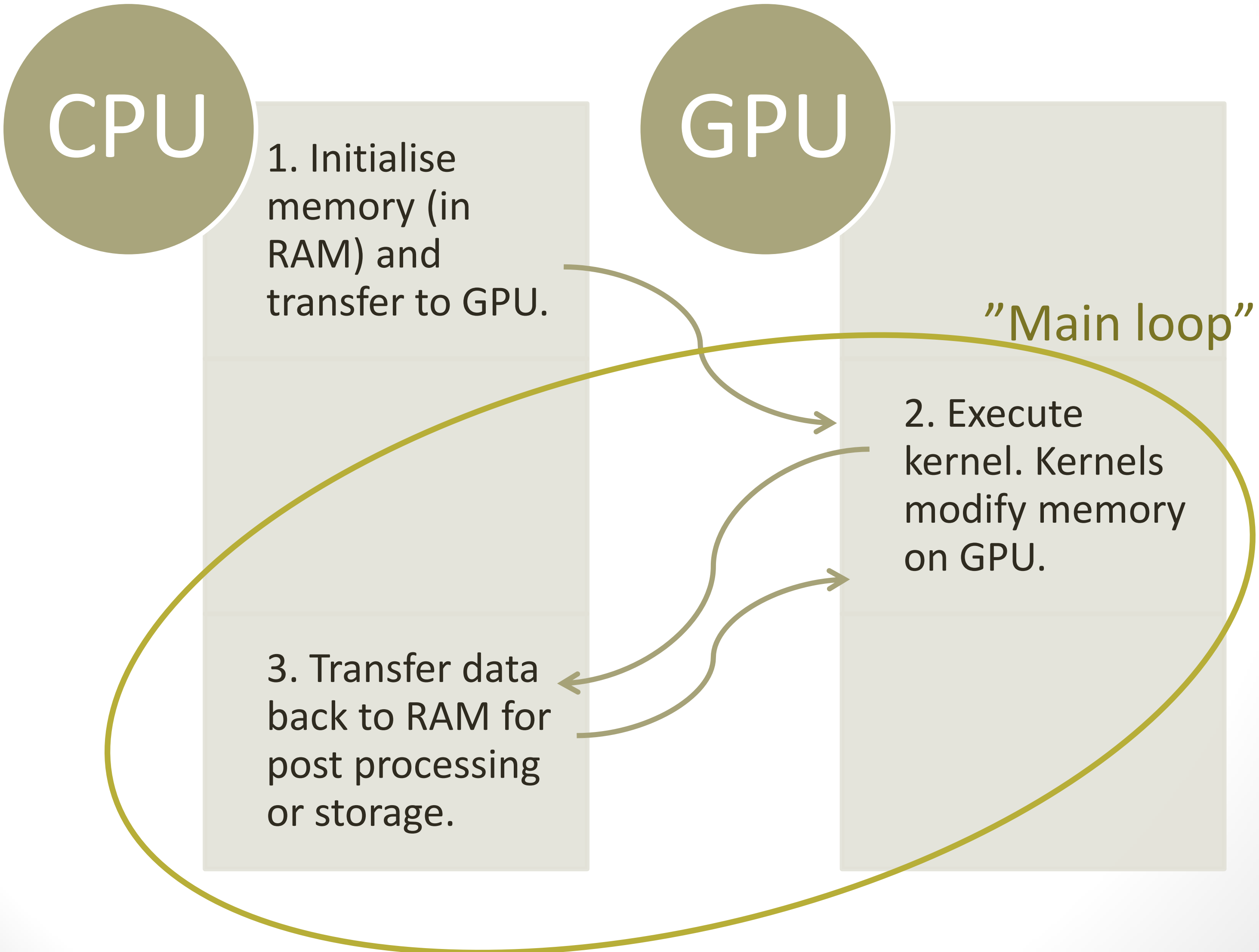
GPU

2. Execute kernel. Kernels modify memory on GPU.

CPU

3. Transfer data back to RAM for post processing or storage.

Typical workflow



What does PyCUDA give?

- All tedious boilerplate code
- Use your numpy arrays directly with the GPU
- Simple operations can be built without caring about threads, blocks and all that jazz

But..

- For most ”real” problems, we still have to write, at least partially, GPU kernels in C

- In short: PyCUDA takes away the boring, and leaves the fun

See if PyCUDA finds your GPU

```python
import pycuda.driver as drv

drv.init()
print "%d device(s) found." % drv.Device.count()

for ordinal in range(drv.Device.count()):
    dev = drv.Device(ordinal)
    print "Device #%d: %s" % (ordinal, dev.name())
    print "  Compute Capability: %d.%d" % dev.compute_capability()
    print "  Total Memory: %s KB" % (dev.total_memory()//(1024))
```

```
d:\Phd\Teaching\PyCUDA\programs>list_gpus.py
1 device(s) found.
Device #0: Quadro 600
  Compute Capability: 2.1
  Total Memory: 984768 KB

d:\Phd\Teaching\PyCUDA\programs>_
```

Elementwise operations

- We're used to numpy operating on elements like this

```
import numpy

size = 1e7
X = numpy.linspace(1,size,size).astype(numpy.float32)
Y = numpy.sin(X)
```

- How does this translate to PyCUDA?

Elementwise - gpuarray

- gpuarray is a numpy-array look-alike on the GPU

Elementwise - gpuarray

- gpuarray is a numpy-array look-alike on the GPU

```
import pycuda.gpuarray as gpuarray
import pycuda.cumath as cumath
import pycuda.autoinit
import numpy

size = 1e7
X = numpy.linspace(1,size,size).astype(numpy.float32)
X_gpu = gpuarray.to_gpu(X) # 1. transfer -> gpu
Y_gpu = cumath.sin(X_gpu)  # 2. execute kernel
Y = Y_gpu.get()                 # 3. retrieve result
```

Elementwise timing

```
d:\Phd\Teaching\PyCUDA\programs>timeit elementwise_numpy.py
3 loops, best of 3: 438 msec per loop

d:\Phd\Teaching\PyCUDA\programs>timeit elementwise_gpuarray.py
3 loops, best of 3: 162 msec per loop

d:\Phd\Teaching\PyCUDA\programs>
```

- Numpy 438 ms, PyCUDA 162 ms
- Not a bad return on almost no work!

Elementwise + sum

- Probably your expression is a bit more complicated, like this
- $\sum \cos x \, e^{\sin x - \sqrt{x^2}}$

```
import numpy

size = 1e7
X = numpy.linspace(1,size,size).astype(numpy.float32)
Y = numpy.cos(X)*numpy.exp(numpy.sin(X)-numpy.sqrt(X*X))
answer = numpy.sum(Y)
```

- How can we do this with gpuarray?

Elementwise + sum

- This time we will create a kernel from a template

```
import pycuda.gpuarray as gpuarray
from pycuda.elementwise import ElementwiseKernel
import pycuda.autoinit
import numpy

involved_kernel = ElementwiseKernel(
        "float *y, float *x",
        "y[i] = cos(x[i])*exp(sin(x[i])-sqrt(x[i]*x[i]))",
        "involved_kernel")

size = 1e7
X = numpy.linspace(1,size,size).astype(numpy.float32)
X_gpu = gpuarray.to_gpu(X)
Y_gpu = gpuarray.empty_like(X_gpu) # 1. transfer to GPU
involved_kernel(Y_gpu, X_gpu) # 2. execute kernel
answer = gpuarray.sum(Y_gpu) # 3. use PyCUDA sum reduction
```

Elementwise + sum

- This time we will create a kernel from a template

```
import pycuda.gpuarray as gpuarray
from pycuda.elementwise import ElementwiseKernel
import pycuda.autoinit
import numpy

involved_kernel = ElementwiseKernel(
        "float *y, float *x",
        "y[i] = cos(x[i])*exp(sin(x[i])-sqrt(x[i]*x[i]))",
        "involved_kernel")

size = 1e7
X = numpy.linspace(1,size,size).astype(numpy.float32)
X_gpu = gpuarray.to_gpu(X)
Y_gpu = gpuarray.empty_like(X_gpu) # 1. transfer to GPU
involved_kernel(Y_gpu, X_gpu) # 2. execute kernel
answer = gpuarray.sum(Y_gpu) # 3. use PyCUDA sum reduction
```

Elementwise + sum timing

```
d:\Phd\Teaching\PyCUDA\programs>timeit elementwise_numpy_involved.py
3 loops, best of 3: 1.1 sec per loop

d:\Phd\Teaching\PyCUDA\programs>timeit elementwise_gpuarray_involved.py
3 loops, best of 3: 203 msec per loop

d:\Phd\Teaching\PyCUDA\programs>_
```

- Numpy 1100 ms, PyCUDA 200 ms
- That’s more than 5x for a very moderate effort

- (You just learned the map/reduce basics)

Writing your own kernel

- So far we used the built in PyCUDA kernels
- We will now do the elementwise kernel, manually
  1. Write the CUDA C-code
  2. Have PyCUDA compile it for us
  3. Decide a block and grid structure for the threads
  4. Copy data from Host to GPU
  5. Execute kernel
  6. Copy data back
  7. ??
  8. Profit

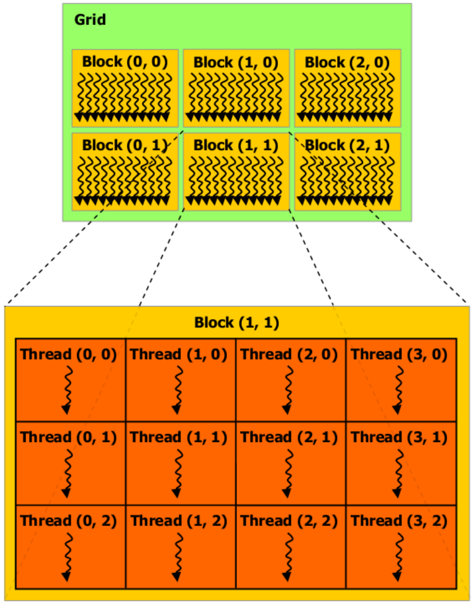
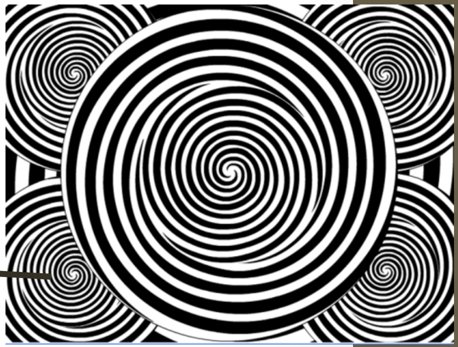


Figure 2-1. Grid of Thread Blocks

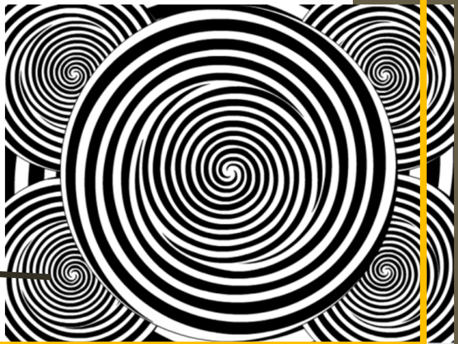
```
import pycuda.driver as cuda
import pycuda.gpuarray as gpuarray
import pycuda.autoinit
from pycuda.compiler import SourceModule
import numpy

kernels = SourceModule("""
__global__ void custom_kernel(float *g_y, float *g_x)
{
      const int i = blockDim.x * blockIdx.x + threadIdx.x;

      const float x = g_x[i];
      g_y[i] = cos(x)*exp(sin(x)-sqrt(x*x));
}
""")
custom_kernel = kernels.get_function("custom_kernel");

size = 5120000
block_size = 512 # design a 1d block and grid structure
grid_size = size/block_size
block = (block_size,1,1)
grid = (grid_size,1)

X = numpy.linspace(1,size,size).astype(numpy.float32)
X_gpu = gpuarray.to_gpu(X)
Y_gpu = gpuarray.empty_like(X_gpu) # 1. transfer to GPU
custom_kernel(Y_gpu, X_gpu, block=block, grid=grid) # 2. execute kernel
answer = gpuarray.sum(Y_gpu) # 3. use PyCUDA sum reduction
```



```python
import pycuda.driver as cuda
import pycuda.gpuarray as gpuarray
import pycuda.autoinit
from pycuda.compiler import SourceModule
import numpy
```

New stuff

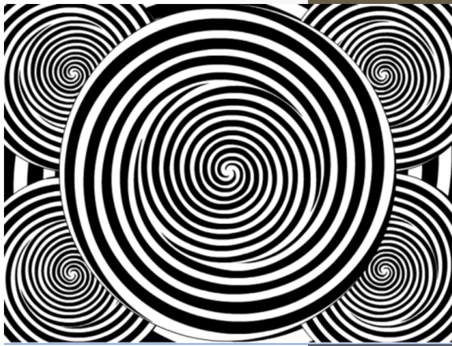
```python
kernels = SourceModule("""
__global__ void custom_kernel(float *g_y, float *g_x)
{
      const int i = blockDim.x * blockIdx.x + threadIdx.x;

      const float x = g_x[i];
      g_y[i] = cos(x)*exp(sin(x)-sqrt(x*x));
}
""")
custom_kernel = kernels.get_function("custom_kernel");

size = 5120000
block_size = 512 # design a 1d block and grid structure
grid_size = size/block_size
block = (block_size,1,1)
grid = (grid_size,1)
```



```python
X = numpy.linspace(1,size,size).astype(numpy.float32)
X_gpu = gpuarray.to_gpu(X)
Y_gpu = gpuarray.empty_like(X_gpu) # 1. transfer to GPU
custom_kernel(Y_gpu, X_gpu, block=block, grid=grid) # 2. execute kernel
answer = gpuarray.sum(Y_gpu) # 3. use PyCUDA sum reduction
```

Threads, Blocks and Grids

```
size = 5120000
block_size = 512 # design a 1d block and grid structure
grid_size = size/block_size
block = (block_size,1,1)
grid = (grid_size,1)

………

custom_kernel(Y_gpu, X_gpu, block=block, grid=grid)
```



- This is a non-trivial problem!
- Rule of thumb:
  - Choose number of threads per block
    - Multiples of 32
    - Hardware and kernel dependent! 128 or 256 is a good initial guess
  - Adjust grid size to match problem size
  - Time and test, adjust, time and test …

Custom kernel C code

```
kernels = SourceModule("""
__global__ void custom_kernel(float *g_y, float *g_x)
{
      const int i = blockDim.x * blockIdx.x + threadIdx.x;

      const float x = g_x[i];
      g_y[i] = cos(x)*exp(sin(x)-sqrt(x*x));
}
""")
custom_kernel = kernels.get_function("custom_kernel");

…

custom_kernel(Y_gpu, X_gpu, block=block, grid=grid)
```

- SourceModule compiles your C code for the GPU
  - It uses caching by default
- Arguments that are pointers in C are automatically converted when called with gpuarrays of the corresponding dtype

Custom

```
kernels = SourceM
__global__ void c
{
      const int i

      const float
      g_y[i] = co
}
""")
custom_kernel = k

…

custom_kernel(Y_g
```

- SourceModu
  - It uses cac
- Arguments t … nverted
  when called … e

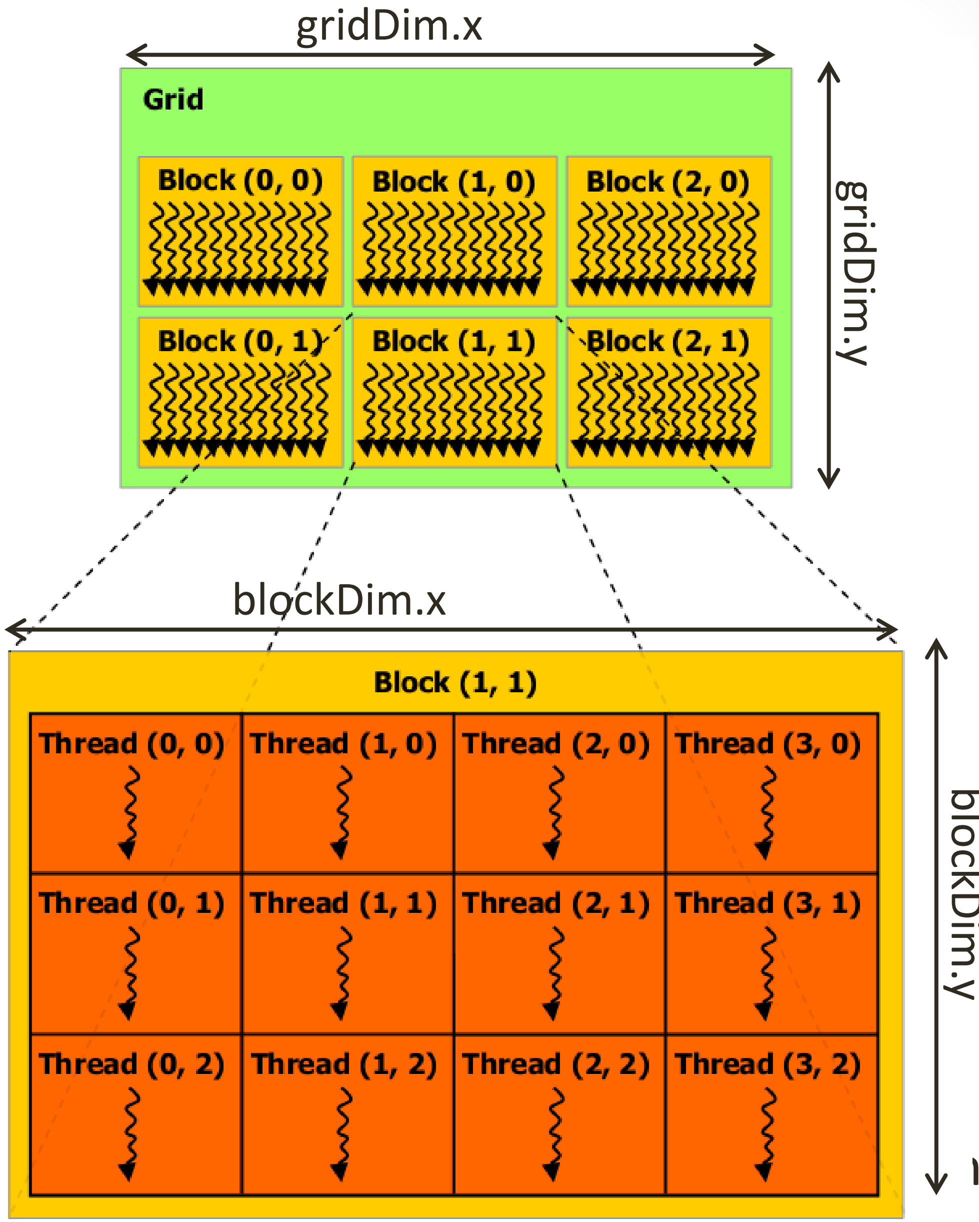
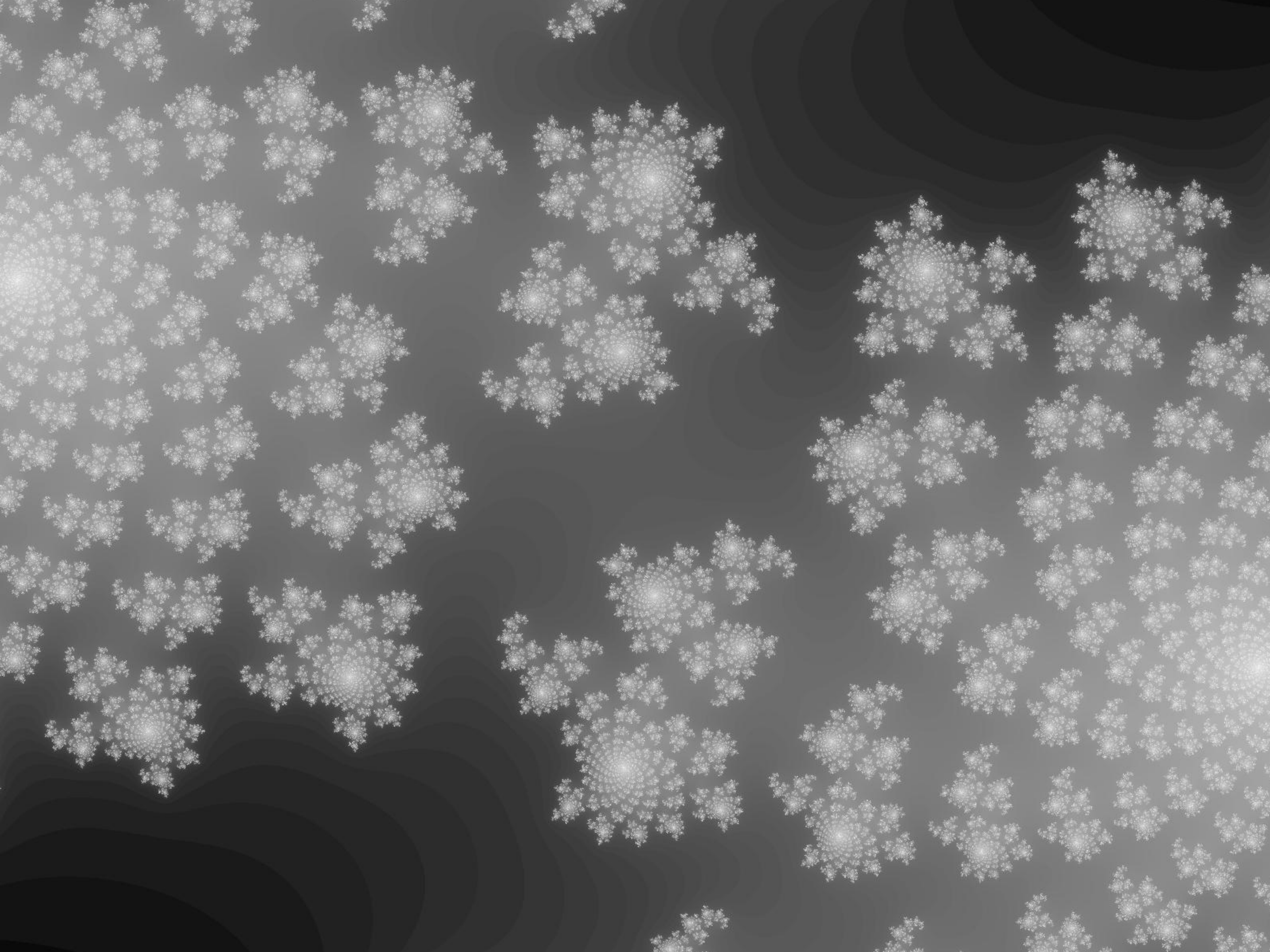
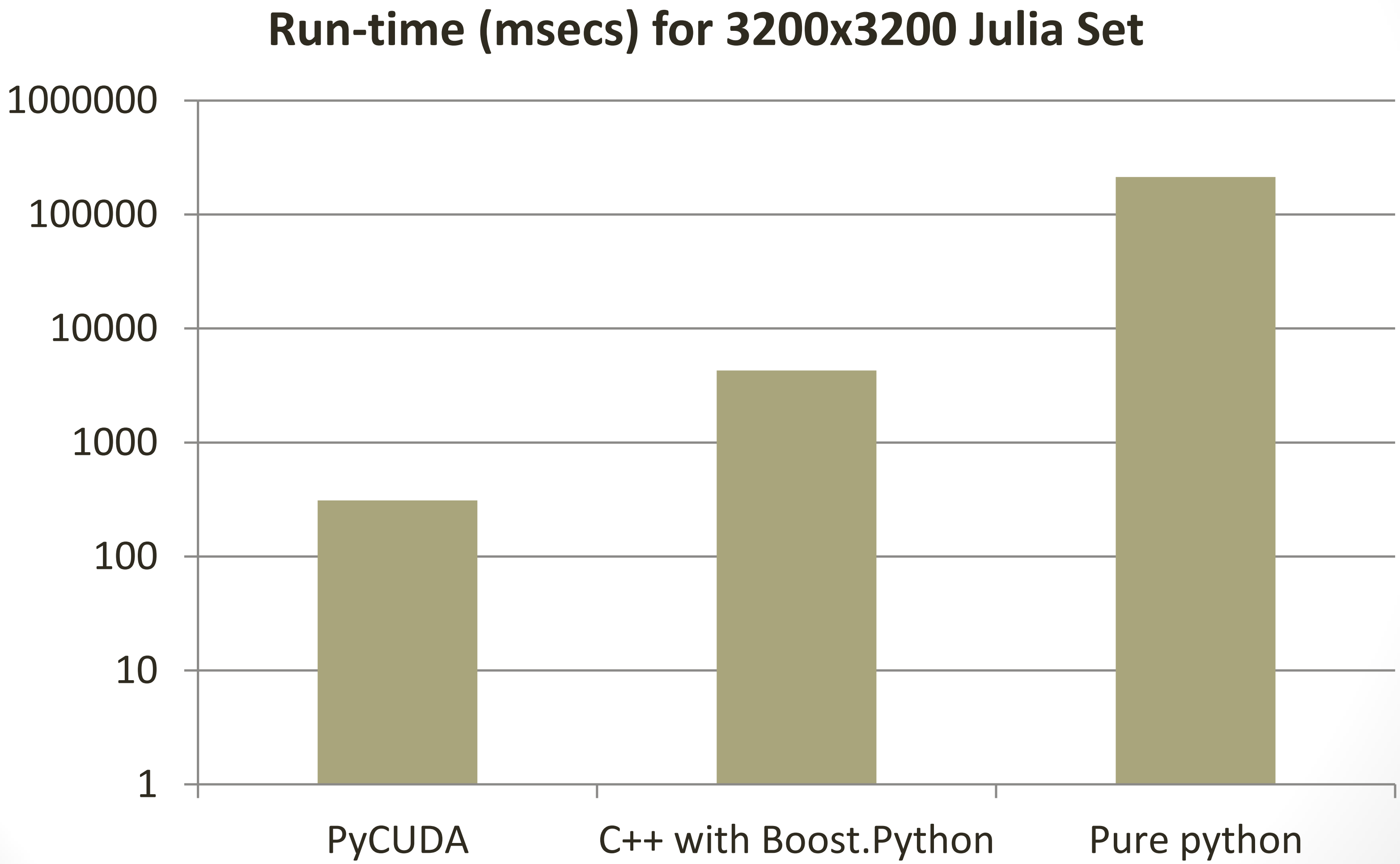


Figure 2-1. Grid of Thread Blocks

```python
import pycuda.driver as cuda
import pycuda.gpuarray as gpuarray
import pycuda.autoinit
from pycuda.compiler import SourceModule
import numpy

kernels = SourceModule("""
__global__ void custom_kernel(float *g_y, float *g_x)
{
      const int i = blockDim.x * blockIdx.x + threadIdx.x;

      const float x = g_x[i];
      g_y[i] = cos(x)*exp(sin(x)-sqrt(x*x));
}
""")
custom_kernel = kernels.get_function("custom_kernel");

size = 5120000
block_size = 512 # design a 1d block and grid structure
grid_size = size/block_size
block = (block_size,1,1)
grid = (grid_size,1)

X = numpy.linspace(1,size,size).astype(numpy.float32)
X_gpu = gpuarray.to_gpu(X)
Y_gpu = gpuarray.empty_like(X_gpu) # 1. transfer to GPU
custom_kernel(Y_gpu, X_gpu, block=block, grid=grid) # 2. execute kernel
answer = gpuarray.sum(Y_gpu) # 3. use PyCUDA sum reduction
```



Toy example: Julia Sets



What's next?

- Consider if you have data-parallel problems. If so, go home and hack something together just to get started.

- Don't reinvent wheels: libraries exist for common tasks
  - Random number generation, FFT, map/reduce templates

- Recommended reading
  - The official CUDA documentation is pretty good:
    - http://developer.nvidia.com/nvidia-gpu-computing-documentation
    - Most things in the CUDA C Programming Guide applies
  - Get on the mail list for PyCUDA
    - http://lists.tiker.net/listinfo/pycuda

Getting PyCUDA

- CUDA Toolkit (4.1 as of now)
  - http://developer.nvidia.com/cuda-toolkit-41
  - (I'm still on 4.0 http://developer.nvidia.com/cuda-toolkit-40)
- Get Python+numpy (I use Python 2.7)
  - http://python.org/download/
  - http://sourceforge.net/projects/numpy/files/NumPy/1.6.1/
  - http://sourceforge.net/projects/scipy/files/scipy/0.10.0/
  - http://sourceforge.net/projects/matplotlib/files/matplotlib/matplotlib-1.1.0/
- PyCUDA:
  - http://mathema.tician.de/software/pycuda
  - http://wiki.tiker.net/PyCuda/Installation

- Windows users: Check out Christoph Gohlke's repository of precompiled Python Libraries:
- http://www.lfd.uci.edu/~gohlke/pythonlibs/
  - Among other: PyCUDA, PyOpenCL, also h5py (HDF scientific data format) and more
  - The current version supports Toolkit 4.0, next will support 4.1.

That’s all

Hack the planet!

Extra slides

```
kernels = SourceModule("""

#define MAX_ITERATIONS 500
#define MAX_RANGE 50.0f

__global__ void julia(
const float cx, const float cy,
const float scale, const float x_pos, const float y_pos,
float *g_iterations)
{
const int pixel_x = blockIdx.x * blockDim.x + threadIdx.x;
const int pixel_y = blockIdx.y * blockDim.y + threadIdx.y;

const int pixel_width = blockDim.x*gridDim.x;
const int pixel_height = blockDim.y*gridDim.y;

const int i = pixel_y * pixel_width + pixel_x;

float x = scale*((float)pixel_x/(float)pixel_width - 0.5f) - x_pos;
float y = scale*((float)pixel_y/(float)pixel_height - 0.5f) - y_pos;

float x2 = x*x, y2=y*y;
int k;
for(k=1; k<=MAX_ITERATIONS; k++)
{
     // z = x + iy, z^2 = x^2-y^2 + i(2xy)
     y = 2*x*y + cy;
     x = x2 - y2 + cx;
     x2 =x*x;
     y2 =y*y;

     if (x2+y2 > MAX_RANGE) break;
}

g_iterations[i] = log((float)k);
}
""")
```

```python
block_size = 32
grid_size = 100

# 2d picture - map to 2d grid
grid = (grid_size,grid_size)
block = (block_size,block_size,1)
size = block_size * grid_size

# Allocate memory
iterations = numpy.zeros((size*size,), dtype=numpy.float32)
iterations_gpu = cuda.mem_alloc(iterations.nbytes)

# Run kernel
julia_c = -0.7 -0.3j
scale = 1.0;
x_pos = 0.0;
y_pos = 0.0;
kernel_julia(
                      numpy.float32(julia_c.real),
                      numpy.float32(julia_c.imag),
                      numpy.float32(scale),
                      numpy.float32(x_pos),
                      numpy.float32(y_pos),
                      iterations_gpu, block=block,grid=grid)

cuda.memcpy_dtoh(iterations, iterations_gpu)
iterations=iterations.reshape((size,size))

fig = plt.figure(figsize=(10,10))
ax = fig.add_subplot(111);
ax.imshow(iterations, cmap=colormaps.gray)
plt.show()
```