Programming GPUs with PyCuda

Nicolas Pinto (MIT) and Andreas Klöckner (Brown)

SciPy Conference 2009 / Advanced Tutorial
http://conference.scipy.org/advanced_tutorials

August 19, 2009

Thanks

- SciPy2009 Organizers
- **Andreas Klöckner (!)**
- PyCuda contributors
- Nvidia Corporation

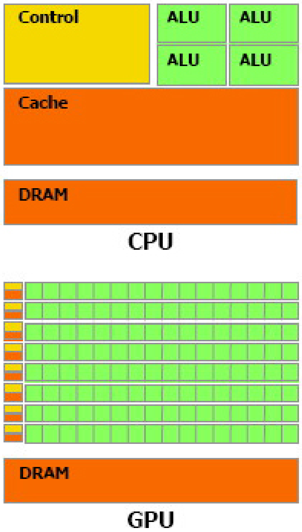
Outline

Outline

Outline

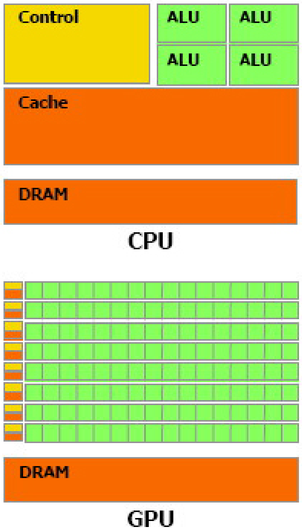
Stream Processing?

- Design target for CPUs:
  - Focus on *Task* parallelism
  - Make a single thread very fast
  - Hide latency through large caches
  - Predict, speculate

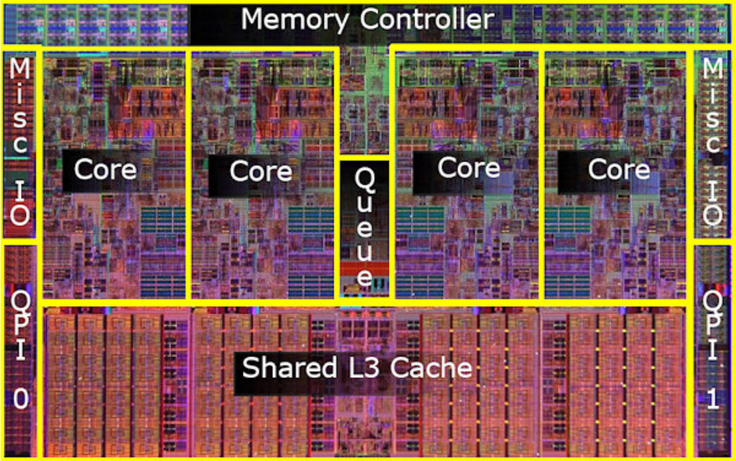


Stream Processing?

- Design target for CPUs:
  - Focus on *Task* parallelism
  - Make a single thread very fast
  - Hide latency through large caches
  - Predict, speculate
- *Stream* Processing takes a different approach:
  - Focus on *Data* parallelism
  - Throughput matters – single threads do not
  - Hide latency through parallelism
  - Let programmer deal with "raw" memory hierarchy

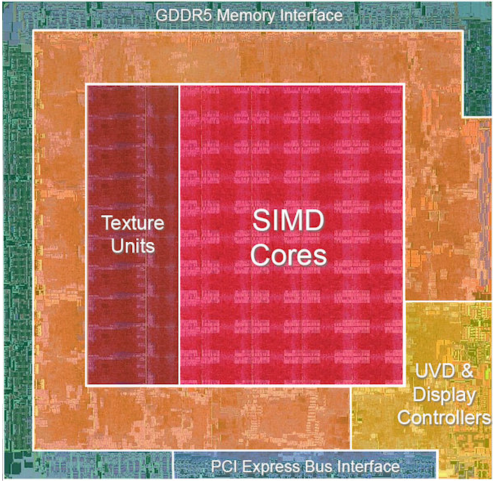


CPU Chip Real Estate



Die floorplan: Intel Core i7 (2008).
45 nm, 4x4 SP ops at a time, 4x256KB L2, 8MB L3

GPU Chip Real Estate



Die floorplan: AMD RV770 (2008).
55 nm, 800 SP ops at a time.

Market Overview

Quote Linus Torvalds:

> "Hardware that isn't mass market tends to not be worth it in the long run."

Market Overview

Quote Linus Torvalds:

> "Hardware that isn't mass market tends to not be worth it in the long run."

Based on that:

- Sony/Toshiba/IBM: Cell Broadband Engine
- ATI: R580 and later
- Nvidia: G80 and later
- Intel: Larabee

Outline

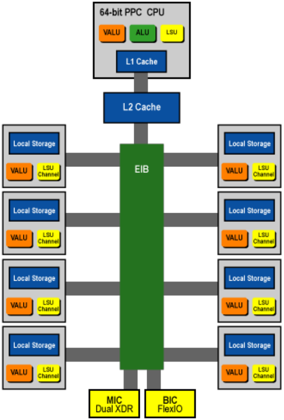
1 Introduction
- GPU Computing: Overview
- Architectures and Programming Models

2 Programming GPUs

3 GPU Scripting

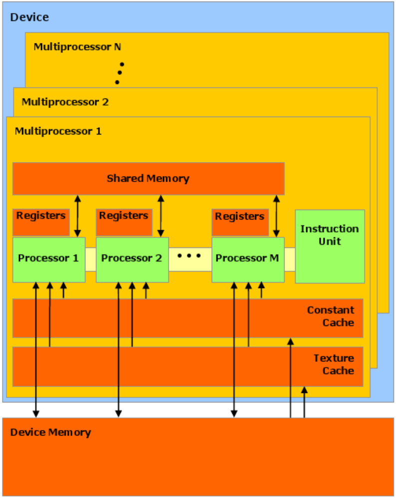
4 PyCuda Hands-on: Matrix Multiplication

Cell BE: Architecture



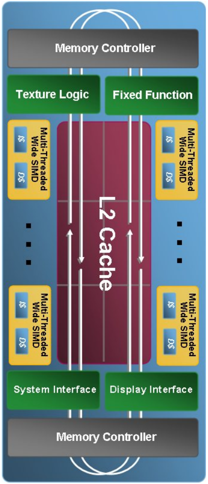
- 1 Cell BE = 1 dual-core Power + 8 SPEs + Bus
- 1 SPE = SPU + DMA + 256 KiB Local Store
- 1 SPU = 128-bit Vector ALU
- Bus = 200 GB/s Ring
- Ded. RAM (25 GB/s)

GPU: Architecture (e.g. Nvidia)



- 1 GPU = 30 MPs
- 1 MP = 1 ID (1/4 clock) + 8 SP + 1 DP + 16 KiB Shared + 32 KiB Reg + HW Sched
- Scalar cores max 512 threads/MP
- Ded. RAM (140 GB/s)
- PCIe2 Host DMA (6 GB/s)
- Limited Caches

Intel Larabee: Architecture



- Unreleased (2010?)
- x86-64 + SSE + "vector-complete" 512-bit ISA ("LRBni")
- 4x "Hyperthreading"
- 32 (?) cores per chip
- "Fiber/Strand" software threads
- Recursive Launches
- Coherent Caches (w/ explicit control)
- Performance?

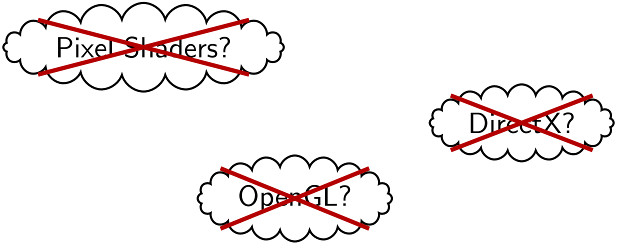
Programming Models

Pixel Shaders?

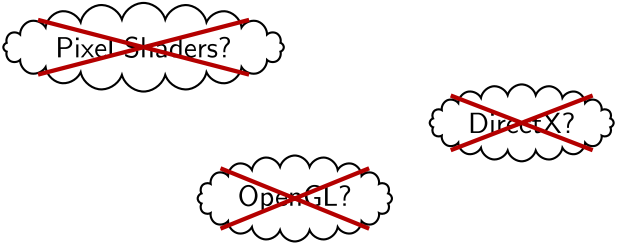
DirectX?

OpenGL?

Programming Models

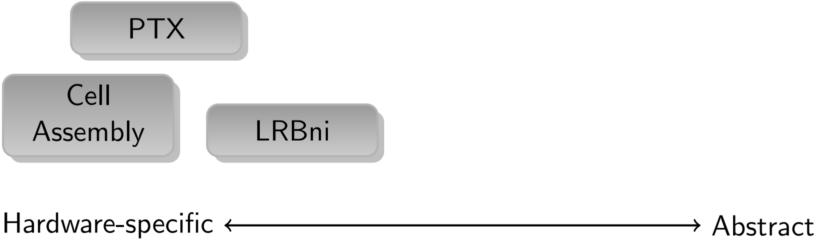


Programming Models

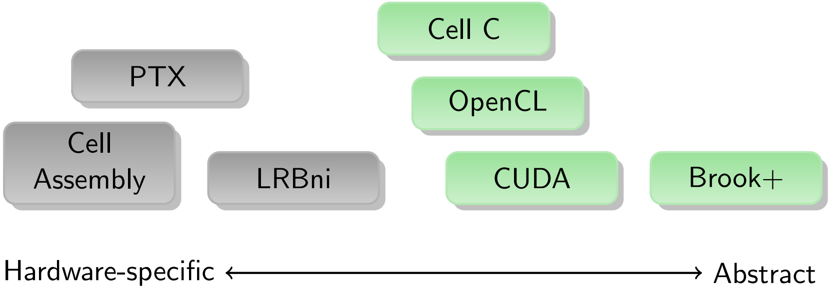


- Dedicated Compute APIs
- Not much “graphicsy” stuff visible

Programming Models



Programming Models



Architecture Comparison

Cell	GPU	Larabee
⊕ $\sim$ Multicore ⊕ Open Spec ⊖ Hard: DMA sched, Alignment, Small LS ⊖ HW Avail. ($) ⊖ Mem BW		

Architecture Comparison

Cell	GPU	Larabee
⊕ ~ Multicore ⊕ Open Spec ⊖ Hard: DMA sched, Alignment, Small LS ⊖ HW Avail. ($) ⊖ Mem BW	⊕ Available now ⊕ OpenCL ⊕ Maturing (tools etc.) ⊕ Mem BW ⊙ Reasonably easy ⊖ Perf. opaque	

Architecture Comparison

Cell	GPU	Larabee
+ ∼ Multicore + Open Spec − Hard: DMA sched, Alignment, Small LS − HW Avail. ($) − Mem BW	+ Available now + OpenCL + Maturing (tools etc.) + Mem BW o Reasonably easy − Perf. opaque	− Unavailable + Programmability? o OpenCL Support (→ why wait?) − Competitive with next-gen GPU? (→ why specialize?)

Architecture Comparison

Cell	GPU	Larabee
+ $\sim$ Multicore + Open Spec − Hard: DMA sched, Alignment, Small LS − HW Avail. ($) − Mem BW	+ Available now + OpenCL + Maturing (tools etc.) + Mem BW o Reasonably easy − Perf. opaque Focus Here	− Unavailable + Programmability? o OpenCL Support ($\rightarrow$ why wait?) − Competitive with next-gen GPU? ($\rightarrow$ why specialize?)

Questions?

?

Outline

Outline

What is CUDA?

- CUDA is Nvidia's proprietary compute abstraction.

What is CUDA?

- CUDA is Nvidia's proprietary compute abstraction.
- Main merit: A well-balanced model of GPU computing.
    - Abstract enough to not be hardware-specific.
    - Concrete enough to expose most hardware features.

What is CUDA?

- CUDA is Nvidia's proprietary compute abstraction.
- Main merit: A well-balanced model of GPU computing.
  - Abstract enough to not be hardware-specific.
  - Concrete enough to expose most hardware features.
- (Very) close semantic relative of OpenCL.

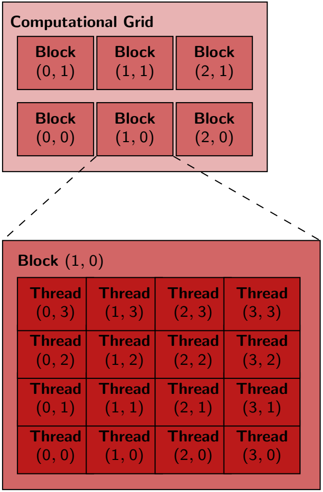
Gains and Losses

Gains	Losses
⊕ Memory Bandwidth (140 GB/s vs. 12 GB/s) ⊕ Compute Bandwidth (Peak: 1 TF/s vs. 50 GF/s, Real: 200 GF/s vs. 10 GF/s)	

Gains and Losses

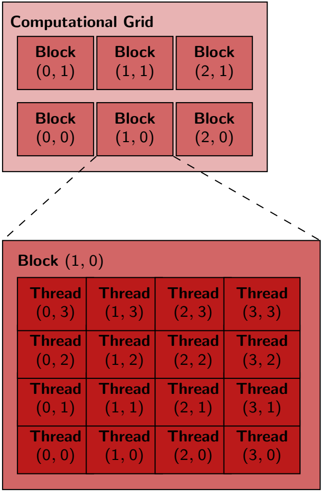
Gains	Losses
Memory Bandwidth (140 GB/s vs. 12 GB/s)	Recursion
Compute Bandwidth (Peak: 1 TF/s vs. 50 GF/s, Real: 200 GF/s vs. 10 GF/s)	Function pointers
	Exceptions
	IEEE 754 FP compliance
	Cheap branches (i.e. `ifs`)

GPUs: Threading Model



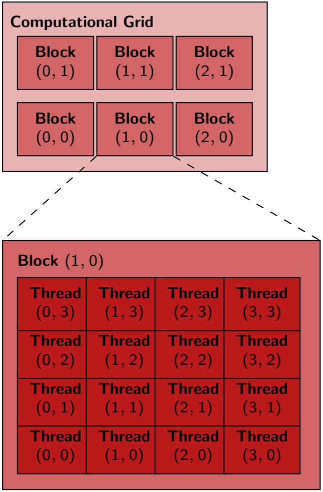
- Multi-tiered Parallelism
  - *Grid* of Blocks
  - *Block* of Threads

GPUs: Threading Model



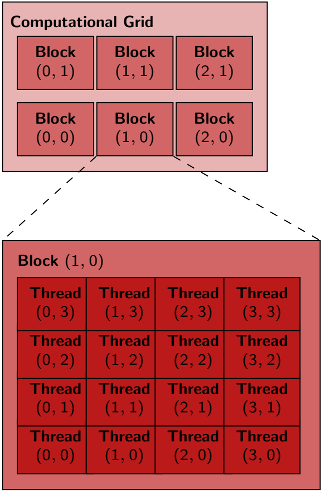
- Multi-tiered Parallelism
  - *Grid* of Blocks
  - *Block* of Threads
- Only threads within a block can communicate

GPUs: Threading Model



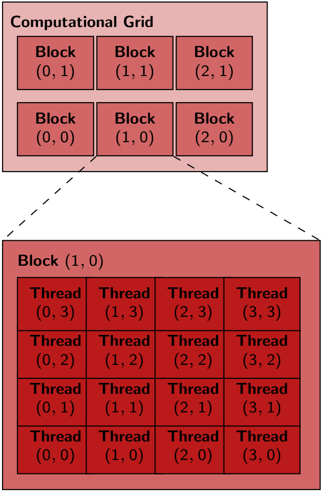
- Multi-tiered Parallelism
  - *Grid* of Blocks
  - *Block* of Threads
- Only threads within a block can communicate
  - Each Block is assigned to a physical execution unit.

GPUs: Threading Model



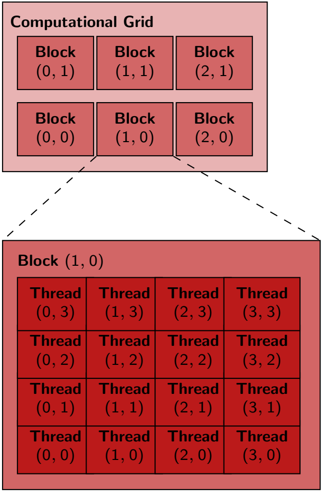
- Multi-tiered Parallelism
  - *Grid* of Blocks
  - *Block* of Threads
- Only threads within a block can communicate
  - Each Block is assigned to a physical execution unit.
- Algorithm must work with blocks executed in any order

GPUs: Threading Model



- Multi-tiered Parallelism
  - *Grid* of Blocks
  - *Block* of Threads
- Only threads within a block can communicate
  - Each Block is assigned to a physical execution unit.
- Algorithm must work with blocks executed in any order
- Grids and Blocks replace outer loops in an algorithm.

GPUs: Threading Model



- Multi-tiered Parallelism
  - *Grid* of Blocks
  - *Block* of Threads
- Only threads within a block can communicate
  - Each Block is assigned to a physical execution unit.
- Algorithm must work with blocks executed in any order
- Grids and Blocks replace outer loops in an algorithm.
- Indices available at run time

My first CUDA program

```
// GPU-side

__global__ void square_array(float *a, int n)
{
  int i = blockIdx.x * blockDim.x + threadIdx.x;
  if (i < n)
    a[i] = a[i] * a[i];
}
```

My first CUDA program

```
int main() // CPU-side
{
  cudaSetDevice(0); // EDIT ME

  int n = 4096; int bytes = n*sizeof(float);
  float *a_host = (float *) malloc(bytes);
  for (int i = 0; i < n; i++) a_host[i] = i;

  float *a_device;
  cudaMalloc((void **) &a_device, bytes);
  cudaMemcpy(a_device, a_host, bytes, cudaMemcpyHostToDevice);

  int block_size = 256;
  int n_blocks = (n + block_size-1) / block_size;
  square_array <<<n_blocks, block_size>>>(a_device, n);

  free(a_host); cudaFree(a_device);
}
```

A Dose of Reality: Error Checking

```
#define CUDA_CHK(NAME, ARGS) { \
  cudaError_t cuda_err_code = NAME ARGS; \
  if (cuda_err_code != cudaSuccess) { \
    printf ("%s failed with code %d\n", #NAME, cuda_err_code); \
    abort (); \
  } \
}

CUDA_CHK(cudaMalloc, (&result, m_size*sizeof(float)));
```

Typical errors:

- GPUs have (some) memory protection → "Launch failure"
- Invalid sizes (block/grid/...)

Invisible Subtleties

Host Pointer or Device Pointer?

```
float *h_data = (float*) malloc(mem_size);
float *d_data;
CUDA_CHK(cudaMalloc, ((void**) &d_data, mem_size));
```

→ *Both kinds of pointer share the same data type!*

Invisible Subtleties

Host Pointer or Device Pointer?

```
float *h_data = (float*) malloc(mem_size);
float *d_data;
CUDA_CHK(cudaMalloc, ((void**) &d_data, mem_size));
```

→ *Both kinds of pointer share the same data type!*

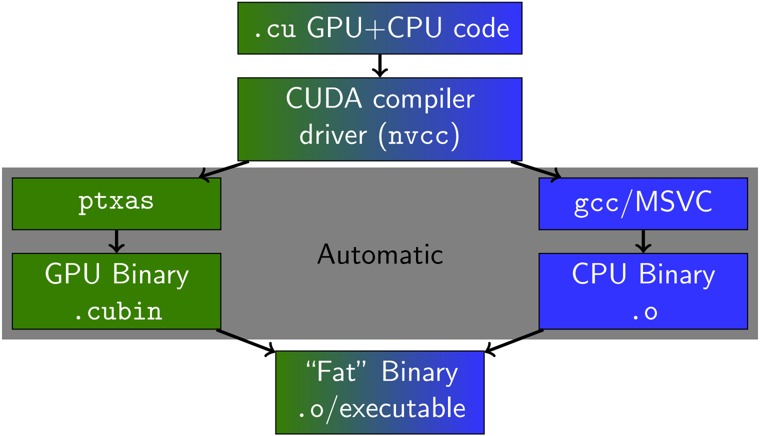
Kernel Launches

Execution configuration:

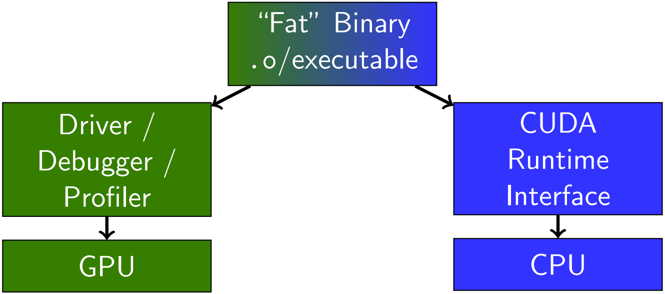
```
dim3 grid_size(gx, gy); // max 2D
dim3 block_size(bx, by, bz); // max 3D
kernel <<<grid_size, block_size>>>(arg, ...);
```

- Do not wait for completion.
- Cheap! ($\sim 2\ \mu s$ overhead)

The CUDA Toolchain



Executing CUDA Binaries



GPU Demo Machines

Machine	GPUs
iapcuda-01	Device 0: "GeForce GTX 285"
iapcuda-01	Device 1: "Tesla C1060"
iapcuda-01	Device 2: "Tesla C1060"
iapcuda-02	Device 0: "GeForce GTX 295"
iapcuda-02	Device 1: "GeForce GTX 295"
iapcuda-02	Device 2: "Tesla C1060"
iapcuda-02	Device 3: "Tesla C1060"

Prepare your workspace in one of our CUDA demo machine:

1. `ssh scipy09@iapcuda-NN.no-ip.org` (`password: GpUh4cK3r`)

GPU Demo Machines

Machine	GPUs
iapcuda-01	Device 0: "GeForce GTX 285"
iapcuda-01	Device 1: "Tesla C1060"
iapcuda-01	Device 2: "Tesla C1060"
iapcuda-02	Device 0: "GeForce GTX 295"
iapcuda-02	Device 1: "GeForce GTX 295"
iapcuda-02	Device 2: "Tesla C1060"
iapcuda-02	Device 3: "Tesla C1060"

Prepare your workspace in one of our CUDA demo machine:

1. `ssh scipy09@iapcuda-NN.no-ip.org` (`password: GpUh4cK3r`)
2. `mkdir` *`lastname.firstname`*

GPU Demo Machines

Machine	GPUs
iapcuda-01	Device 0: "GeForce GTX 285"
iapcuda-01	Device 1: "Tesla C1060"
iapcuda-01	Device 2: "Tesla C1060"
iapcuda-02	Device 0: "GeForce GTX 295"
iapcuda-02	Device 1: "GeForce GTX 295"
iapcuda-02	Device 2: "Tesla C1060"
iapcuda-02	Device 3: "Tesla C1060"

Prepare your workspace in one of our CUDA demo machine:

1. `ssh scipy09@iapcuda-NN.no-ip.org` (`password: GpUh4cK3r`)
2. `mkdir` *`lastname.firstname`*
3. `cd` *`lastname.firstname`*

GPU Demo Machines

Machine	GPUs
iapcuda-01	Device 0: "GeForce GTX 285"
iapcuda-01	Device 1: "Tesla C1060"
iapcuda-01	Device 2: "Tesla C1060"
iapcuda-02	Device 0: "GeForce GTX 295"
iapcuda-02	Device 1: "GeForce GTX 295"
iapcuda-02	Device 2: "Tesla C1060"
iapcuda-02	Device 3: "Tesla C1060"

Prepare your workspace in one of our CUDA demo machine:

1. `ssh scipy09@iapcuda-NN.no-ip.org` (`password: GpUh4cK3r`)
2. `mkdir` *`lastname.firstname`*
3. `cd` *`lastname.firstname`*
4. `wget is.gd/2o40o` && tar xzf scipy09-pycuda-tut.tar.gz

Getting your feet wet

Hands-on Exercise

1. Edit `1-cuda-simple/simple.cu`:
   `cudaSetDevice(`*`Your GPU #`*`);`
2. Compile and run:
   ```
   nvcc -o simple.x simple.cu
   ./simple.x
   ```
3. Add error checking to the example.
4. Modify `simple.cu` to print the contents of the result.
5. Modify `simple.cu` to compute $c_i = a_i b_i$.
6. Modify `simple.cu` to use blocks of 16×16 threads.

My first CUDA program (Solution)

Preliminary bits:

```
#include <stdio.h>

#define CUDA_CHK(NAME, ARGS) { \
  cudaError_t cuda_err_code = NAME ARGS; \
  if (cuda_err_code != cudaSuccess) { \
    printf ("%s failed with code %d\n", #NAME, cuda_err_code); \
    abort (); \
  } \
}
```

My first CUDA program (Solution)

The GPU kernel:

```
__global__ void square_array(float *a, float *b, int n)
{
  int i = (blockIdx.x * blockDim.y + threadIdx.y)
    * blockDim.x + threadIdx.x;
  if (i < n)
    a[i] = a[i] * b[i];
}
```

My first CUDA program (Solution)

Allocating memory:

```
int main()
{
  cudaSetDevice(0); // EDIT ME

  const int n = 4096;

  float *a_host = (float *) malloc(n*sizeof(float));
  float *b_host = (float *) malloc(n*sizeof(float));

  float *a_device, *b_device;
  CUDA_CHK(cudaMalloc, ((void **) &a_device, n*sizeof(float)));
  CUDA_CHK(cudaMalloc, ((void **) &b_device, n*sizeof(float)));
```

My first CUDA program (Solution)

Transfer and Launch:

```
for (int i = 0; i < n; i++) { a_host[i] = i; b_host[i] = i+1; }

CUDA_CHK(cudaMemcpy, (a_device, a_host, n*sizeof(float),
        cudaMemcpyHostToDevice));
CUDA_CHK(cudaMemcpy, (b_device, b_host, n*sizeof(float),
        cudaMemcpyHostToDevice));

dim3 block_dim(16, 16);
int block_size = block_dim.x*block_dim.y;
int n_blocks = (n + block_size-1) / block_size;
square_array<<<n_blocks, block_dim>>>(a_device, b_device, n);
```

My first CUDA program (Solution)

Output and Clean-up:

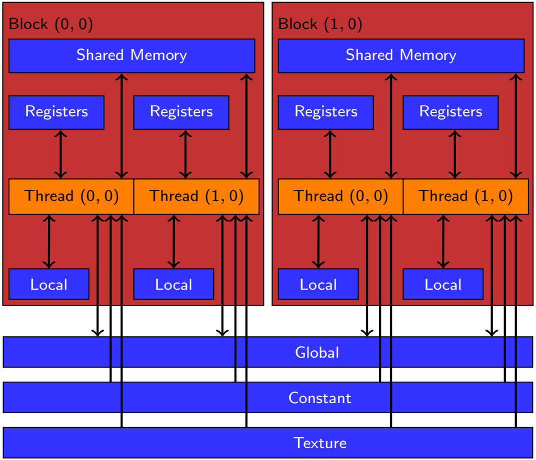
```
  CUDA_CHK(cudaMemcpy, (a_host, a_device, n*sizeof(float),
        cudaMemcpyDeviceToHost));

  for (int i = 0; i < n; i++)
    printf("%.0f ", a_host[i]);
  puts("\n");

  free(a_host);
  CUDA_CHK(cudaFree, (a_device));
}
```

Questions?

?

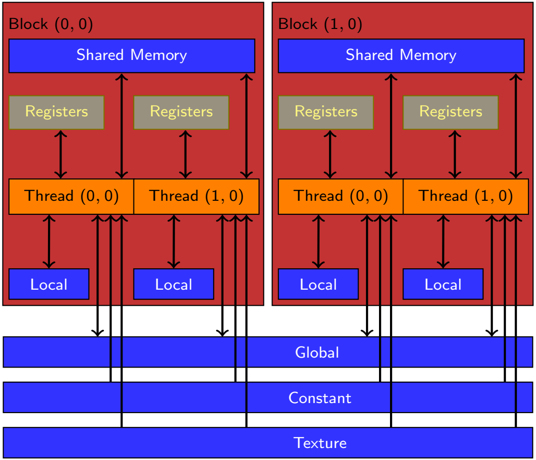
Outline

Memory Model



Already seen:

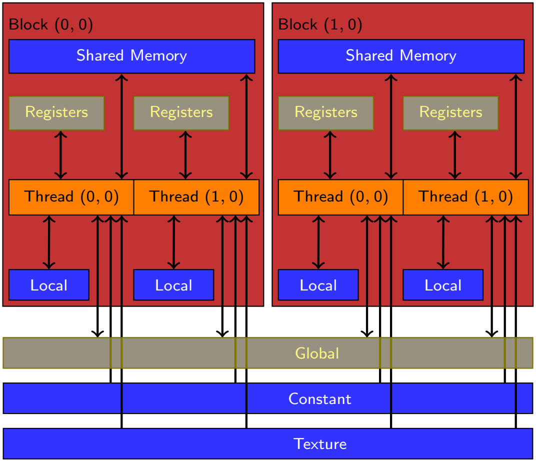
Memory Model



Already seen:

- Registers

Memory Model

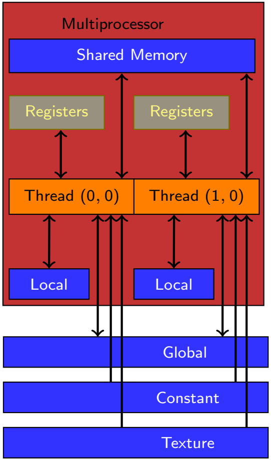


Already seen:

- Registers
- Global

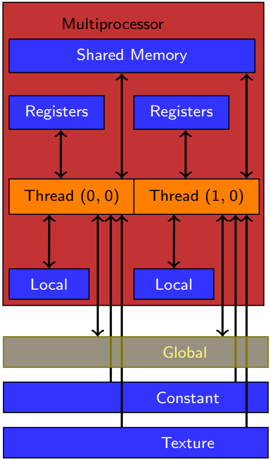
Registers

- 32 KiB of registers per MP
- Per-thread
- Latency: 1 clock
- Variable amount per thread
  - Register count limits max. threads/MP
  - CPUs: Fixed register file ($\sim$)

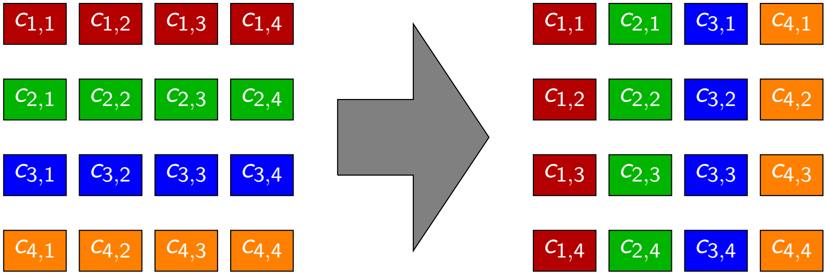


Global Memory

- Several GiB usually
- Per-GPU
- Latency: ~1000 clocks
- 512 bit memory bus
    - Best throughput: 16 consecutive threads read aligned chunk



Example: Matrix Transpose



Naive: Using global memory

First attempt: Naive port of the CPU code.

```
__global__ void transpose(float *out, float *in, int w, int h) {
  unsigned int xIdx = blockDim.x * blockIdx.x + threadIdx.x;
  unsigned int yIdx = blockDim.y * blockIdx.y + threadIdx.y;

  if ( xIdx < w && yIdx < h ) {
    unsigned int idx_in  = xIdx + w * yIdx;
    unsigned int idx_out = yIdx + h * xIdx;

    out[idx_out] = in[idx_in];
  }
}
```

Measuring Performance

Writing high-performance Codes

Mindset: What is going to be the limiting factor?

- Floating point throughput?
- Memory bandwidth?
  - Cache sizes?

Measuring Performance

Writing high-performance Codes

Mindset: What is going to be the limiting factor?

- Floating point throughput?
- Memory bandwidth?
  - Cache sizes?

Benchmark the assumed limiting factor right away.

Measuring Performance

Writing high-performance Codes

Mindset: What is going to be the limiting factor?

- Floating point throughput?
- Memory bandwidth?
  - Cache sizes?

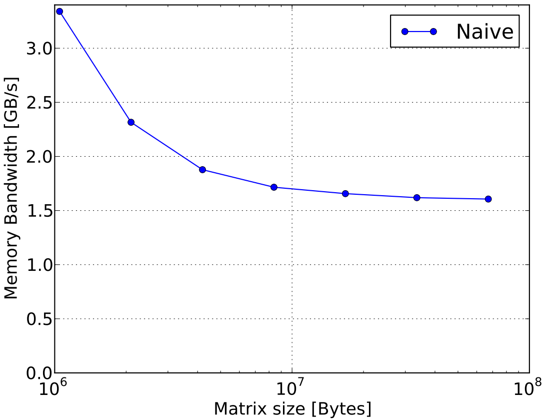
Benchmark the assumed limiting factor right away.

Evaluate

- Know your peak throughputs (roughly)
- Are you getting close?
- Are you tracking the right limiting factor?

Performance: Matrix transpose

Very likely: Bound by memory bandwidth.



Fantastic! About same as CPU. Why?

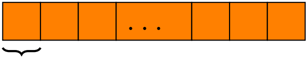
Naive: Using global memory

```
__global__ void transpose(float *out, float *in, int w, int h) {
  unsigned int xIdx = blockDim.x * blockIdx.x + threadIdx.x;
  unsigned int yIdx = blockDim.y * blockIdx.y + threadIdx.y;

  if ( xIdx < w && yIdx < h ) {
    unsigned int idx_in  = xIdx + w * yIdx;
    unsigned int idx_out = yIdx + h * xIdx;

    out[idx_out] = in[idx_in];
  }
}
```

Naive: Using global memory

```
__global__ void transpose(float *out, float *in, int w, int h) {
  unsigned int xIdx = blockDim.x * blockIdx.x + threadIdx.x;
  unsigned int yIdx = blockDim.y * blockIdx.y + threadIdx.y;

  if ( xIdx < w && yIdx < h ) {
    unsigned int idx_in = xIdx + w * yIdx;
    unsigned int idx_out = yIdx + h * xIdx;

    out[idx_out] = in[idx_in];
  }
}
```

Reading from global mem:

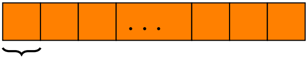


stride: 1

Naive: Using global memory

```
__global__ void transpose(float *out, float *in, int w, int h) {
  unsigned int xIdx = blockDim.x * blockIdx.x + threadIdx.x;
  unsigned int yIdx = blockDim.y * blockIdx.y + threadIdx.y;

  if ( xIdx < w && yIdx < h ) {
    unsigned int idx_in  = xIdx + w * yIdx;
    unsigned int idx_out = yIdx + h * xIdx;

    out[idx_out] = in[idx_in];
  }
}
```

Reading from global mem:

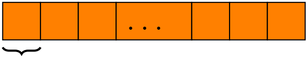


stride: 1 → one mem.trans.

Naive: Using global memory

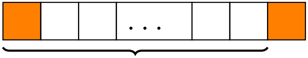
```
__global__ void transpose(float *out, float *in, int w, int h) {
  unsigned int xIdx = blockDim.x * blockIdx.x + threadIdx.x;
  unsigned int yIdx = blockDim.y * blockIdx.y + threadIdx.y;

  if ( xIdx < w && yIdx < h ) {
    unsigned int idx_in  = xIdx + w * yIdx;
    unsigned int idx_out = yIdx + h * xIdx;

    out[idx_out] = in[idx_in];
  }
}
```

Reading from global mem:



stride: 1 → one mem.trans.

Writing to global mem:

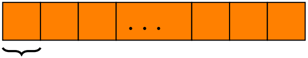


stride: 16

Naive: Using global memory

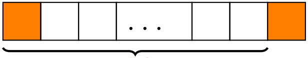
```
__global__ void transpose(float *out, float *in, int w, int h) {
  unsigned int xIdx = blockDim.x * blockIdx.x + threadIdx.x;
  unsigned int yIdx = blockDim.y * blockIdx.y + threadIdx.y;

  if ( xIdx < w && yIdx < h ) {
    unsigned int idx_in  = xIdx + w * yIdx;
    unsigned int idx_out = yIdx + h * xIdx;

    out[idx_out] = in[idx_in];
  }
}
```

Reading from global mem:



stride: 1 → one mem.trans.

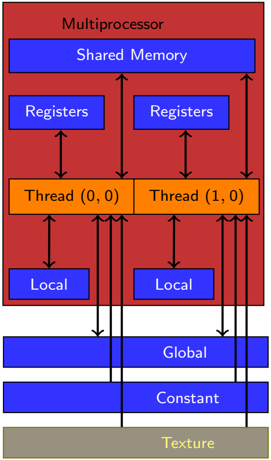
Writing to global mem:



stride: 16 → **16 mem.trans.!**

Texture Memory

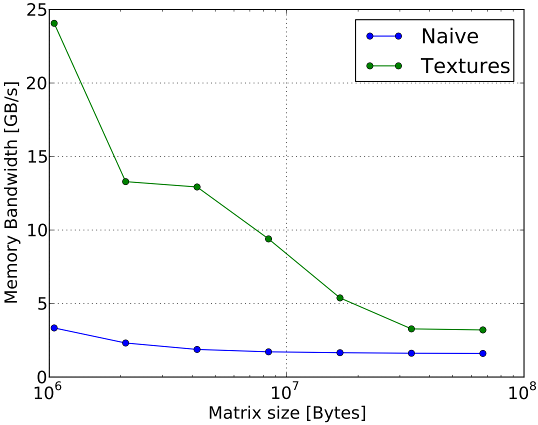
- Same memory as global
- But: more access patterns achieve usable bandwidth
- Optional: 2D and 3D indexing
- Small, incoherent Cache (prefers *n*D-local access)
- Read-only
- Latency: ~1000 clocks (despite cache!)
- Optional: Linear Interpolation



Transpose with Textures

```
texture <float, 1, cudaReadModeElementType> in_tex;

__global__ void transpose(float *out, int w, int h) {
  unsigned int yIdx = blockDim.x * blockIdx.x + threadIdx.x;
  unsigned int xIdx = blockDim.y * blockIdx.y + threadIdx.y;

  if ( xIdx < w && yIdx < h ) {
    unsigned int idx_in  = xIdx + w * yIdx;
    unsigned int idx_out = yIdx + h * xIdx;

    out[idx_out] = tex1Dfetch(in_tex, idx_in);
  }
}

#define PREPARE \
  cudaBindTexture(0, in_tex, d_idata, mem_size); \
  std::swap(grid.x, grid.y); \
  std::swap(threads.x, threads.y);
```

Performance: Transpose with Textures

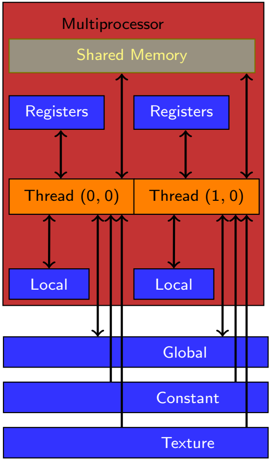


Better! But texture units can't quite hide wide data bus.

Need different idea.

Shared Memory

- 16 KiB of shared mem per MP
- Per-block
- Latency: 2 clocks
- Variable amount per block
  - Shared memory limits max. blocks/MP
- Banked



Transpose: Idea

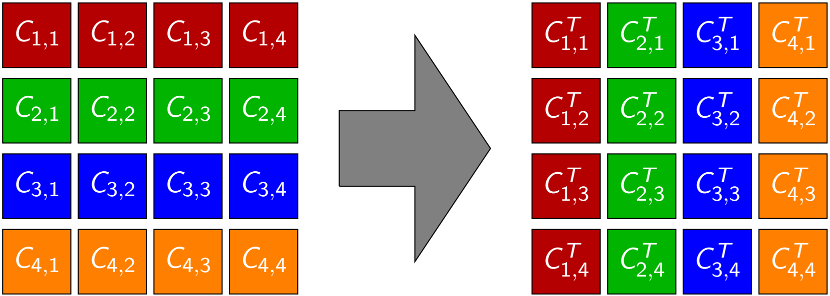
- Global memory dislikes non-unit strides.
- Shared memory doesn't mind.

Idea

- Don't transpose element-by-element.
- Transpose block-by-block instead.

1. Read untransposed block from global and write to shared
2. Read block transposed from shared and write to global

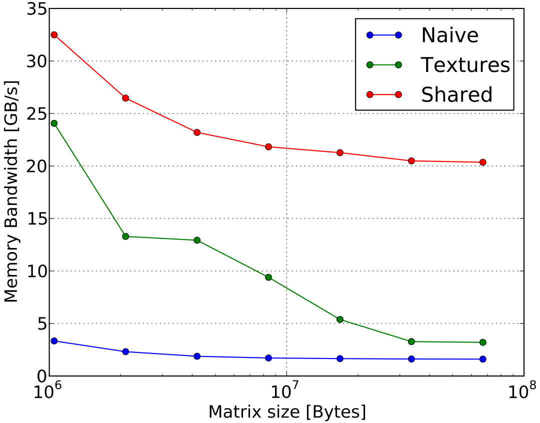
Illustration: Blockwise Transpose



Improved: Using shared memory

```
__global__ void transpose( float *out, float *in, int w, int h ) {
  __shared__ float block[BLOCK_DIM*BLOCK_DIM];

  unsigned int xBlock = blockDim.x * blockIdx.x;
  unsigned int yBlock = blockDim.y * blockIdx.y;

  unsigned int xIndex = xBlock + threadIdx.x;
  unsigned int yIndex = yBlock + threadIdx.y;

  unsigned int index_out, index_transpose;

  if ( xIndex < w && yIndex < h ) {
    unsigned int index_in = w * yIndex + xIndex;
    unsigned int index_block = threadIdx.y * BLOCK_DIM + threadIdx.x;

    block[index_block] = in[index_in];
    index_transpose = threadIdx.x * BLOCK_DIM + threadIdx.y;
    index_out = h * (xBlock + threadIdx.y) + yBlock + threadIdx.x;
  }
  __syncthreads();

  if ( xIndex < w && yIndex < h ) {
    out[index_out] = block[index_transpose];
  }
}
```

Performance: Transpose with Shared Memory



Not bad! Are we done?

Review: Memory Model

Type	Per	Access	Latency	
Registers	thread	R/W	1	
Local	thread	R/W	1000	
Shared	block	R/W	2	
Global	grid	R/W	1000	Not cached
Constant	grid	R/O	1-1000	Cached
Texture	grid	R/O	1000	Spatially cached

Important

Don't "*choose one*" type of memory.
Successful algorithms combine many types' strengths.

Questions?

?

Outline

1 Introduction

2 Programming GPUs

3 GPU Scripting
- Scripting + GPUs: A good combination
- Whetting your Appetite
- Working with PyCuda
- A peek under the hood
- Metaprogramming CUDA

4 PyCuda Hands-on: Matrix Multiplication

Outline

Scripting Languages

Python:

- is discoverable and interactive.
- has comprehensive built-in functionality.
- manages resources automatically.
- uses run-time typing.
- works well for "gluing" lower-level blocks together.

Scripting: Goals

Scripting languages aim to reduce the load on the programmer:

- Reduce required knowledge
- Encourage experimentation
- Eliminate sources of error
- Encourage abstraction wherever possible
- Value programmer time over computer time

Think about the tools you use.

Use the right tool for the job.

Our mantra: always use the right tool !

Why do Scripting for GPUs?

- GPUs are everything that scripting languages are not.
  - Highly parallel
  - Very architecture-sensitive
  - Built for maximum compute/memory throughput

  $\rightarrow$ complement each other
- CPU: largely restricted to control tasks (~1000/sec)
  - Scripting fast enough
- Realize a promise: Use Scripting...
  - from first prototype
  - to full-scale production code.

Scripting: Speed

- Usual answer to the "Speed Question": Hybrid ("mixed") Code.
- Plays to the strengths of each language.
- But: Introduces (some) complexity.

Observation: GPU code is already hybrid.

Consequence: No added complexity through hybrid code.

Questions?

?

Outline

Whetting your appetite

```python
import pycuda.driver as cuda
import pycuda.autoinit
import numpy

a = numpy.random.randn(4,4).astype(numpy.float32)
a_gpu = cuda.mem_alloc(a.nbytes)
cuda.memcpy_htod(a_gpu, a)
```

[This is `examples/demo.py` in the PyCuda distribution.]

Whetting your appetite

```
mod = cuda.SourceModule("""
    __global__ void doublify(float *a)
    {
      int idx = threadIdx.x + threadIdx.y*4;
      a[idx] *= 2;
    }
    """)

func = mod.get_function("doublify")
func(a_gpu, block=(4,4,1))

a_doubled = numpy.empty_like(a)
cuda.memcpy_dtoh(a_doubled, a_gpu)
print a_doubled
print a
```

Compute kernel

Whetting your appetite, Part II

Did somebody say "Abstraction is good"?

Whetting your appetite, Part II

```python
import numpy
import pycuda.autoinit
from pycuda import gpuarray

a_cpu = numpy.random.randn(4,4).astype(numpy.float32)
b_cpu = numpy.random.randn(4,4).astype(numpy.float32)
c_cpu = a_cpu * b_cpu

a_gpu = gpuarray.to_gpu(a_cpu)
b_gpu = gpuarray.to_gpu(b_cpu)
c_gpu = (a_gpu * b_gpu).get()

print c_cpu - c_gpu
```

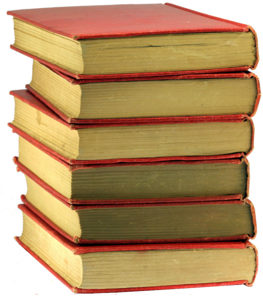
Remember me?

```
// trivia
#include <stdio.h>

#define CUDA_CHK(NAME, ARGS) { \
  cudaError_t cuda_err_code = NAME ARGS; \
  if (cuda_err_code != cudaSuccess) { \
    printf ("%s failed with code %d\n", #NAME, cuda_err_code); \
    abort (); \
  } \
}
// end

// kernel
__global__ void square_array ( float *a, float *b, int n)
{
  int i = (blockIdx.x * blockDim.y + threadIdx.y)
    * blockDim.x + threadIdx.x;
  if (i < n)
    a[i] = a[i] * b[i];
}
// end

// main1
int main()
{
  cudaSetDevice(0); // EDIT ME

  const int n = 4096;

  float *a_host = (float *) malloc(n*sizeof( float ));
  float *b_host = (float *) malloc(n*sizeof( float ));

  float *a_device, *b_device;
  CUDA_CHK(cudaMalloc, ((void **) &a_device, n*sizeof(float)));
  CUDA_CHK(cudaMalloc, ((void **) &b_device, n*sizeof(float)));
// end
```

```
// main2
  for (int i = 0; i < n; i++) { a_host[i] = i; b_host[i] = i+1; }

  CUDA_CHK(cudaMemcpy, (a_device, a_host, n*sizeof(float),
        cudaMemcpyHostToDevice));
  CUDA_CHK(cudaMemcpy, (b_device, b_host, n*sizeof(float),
        cudaMemcpyHostToDevice));

  dim3 block_dim(16, 16);
  int block_size = block_dim.x*block_dim.y;
  int n_blocks = (n + block_size-1) / block_size;
  square_array <<<n_blocks, block_dim>>>(a_device, b_device, n);
// end

// main3
  CUDA_CHK(cudaMemcpy, (a_host, a_device, n*sizeof(float),
        cudaMemcpyDeviceToHost));

  for (int i = 0; i < n; i++)
    printf ("%.0f ", a_host[i]);
  puts("\n");

  free (a_host);
  CUDA_CHK(cudaFree, (a_device));
}
// end
```

Outline

PyCuda Philosophy



- Provide complete access
- Automatically manage resources
- Provide abstractions
- Allow interactive use
- Check for and report errors automatically
- Integrate tightly with `numpy`

PyCuda: Completeness

PyCuda exposes *all* of CUDA.

For example:

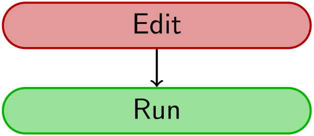
- Arrays and Textures
- Pagelocked host memory
- Memory transfers (asynchronous, structured)
- Streams and Events
- Device queries
- (GL Interop)

PyCuda: Completeness

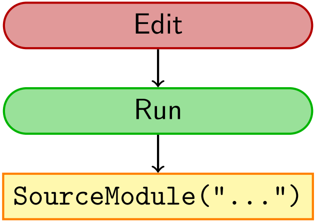
PyCuda supports every OS that CUDA supports.

- Linux
- Windows
- OS X

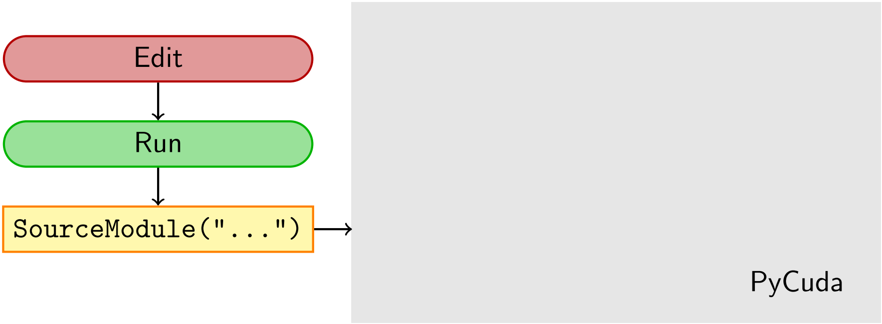
PyCuda: Workflow



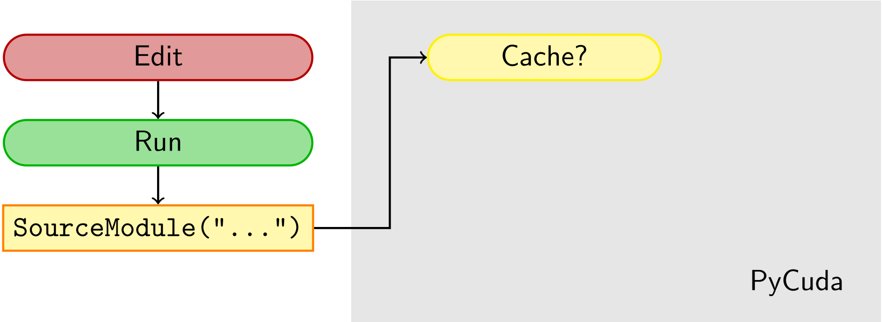
PyCuda: Workflow



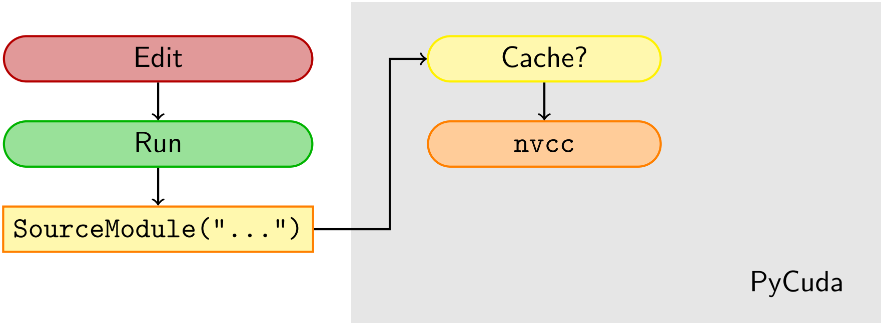
PyCuda: Workflow



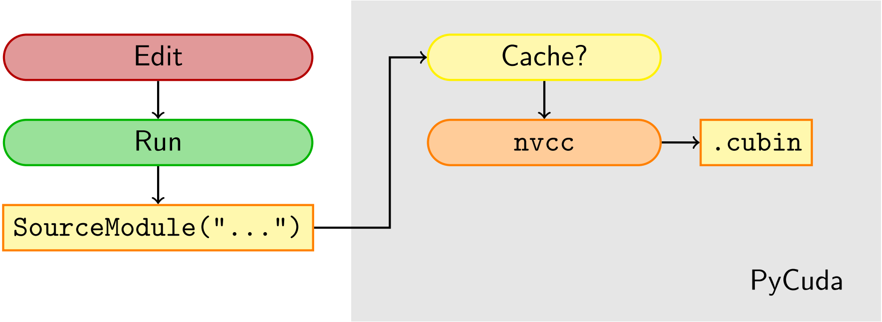
PyCuda: Workflow



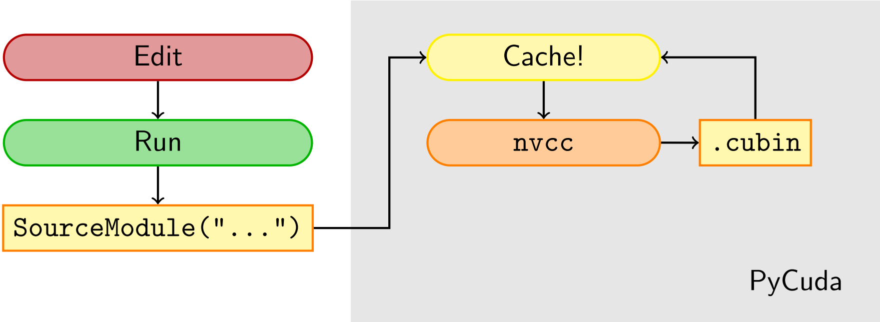
PyCuda: Workflow



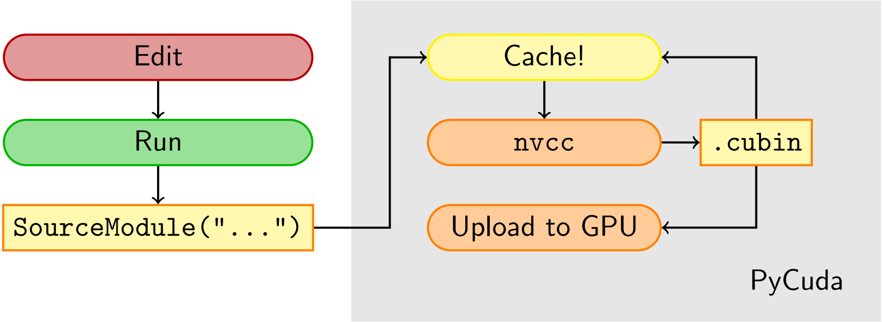
PyCuda: Workflow



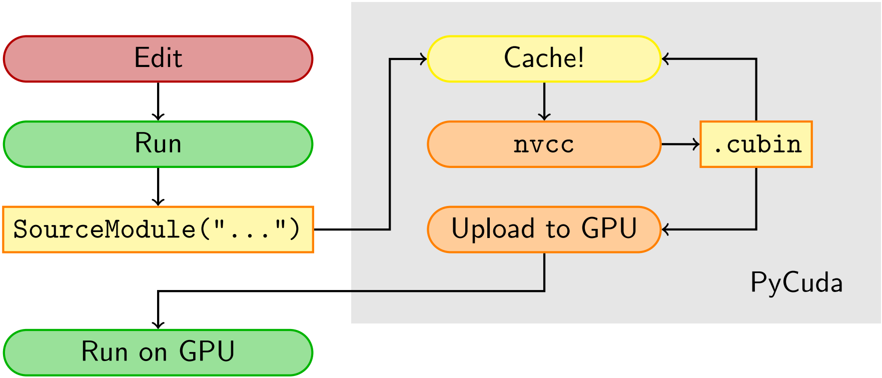
PyCuda: Workflow



PyCuda: Workflow



PyCuda: Workflow



Kernel Invocation: Automatic Copies

```
mod = pycuda.driver.SourceModule(
    "__global__ my_func(float *out, float *in){...}")
func = mod.get_function("my_func")

src = numpy.random.randn(400).astype(numpy.float32)
dest = numpy.empty_like(src)
```

```
my_func(
        cuda.Out(dest),
        cuda.In(src),
        block=(400,1,1))
```

- "InOut" exists, too.
- Only for immediate invocation style.

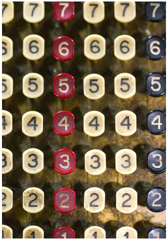
Automatic Cleanup

- Reachable objects (memory, streams, . . .) are never destroyed.
- Once unreachable, released at an unspecified future time.
- Scarce resources (memory) can be explicitly freed. (`obj.free()`)
- Correctly deals with multiple contexts and dependencies.

gpuarray: Simple Linear Algebra

`pycuda.gpuarray`:

- Meant to look and feel just like `numpy`.
  - `gpuarray.to_gpu(numpy_array)`
  - `numpy_array = gpuarray.get()`
- No: nd indexing, slicing, etc. (yet!)
- Yes: +, -, ∗, /, fill, sin, exp, rand, take, . . .
- Random numbers using `pycuda.curandom`
- Mixed types (int32 + float32 = float64)
- `print gpuarray` for debugging.
- Memory behind gpuarray available as .gpudata attribute.
  - Use as kernel arguments, textures, etc.



gpuarray: Elementwise expressions

Avoiding extra store-fetch cycles for elementwise math:

```python
from pycuda.curandom import rand as curand
a_gpu = curand((50,))
b_gpu = curand((50,))

from pycuda.elementwise import ElementwiseKernel
lin_comb = ElementwiseKernel(
        "float a, float *x, float b, float *y, float *z",
        "z[i] = a*x[i] + b*y[i]")

c_gpu = gpuarray.empty_like(a_gpu)
lin_comb(5, a_gpu, 6, b_gpu, c_gpu)

assert la.norm((c_gpu - (5*a_gpu+6*b_gpu)).get()) < 1e-5
```

PyCuda: Vital Information

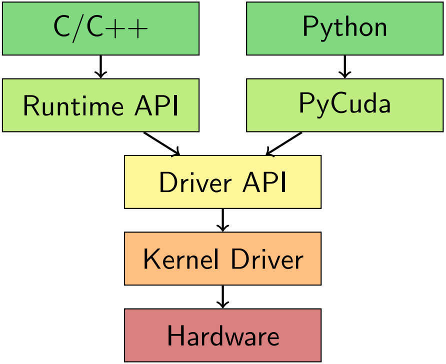
- http://mathema.tician.de/software/pycuda
- X Consortium License (no warranty, free for all use)
- Requires: numpy, Boost C++, Python 2.4+.
- Support via mailing list.

Questions?

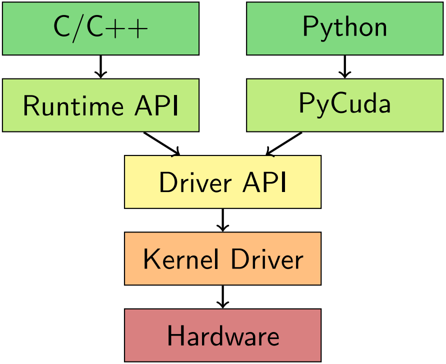
?

Outline

CUDA APIs


C/C++
Python
Runtime API
PyCuda
Driver API
Kernel Driver
Hardware

CUDA APIs



CUDA has two Programming Interfaces:

- "Runtime" high-level (`libcudart.so`, in the "toolkit")
- "Driver" low-level (`libcuda.so`, comes with GPU driver)

(mutually exclusive)

Runtime vs. Driver API

Runtime $\leftrightarrow$ Driver differences:

- Explicit initialization.

Runtime vs. Driver API

Runtime $\leftrightarrow$ Driver differences:

- Explicit initialization.
- Code objects ("Modules") become programming language objects.

Runtime vs. Driver API

Runtime $\leftrightarrow$ Driver differences:

- Explicit initialization.
- Code objects ("Modules") become programming language objects.
- Texture handling requires slightly more work.

Runtime vs. Driver API

Runtime $\leftrightarrow$ Driver differences:

- Explicit initialization.
- Code objects ("Modules") become programming language objects.
- Texture handling requires slightly more work.
- Only needs `nvcc` for compiling GPU code.

Runtime vs. Driver API

Runtime ↔ Driver differences:

- Explicit initialization.
- Code objects ("Modules") become programming language objects.
- Texture handling requires slightly more work.
- Only needs `nvcc` for compiling GPU code.

Driver API:

- Conceptually cleaner
- Less sugar-coating (provide in Python)
- Not very different otherwise

PyCuda: API Tracing

With `./configure --cuda-trace=1`:

PyCuda: API Tracing

With `./configure --cuda-trace=1`:

```
import pycuda.driver as cuda
import pycuda.autoinit
import numpy

a = numpy.random.randn(4,4).astype(numpy.float32)
a_gpu = cuda.mem_alloc(a.nbytes)
cuda.memcpy_htod(a_gpu, a)

mod = cuda.SourceModule("""
    __global__ void doublify(float *a)
    {
      int idx = threadIdx.x + threadIdx.y*4;
      a[idx] *= 2;
    }
    """)

func = mod.get_function("doublify")
func(a_gpu, block=(4,4,1))

a_doubled = numpy.empty_like(a)
cuda.memcpy_dtoh(a_doubled, a_gpu)
print a_doubled
print a
```

```
cuInit
cuDeviceGetCount
cuDeviceGet
cuCtxCreate
cuMemAlloc
cuMemcpyHtoD
cuCtxGetDevice
cuDeviceComputeCapability
cuModuleLoadData
cuModuleGetFunction
cuFuncSetBlockShape
cuParamSetv
cuParamSetSize
cuLaunchGrid
cuMemcpyDtoH
cuCtxPopCurrent
cuCtxPushCurrent
cuMemFree
cuCtxPopCurrent
cuCtxPushCurrent
cuModuleUnload
cuCtxPopCurrent
cuCtxDestroy
```

Questions?

?

Outline

Human vs Machine

In PyCuda,
CUDA C code
does *not* need to
be a compile-time
constant.

Human vs Machine

In PyCuda,
CUDA C code
does *not* need to
be a compile-time
constant.

(unlike the CUDA Runtime API)

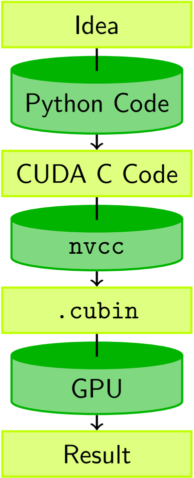
Human vs Machine

Idea

In PyCuda,
CUDA C code
does *not* need to
be a compile-time
constant.

(unlike the CUDA Runtime API)

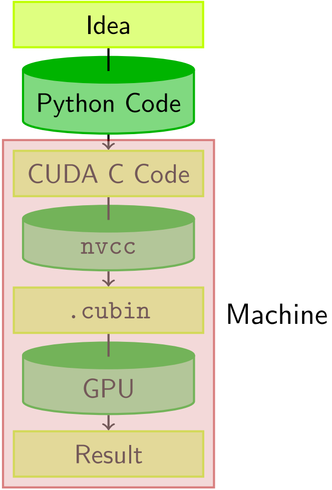
Human vs Machine



In PyCuda,
CUDA C code
does *not* need to
be a compile-time
constant.

(unlike the CUDA Runtime API)

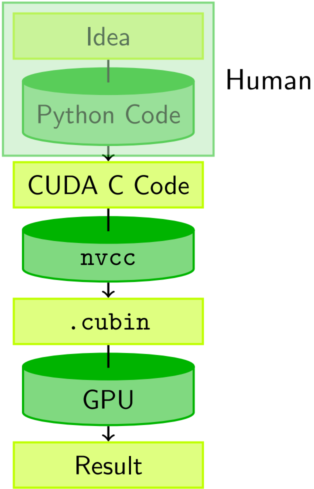
Human vs Machine



In PyCuda,
CUDA C code
does *not* need to
be a compile-time
constant.

(unlike the CUDA Runtime API)

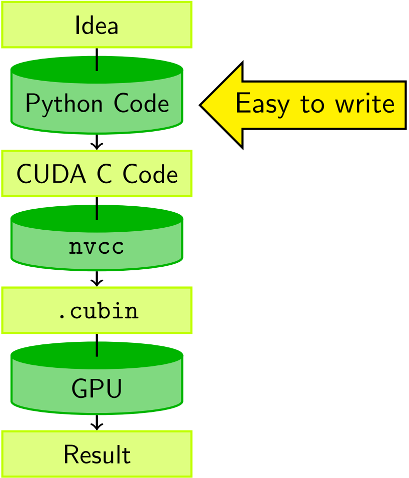
Human vs Machine



In PyCuda,
CUDA C code
does *not* need to
be a compile-time
constant.

(unlike the CUDA Runtime API)

Human vs Machine



In PyCuda,
CUDA C code
does *not* need to
be a compile-time
constant.

(unlike the CUDA Runtime API)

Metaprogramming: machine-generated code

Why machine-generated code?

- Automated Tuning
  (cf. ATLAS, FFTW)
- Data types
- Specialize code for given problem
- Constants faster than variables
  ($\rightarrow$ register pressure)
- Loop Unrolling

PyCuda: Support for Metaprogramming

- Access properties of compiled code:
  `func.{num_regs,shared_size_bytes,local_size_bytes}`
- Exact GPU timing via events
- Can calculate hardware-dependent MP occupancy
- `codepy` (by Andreas):
  - Build C syntax trees from Python
  - Generates readable, indented C
- Or use a templating engine (many available, e.g. Cheetah)

Questions?

?

Outline

Outline

Goal

- Multiply two (small) square matrices together.
- Use global memory only.
- Use a *single* block of threads.
- Each thread computes one element of the resulting matrix.

Instructions

1. cd *3-pycuda-matrixmul-simple*
2. Edit *matrixmul_simple.py*
3. Complete the *TODOs*.

Code

Initialization:

```
import numpy as np
from pycuda import driver, compiler, gpuarray, tools
import atexit

# -- initialize the device
# the following lines are equivalent to "import pycuda.autoinit"
# only if GPU_NUMBER == 0
GPU_NUMBER = 0 # TODO: change me
driver.init()
assert(driver.Device.count() >= 1)
dev = tools.get_default_device(GPU_NUMBER)
ctx = dev.make_context()
atexit.register(ctx.pop)
```

Code

Memory allocation and transfer:

```
# define the (square) matrix size
# note that we'll only use *one* block of threads here
# as a consequence this number (squared) can't exceed max_threads,
# see http://documen.tician.de/pycuda/util.html#pycuda.tools.DeviceData
# for more information on how to get this number for your device
MATRIX_SIZE = 2

# create two random square matrices
a_cpu = np.random.randn(MATRIX_SIZE, MATRIX_SIZE).astype(np.float32)
b_cpu = np.random.randn(MATRIX_SIZE, MATRIX_SIZE).astype(np.float32)

# compute reference on the CPU to verify GPU computation
c_cpu = np.dot(a_cpu, b_cpu)

# transfer host (CPU) memory to device (GPU) memory
a_gpu = gpuarray.to_gpu(a_cpu)
b_gpu = gpuarray.to_gpu(b_cpu)

# create empty gpu array for the result (C = A * B)
c_gpu = gpuarray.empty((MATRIX_SIZE, MATRIX_SIZE), np.float32)
```

Code

GPU code compilation and execution:

```
# by specifying the constant MATRIX_SIZE
kernel_code = kernel_code_template % {
    'MATRIX_SIZE': MATRIX_SIZE
    }

# compile the kernel code
mod = compiler.SourceModule(kernel_code)

# get the kernel function from the compiled module
matrixmul = mod.get_function("MatrixMulKernel")

# call the kernel on the card
matrixmul(
    # inputs
    a_gpu, b_gpu,
    # output
    c_gpu,
    # (only one) block of MATRIX_SIZE x MATRIX_SIZE threads
    block = (MATRIX_SIZE, MATRIX_SIZE, 1),
    )
```

Code

GPU kernel code:

```
kernel_code_template = """
```

```
__global__ void MatrixMulKernel(float *a, float *b, float *c)
{
    // 2D Thread ID (assuming that only *one* block will be executed)
    int tx = threadIdx.x;
    int ty = threadIdx.y;

    // Pvalue is used to store the element of the matrix
    // that is computed by the thread
    float Pvalue = 0;

    // Each thread loads one row of M and one column of N,
    //   to produce one element of P.
    for (int k = 0; k < %(MATRIX_SIZE)s; ++k) {
        float Aelement = a[ ... ]; // TODO
        float Belement = b[ ... ]; // TODO
        Pvalue += Aelement * Belement;
    }

    // Write the matrix to device memory;
    // each thread writes one element
    c[ ... ] = Pvalue; // TODO
}
```

```
"""
```

Code

GPU kernel code (solution):

```
kernel_code_template = """

__global__ void MatrixMulKernel(float *a, float *b, float *c)
{
    // 2D Thread ID (assuming that only *one* block will be executed)
    int tx = threadIdx.x;
    int ty = threadIdx.y;

    // Pvalue is used to store the element of the matrix
    // that is computed by the thread
    float Pvalue = 0;

    // Each thread loads one row of M and one column of N,
    //   to produce one element of P.
    for (int k = 0; k < %(MATRIX_SIZE)s; ++k) {
        float Aelement = a[ty * %(MATRIX_SIZE)s + k];
        float Belement = b[k * %(MATRIX_SIZE)s + tx];
        Pvalue += Aelement * Belement;
    }

    // Write the matrix to device memory;
    // each thread writes one element
    c[ty * %(MATRIX_SIZE)s + tx] = Pvalue;
}

"""
```

Outline

Goal

- Multiply two square matrices together.
- Use global memory and shared memory.
- Each thread block is assigned a "tile" of the resulting matrix and is responsible for generating the elements in that tile.
- Each thread in a block computes one element of the tile.

Code

GPU kernel code:

```
kernel_code_template = """
```

```
__global__ void MatrixMulKernel(float *A, float *B, float *C)
{

  const uint wA = %(MATRIX_SIZE)s;
  const uint wB = %(MATRIX_SIZE)s;

  // Block index
  const uint bx = blockIdx.x;
  const uint by = blockIdx.y;

  // Thread index
  const uint tx = threadIdx.x;
  const uint ty = threadIdx.y;

  // Index of the first sub-matrix of A processed by the block
  const uint aBegin = wA * %(BLOCK_SIZE)s * by;
  // Index of the last sub-matrix of A processed by the block
  const uint aEnd = aBegin + wA - 1;
  // Step size used to iterate through the sub-matrices of A
  const uint aStep = %(BLOCK_SIZE)s;

  // Index of the first sub-matrix of B processed by the block
  const uint bBegin = %(BLOCK_SIZE)s * bx;
  // Step size used to iterate through the sub-matrices of B
  const uint bStep = %(BLOCK_SIZE)s * wB;
```

Code

GPU kernel code (cont'd):

```
// The element of the block sub-matrix that is computed
// by the thread
float Csub = 0;
// Loop over all the sub-matrices of A and B required to
// compute the block sub-matrix
for (int a = aBegin, b = bBegin;
     a <= aEnd;
     a += aStep, b += bStep)
  {
    // Shared memory for the sub-matrix of A
    __shared__ float As[%(BLOCK_SIZE)s][%(BLOCK_SIZE)s];
    // Shared memory for the sub-matrix of B
    __shared__ float Bs[%(BLOCK_SIZE)s][%(BLOCK_SIZE)s];

    // Load the matrices from global memory to shared memory;
    // each thread loads one element of each matrix
    As[ty][tx] = A[a + wA * ty + tx];
    Bs[ty][tx] = B[b + wB * ty + tx];
    // Synchronize to make sure the matrices are loaded
    __syncthreads();
```

Code

GPU kernel code (cont'd):

```
      // Multiply the two matrices together;
      // each thread computes one element
      // of the block sub-matrix
      for (int k = 0; k < %(BLOCK_SIZE)s; ++k)
        Csub += As[ty][k] * Bs[k][tx];

      // Synchronize to make sure that the preceding
      // computation is done before loading two new
      // sub-matrices of A and B in the next iteration
      __syncthreads();
    }

  // Write the block sub-matrix to global memory;
  // each thread writes one element
  const uint c = wB * %(BLOCK_SIZE)s * by + %(BLOCK_SIZE)s * bx;
  C[c + wB * ty + tx] = Csub;
}
```

```
"""
```

Outline

Goal

- Multiply two matrices together (any size).
- Use global memory and shared memory.
- Implement various optimizations:
  - different granularities of parallelism (block and work sizes),
  - loop unrolling,
  - register pressure (spilling),
  - pre-fetching (global memory load).
- "Instrumentalize" the code using a *template engine* (Cheetah).
- Auto-tune depending on the hardware *and* the input data.

Instructions

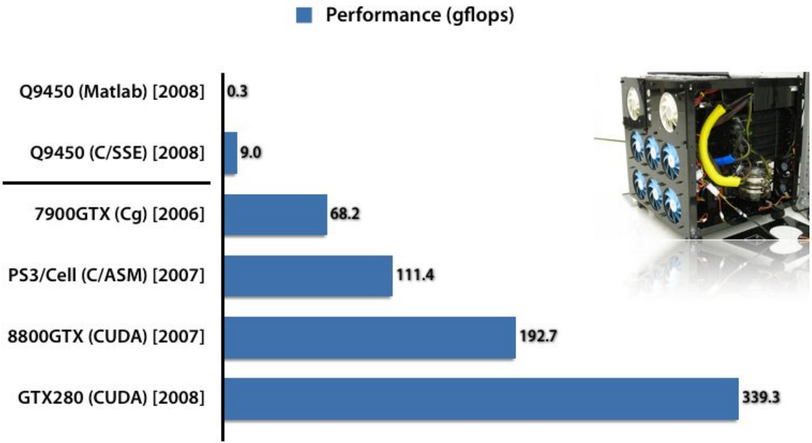
1. cd *5-pycuda-matrixmul-opt*
2. Implement your auto-tuning function.
3. Use PyCuda to gather informations (registers, occupancy).

Code

Show the code ;-)

Some numbers

3D Filterbank Convolutions used in our Visual Cortex Simulations:



Conclusions

- GPUs (or something like them) are here to stay.

Conclusions

- GPUs (or something like them) are here to stay.
- First factor of 5-10 is usually easy to reach.

Conclusions

- GPUs (or something like them) are here to stay.
- First factor of 5-10 is usually easy to reach.
- Next factor of 5-10 is a little bit harder.

Conclusions

- GPUs (or something like them) are here to stay.
- First factor of 5-10 is usually easy to reach.
- Next factor of 5-10 is a little bit harder.
- Next factor of 5-10 is a lot harder.
  - Reqires deep understanding of the hardware architecture.
  - Usually involves significant rethinking of algorithm.

Conclusions

- GPUs (or something like them) are here to stay.
- First factor of 5-10 is usually easy to reach.
- Next factor of 5-10 is a little bit harder.
- Next factor of 5-10 is a lot harder.
  - Reqires deep understanding of the hardware architecture.
  - Usually involves significant rethinking of algorithm.
- GPUs and scripting work surprisingly well together.
  - Favorable balance btw ease-of-use and raw performance.
  - Enable (easy) Metaprogramming.

Conclusions

- GPUs (or something like them) are here to stay.
- First factor of 5-10 is usually easy to reach.
- Next factor of 5-10 is a little bit harder.
- Next factor of 5-10 is a lot harder.
  - Reqires deep understanding of the hardware architecture.
  - Usually involves significant rethinking of algorithm.
- GPUs and scripting work surprisingly well together.
  - Favorable balance btw ease-of-use and raw performance.
  - Enable (easy) Metaprogramming.
- Python / PyCuda rocks!

Thank you