Exploit your GPU power with PyCUDA and friends

Stefano Brilli

developer @ Develer

stefanobrilli@gmail.com

Exploit your GPU power with PyCUDA and friends

Exploit your GPU power with PyCUDA and friends

Exploit your GPU power with PyCUDA and friends

Reference Site

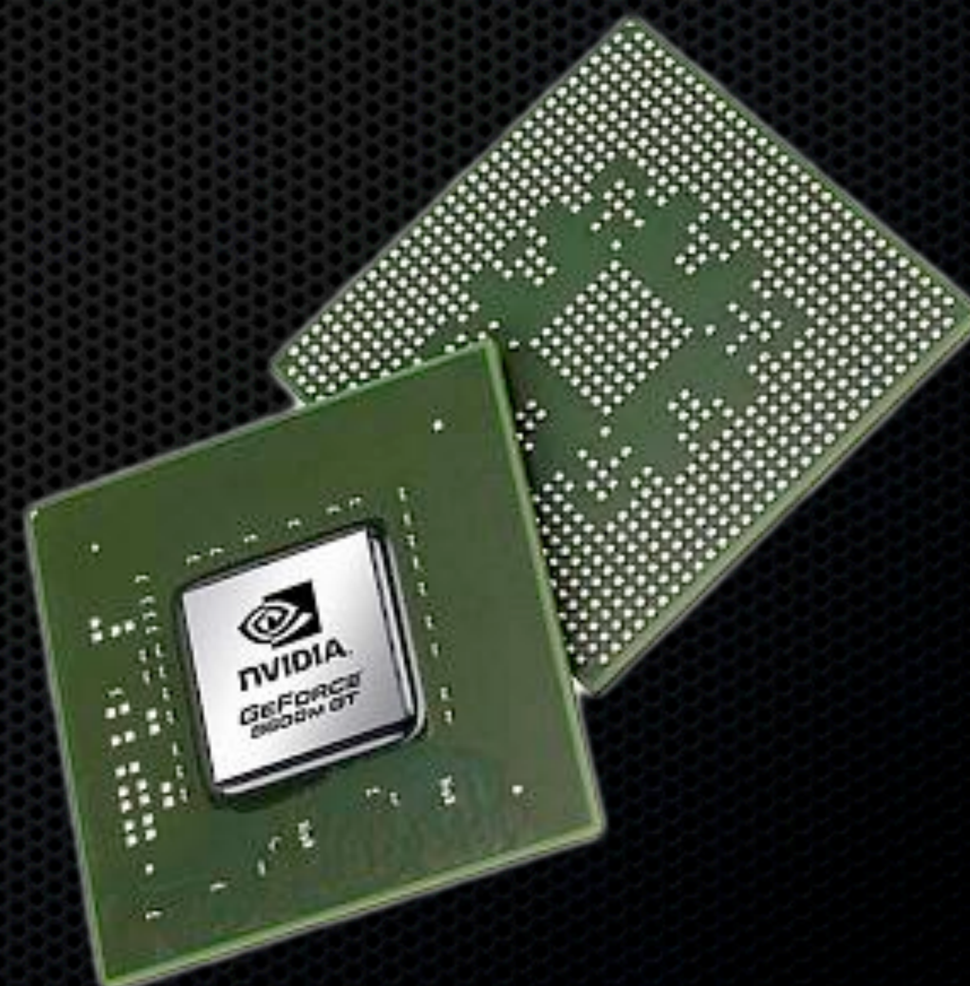
http://sites.google.com/site/ep2011cuda

Topics

- What is a GPU?
- GPU Computing
- CUDA
- PyCUDA
- Short Example
- Tasks that run fast on GPU
- Other libraries

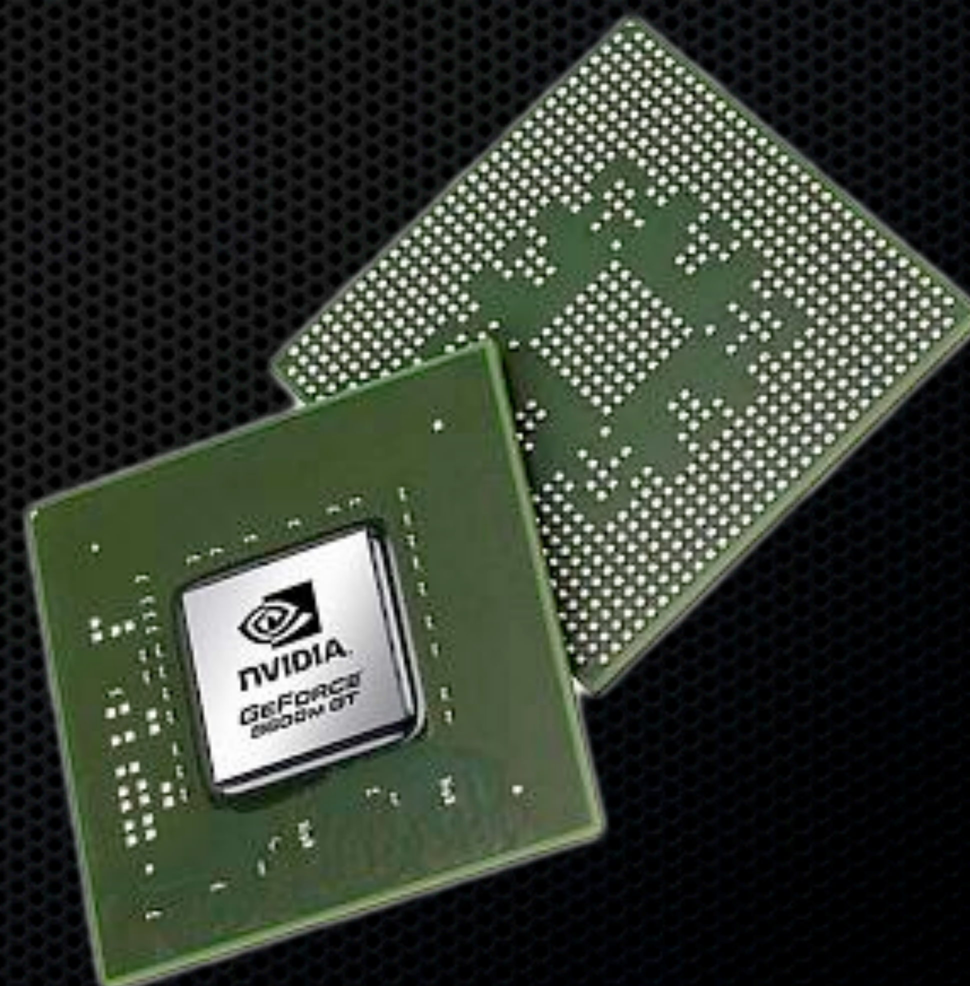
What is a GPU?

What is a GPU?

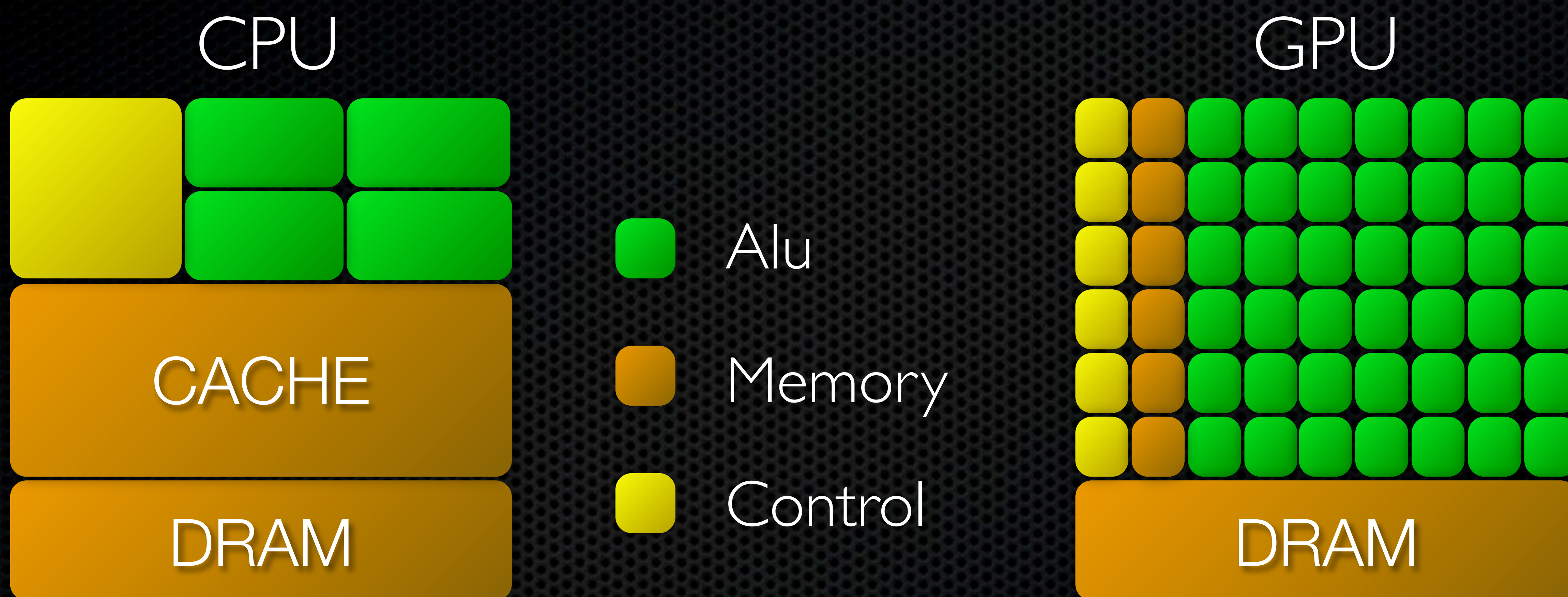


What is a GPU?

- Found in most of all modern devices
- GPUs were initially projected to help the CPU in graphics computations
- Became fast and powerful multicore processors



It's evolution, baby!



Bigger cache

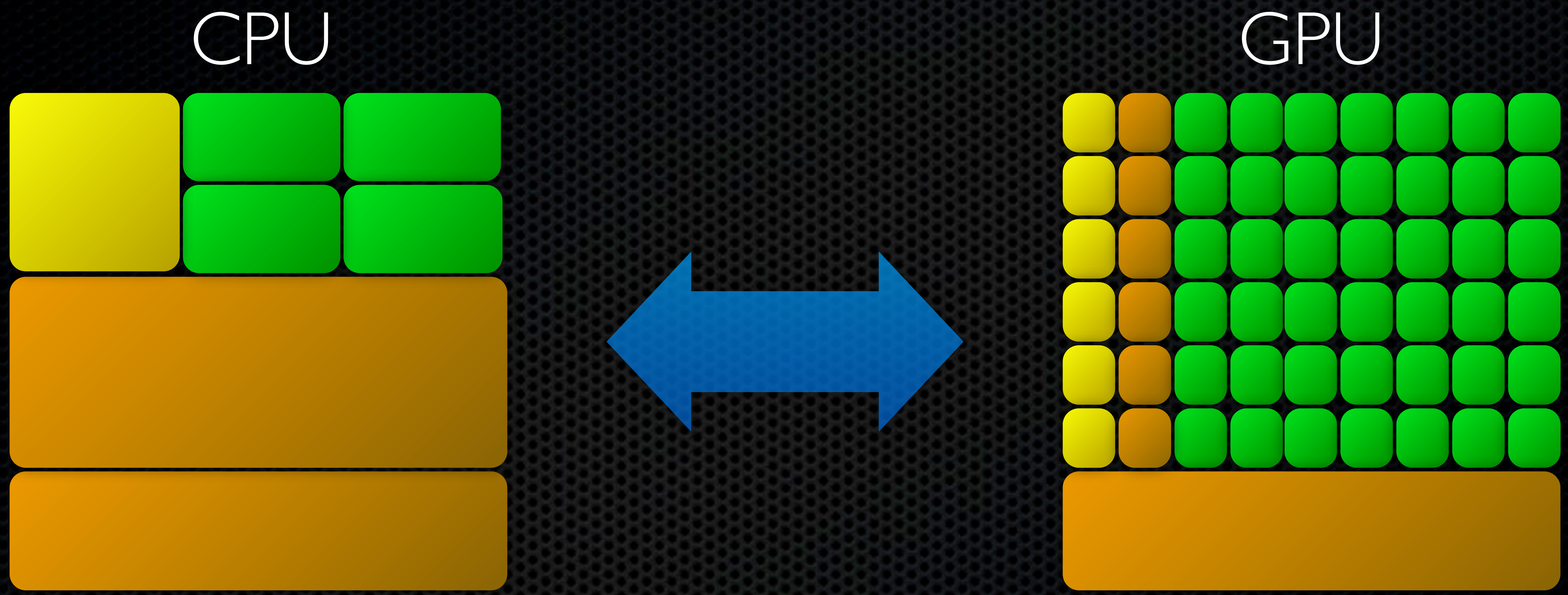
Complex control

High arithmetical throughput

Simple control & small cache

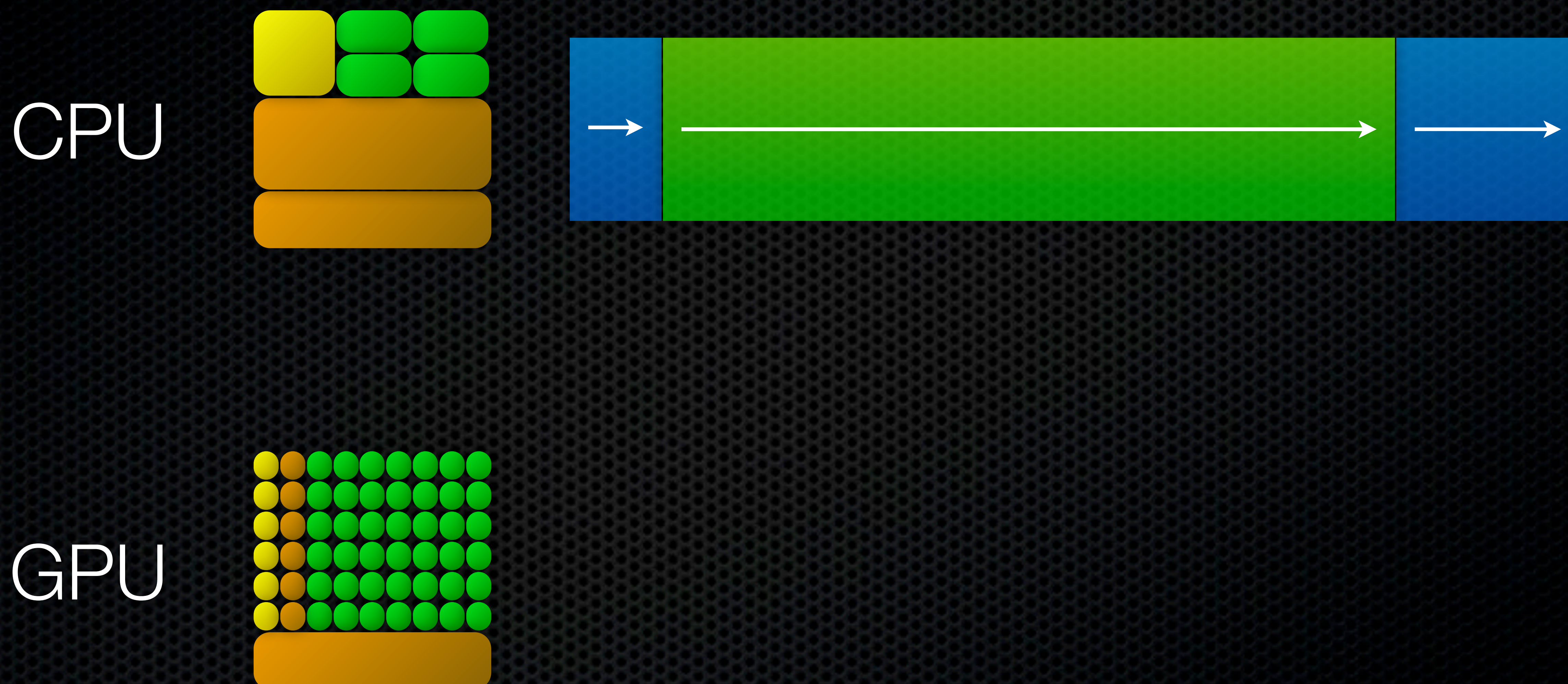
GPU Computing

In GPU Computing



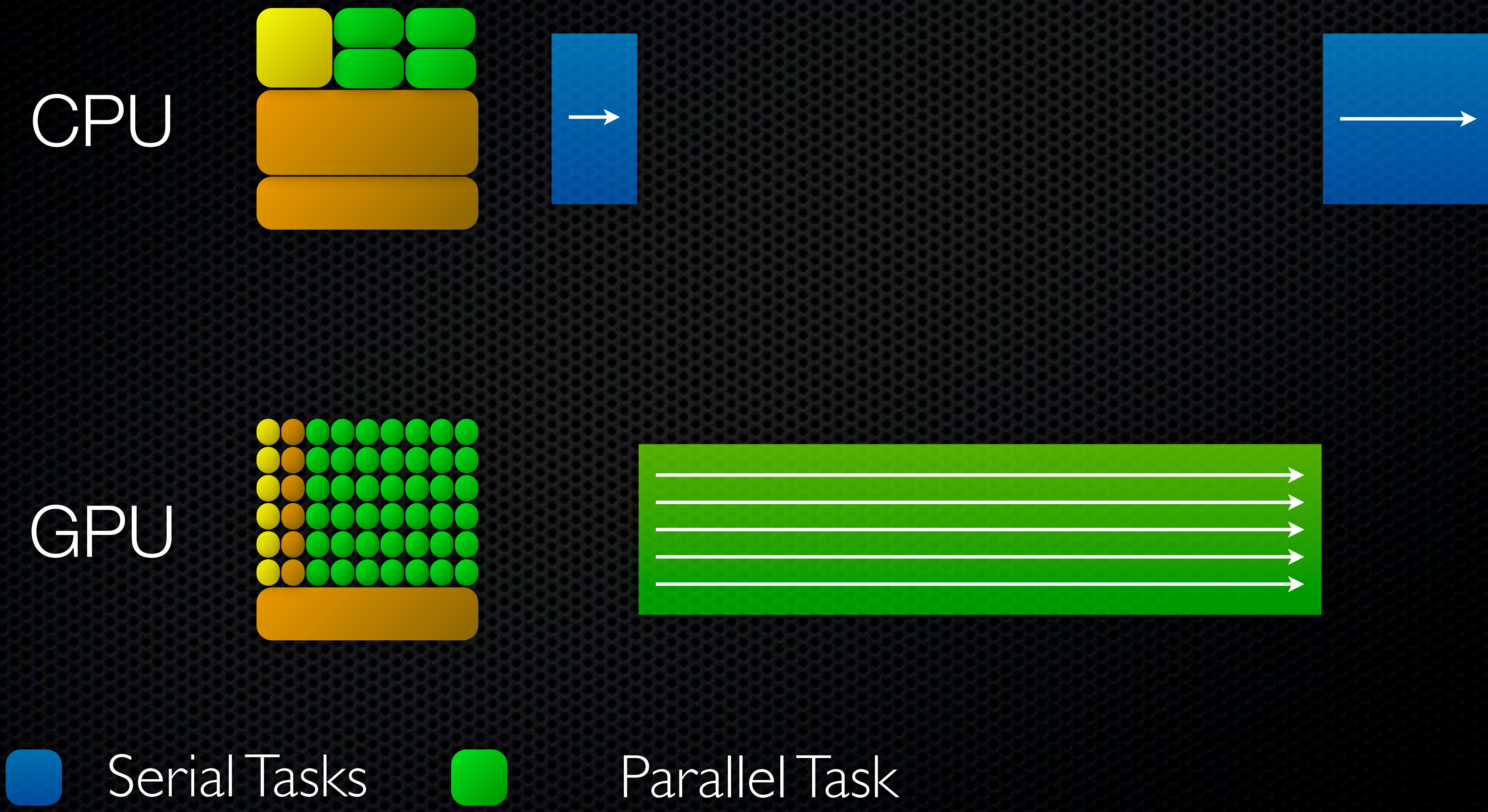
... GPU works as a CPU coprocessor

GPU Computing



Serial Tasks Parallel Task

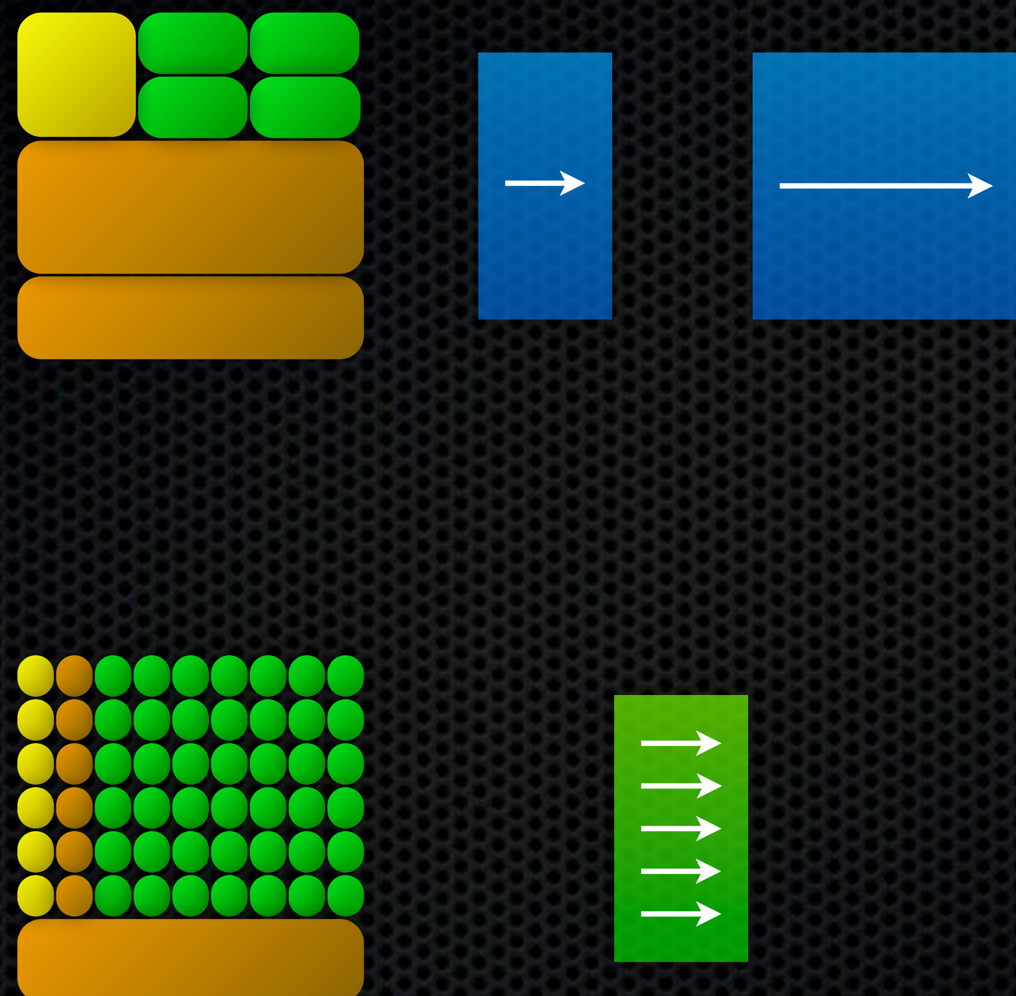
GPU Computing



GPU Computing

CPU

GPU



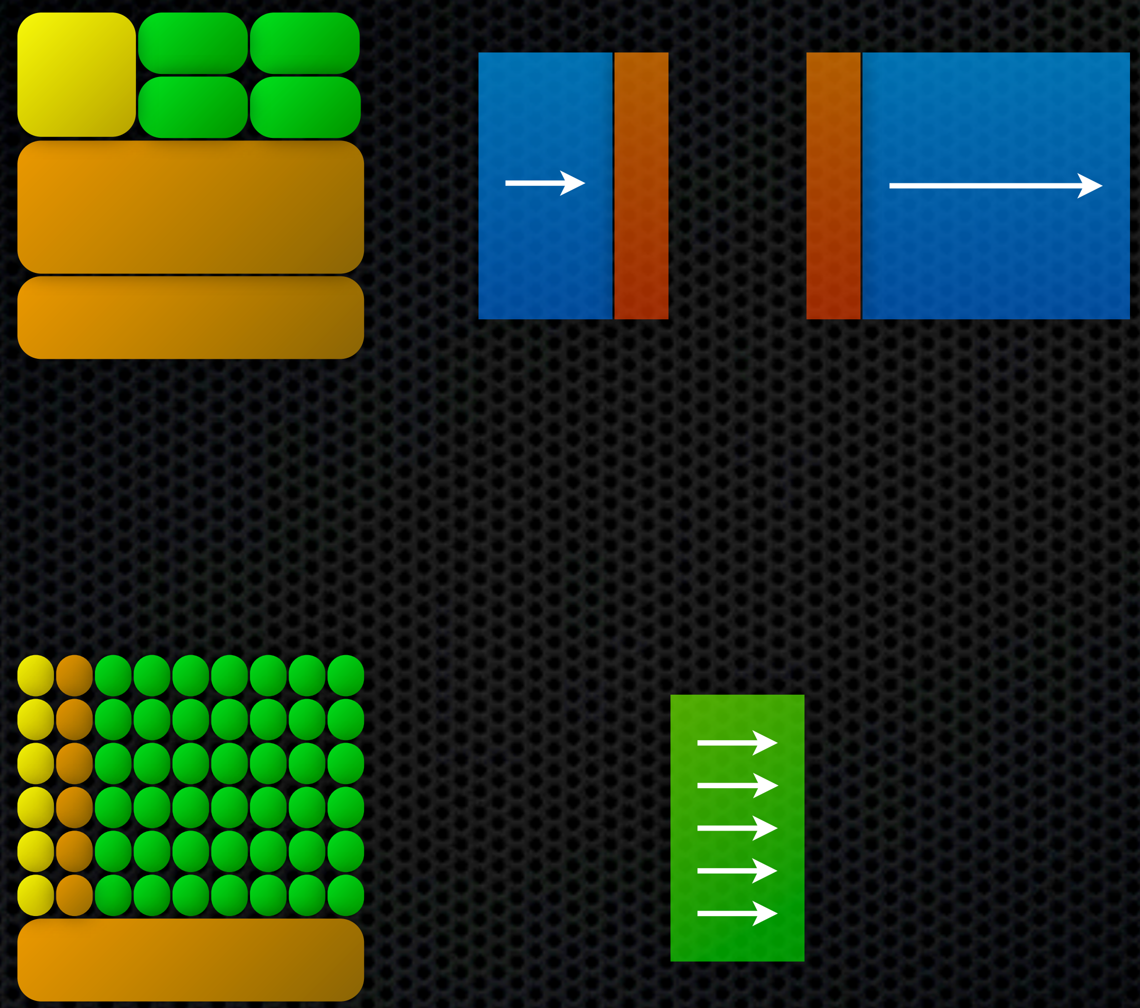
Application speed up's limited by Amdahl law!

Serial Tasks Parallel Task

GPU Computing

CPU

GPU



Data transfers overheads!

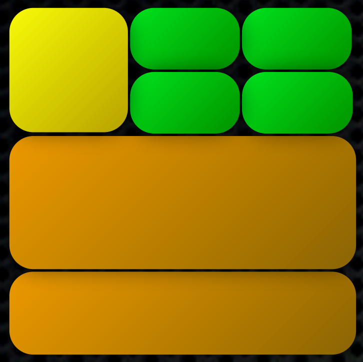
Serial Tasks — Parallel Task — Overhead

CUDA

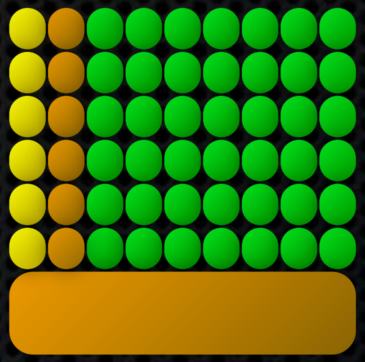
Compute Unified Device Architecture

CUDA

- NVIDIA technology for doing GPU computing
- Largely diffused in HPC and end user software
- Not the only choice for GPU computing: OpenCL
- Some advantages (IMHO)
  - Maturity
  - Support

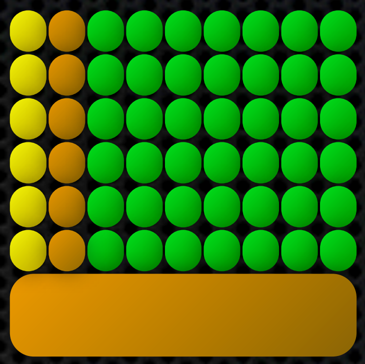


Host



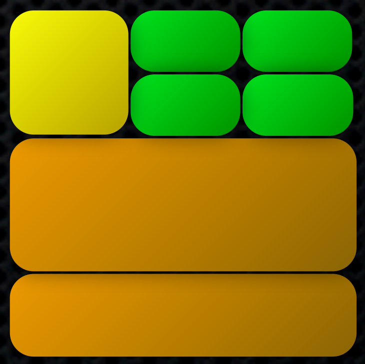
Device

```
__global__ void vecAdd(float* A, float*B, float*C)
{
    int i = threadIdx.x + blockDim.x * blockIdx.x;
    C[i] = A[i] + B[i];
}
```

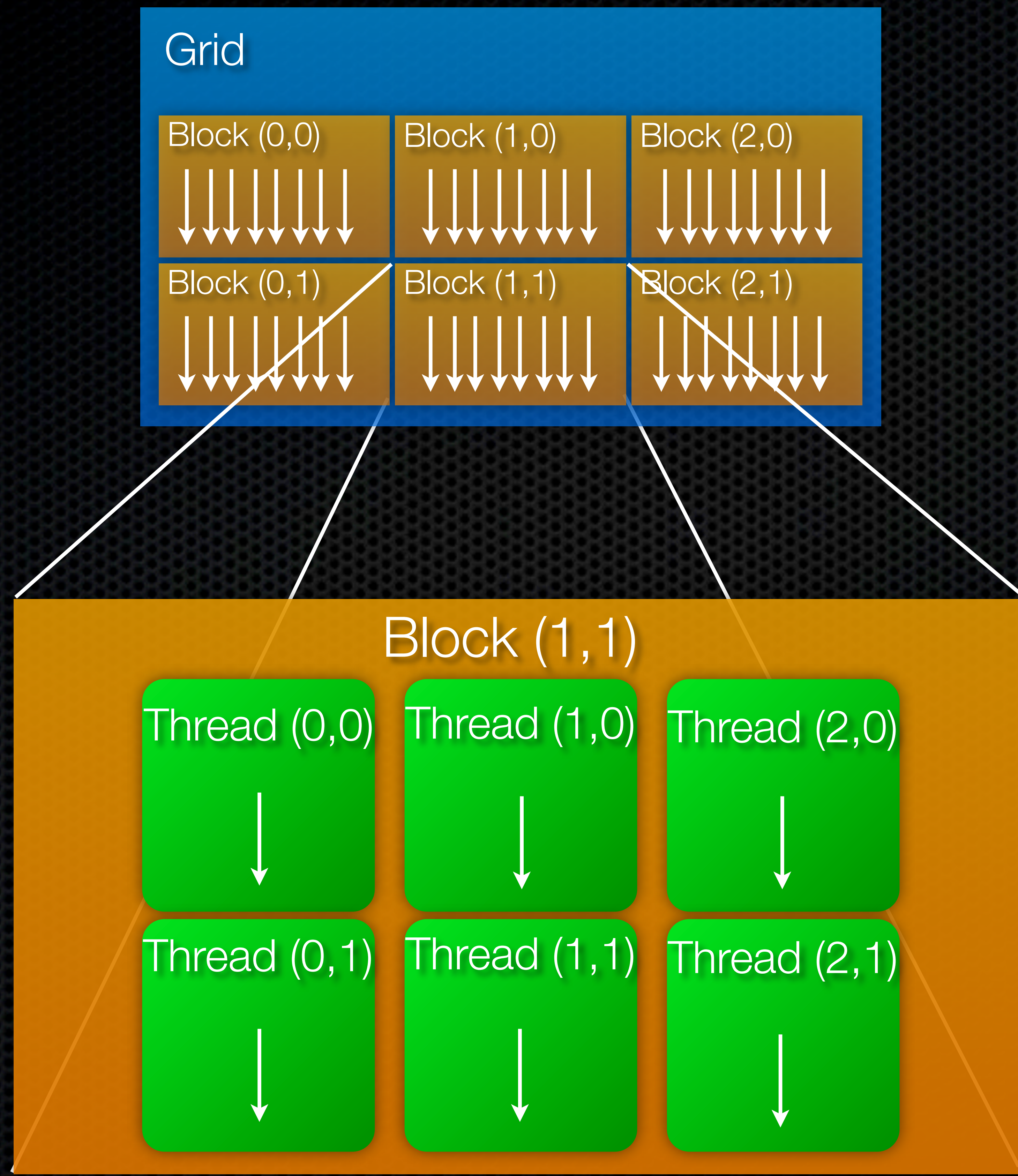


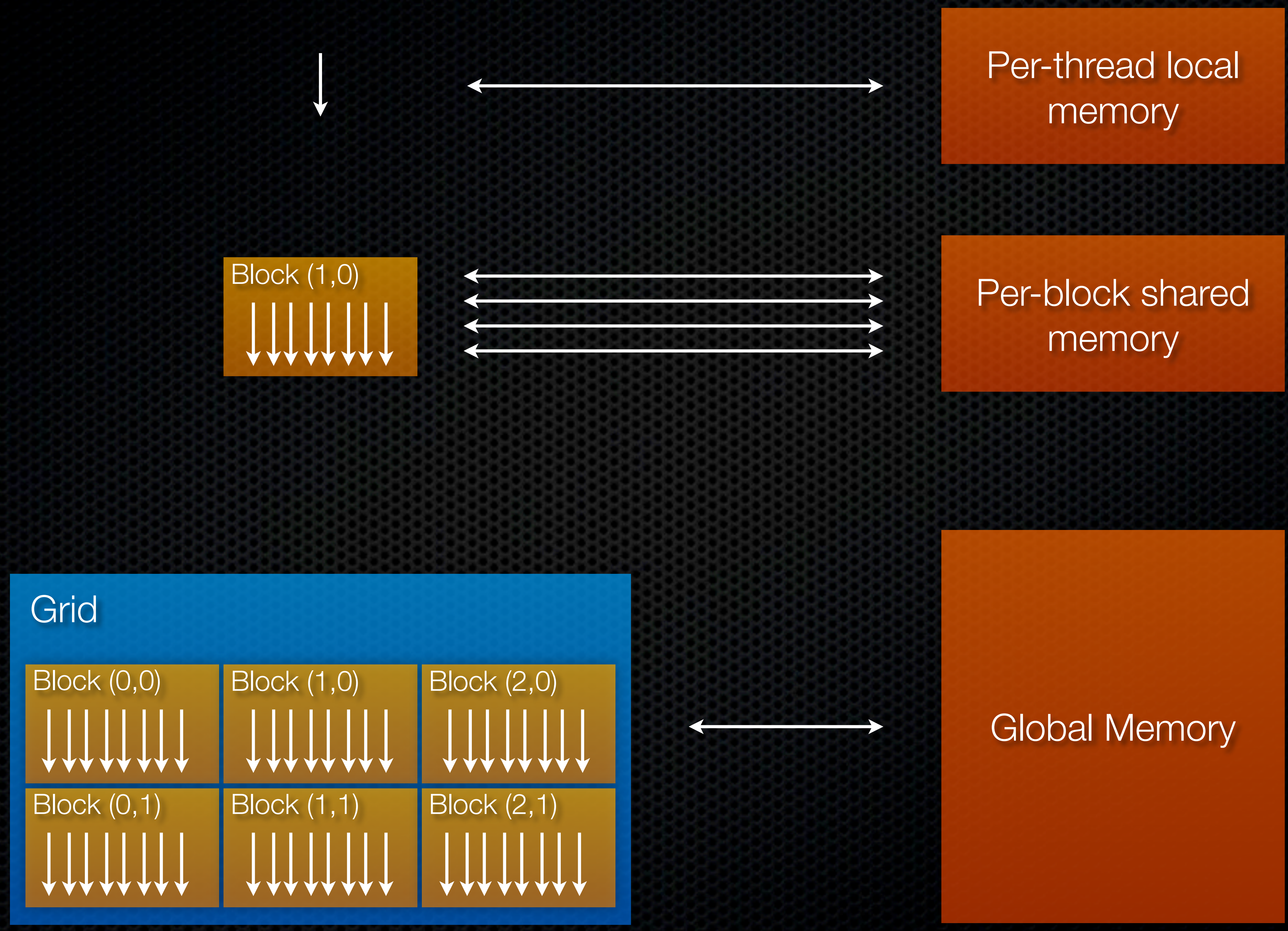
Device Code

```
int main()
{
    float* A, *B, *C;
    vecAdd<<< BLOCKS, THREADS >>> (A, B, C);
}
```



Host Code


Grid
Block (0,0)
Block (1,0)
Block (2,0)
Block (0,1)
Block (1,1)
Block (2,1)
Block (1,1)
Thread (0,0)
Thread (1,0)
Thread (2,0)
Thread (0,1)
Thread (1,1)
Thread (2,1)


Per-thread local memory
Block (1,0)
Per-block shared memory
Grid
Block (0,0)
Block (1,0)
Block (2,0)
Block (0,1)
Block (1,1)
Block (2,1)
Global Memory

PyCUDA

Python wrappers for CUDA

PyCUDA

- Python wrapper for CUDA by Andreas Klöckner
- Pythonic!
    - numpy integration
    - object cleanup
    - error checking
- Metaprogramming

Short Example

Sum of two vectors

CPU way

One thread sum N elements

```
for (int i=0; i<N; ++i)
    dest[i] = a[i] + b[i];
```

GPU way

N threads sum One element

```
dest[id] = a[id] + b[id];
```

CPU way

One thread sum N elements

```
for (int i=0; i<N; ++i)
    dest[i] = a[i] + b[i];
```

GPU way

N threads sum One element

```
dest[id] = a[id] + b[id];
```

NVIDIA C APIs

```c
#include "cuda.h"
#include "stdio.h"

__global__ void vecAdd(float *dest, float *a, float *b, int N)
{
  const int i = threadIdx.x + blockIdx.x * blockDim.x;
  if (i < N)
    dest[i] = a[i] + b[i];
}

void checkError(cudaError_t err)
{
    if (err != 0)
    {
        printf("Error\n");
        exit(1);
    }
}

int main()
{
    float *a, *b, *dest;
    float *d_a, *d_b, *d_dest;

    int SZ = 1000;

    a = (float*)malloc(SZ*sizeof(float));
    b = (float*)malloc(SZ*sizeof(float));
    dest = (float*)malloc(SZ*sizeof(float));

    checkError( cudaMalloc(&d_a, SZ*sizeof(float)) );
    checkError( cudaMalloc(&d_b, SZ*sizeof(float)) );
    checkError( cudaMalloc(&d_dest, SZ*sizeof(float)) );

    for (int i=0; i < SZ; ++i)
    {
        a[i] = i; b[i] = SZ - i; dest[i] = 0;
    }

    checkError( cudaMemcpy(d_a, a, SZ*sizeof(float), cudaMemcpyHostToDevice) );
    checkError( cudaMemcpy(d_b, b, SZ*sizeof(float), cudaMemcpyHostToDevice) );

    vecAdd<<<(SZ+255)/256, 256>>>(d_dest, d_a, d_b, SZ);

    checkError( cudaMemcpy(dest, d_dest, SZ*sizeof(float), cudaMemcpyDeviceToHost) );
    for(int i = 0; i < SZ; ++i)
    {
        if (dest[i] != (a[i] + b[i]) )
        {
            printf("Error\n");
            exit(1);
        }
    }
    printf("Success\n");

    free(a);
    free(b);
    free(dest);
    cudaFree(d_a);
    cudaFree(d_b);
    cudaFree(d_dest);

}
```

PyCUDA

```python
#!/usr/bin/env python
#-*- coding: utf-8 -*-
import pycuda.autoinit
import pycuda.driver as drv
import numpy

from pycuda.compiler import SourceModule
mod = SourceModule("""
__global__ void vecAdd(float *dest, float *a, float *b, int N)
{
  const int i = threadIdx.x + blockIdx.x * blockDim.x;
  if (i < N)
    dest[i] = a[i] + b[i];
}
""")

vecAdd = mod.get_function("vecAdd")

SZ = 1000
a = numpy.arange(1, SZ).astype(numpy.float32)
b = numpy.arange(SZ-1, 0, -1).astype(numpy.float32)
dest = numpy.zeros_like(a)

vecAdd(drv.Out(dest), drv.In(a), drv.In(b), numpy.int32(SZ), block=(256, 1, 1), grid=((SZ+255)/256 ,1))
print "Error" if any( dest-(a+b) != 0) else "Success"
```

NVIDIA C APIs

```c
#include "cuda.h"
#include "stdio.h"
```

PyCUDA

```python
#!/usr/bin/env python
#-*- coding: utf-8 -*-
import pycuda.autoinit
import pycuda.driver as drv
from pycuda.compiler import SourceModule
import numpy
```

NVIDIA C APIs

PyCUDA

```
__global__ void vecAdd(float *dest, float *a, float *b, int N)
{
  const int i = threadIdx.x + blockIdx.x * blockDim.x;
  if (i < N)
    dest[i] = a[i] + b[i];
}
```

NVIDIA C APIs

```
__global__ void vecAdd(float *dest, float *a, float *b, int N)
{
  const int i = threadIdx.x + blockIdx.x * blockDim.x;
  if (i < N)
    dest[i] = a[i] + b[i];
}
```

PyCUDA

```
mod = SourceModule("""
__global__ void vecAdd(float *dest, float *a, float *b, int N)
{
  const int i = threadIdx.x + blockIdx.x * blockDim.x;
  if (i < N)
    dest[i] = a[i] + b[i];
}
""")
```

NVIDIA C APIs

```
void checkError(cudaError_t err)
{
    if (err != 0)
    {
        printf("Error\n");
        exit(1);
    }
}
```

PyCUDA

NO NEED FOR ERROR CHECKING

NVIDIA C APIs

```
int SZ = 1000;
a = (float*)malloc(SZ*sizeof(float));
b = (float*)malloc(SZ*sizeof(float));
dest = (float*)malloc(SZ*sizeof(float));

for (int i=0; i < SZ; ++i) {
    a[i] = i; b[i] = SZ - i; dest[i] = 0;
}
```

PyCUDA

```
SZ = 1000
a = numpy.arange(1, SZ).astype(numpy.float32)
b = numpy.arange(SZ-1, 0, -1).astype(numpy.float32)
dest = numpy.zeros_like(a)
```

NVIDIA C APIs

```
checkError( cudaMalloc(&d_a, SZ*sizeof(float)) );
checkError( cudaMalloc(&d_b, SZ*sizeof(float)) );
checkError( cudaMalloc(&d_dest, SZ*sizeof(float)) );

checkError( cudaMemcpy(d_a, a, SZ*sizeof(float),
cudaMemcpyHostToDevice) );
checkError( cudaMemcpy(d_b, b, SZ*sizeof(float),
cudaMemcpyHostToDevice) );
```

PyCUDA

NO NEED TO EXPLICITLY MOVE
DATAS TO DEVICE MEMORY

NVIDIA C APIs

```
vecAdd<<<(SZ+255)/256, 256>>>(d_dest, d_a, d_b, SZ);

checkError( cudaMemcpy(dest, d_dest, SZ*sizeof(float),
cudaMemcpyDeviceToHost) );
```

PyCUDA

```
vecAdd(drv.Out(dest), drv.In(a), drv.In(b), numpy.int32(SZ),
       block=(256, 1, 1), grid=((SZ+255)/256 ,1))
```

NVIDIA C APIs

```
for(int i = 0; i < SZ; ++i) {
    if (dest[i] != (a[i] + b[i]) ) {
        printf("Error\n");
        exit(1);
    }
}
printf("Success\n");
```

PyCUDA

```
print "Error" if any( dest-(a+b) != 0)
else "Success"
```

NVIDIA C APIs

```
free(a);
free(b);
free(dest);
cudaFree(d_a);
cudaFree(d_b);
cudaFree(d_dest);
```

PyCUDA

NO NEED TO FREE RESOURCES

Task that runs fast on GPU

Making a task run fast

- Arithmetical intensity
- Not so difficult writing CUDA kernels that works
  - Harder to write kernels that run fast
- Optimization is a matter of
  - Hardware Architecture
  - Algorithm
- Beutiful example about Reductions on my reference site
  - It requires good architecture knowledge

Copperhead

Data Parallel Python

Copperhead

```
from copperhead import *

@cu
def axpy(a, x, y):
    return [ a * xi + yi for xi, yi in zip(x, y) ]

x = [ 1.0, 1.0, 1.0, 1.0 ]
y = [ 1.0, 2.0, 3.0, 4.0 ]

gpu = axpy(2.0, x, y)
```

Copperhead

- Subset of Python syntax
  - No classes or metaclasses
  - Strong typing
- Can works with numpy arrays
- Easy switching between CPU and GPU

YES:

```
def axpy(a, x, y):
    return map(lambda xi, yi: a * xi + yi, x, y)
```

NO:

```
def axpy(a, x, y):
    for i in indices(y):
        y[i] = a * x[i] + y[i]
    return y
```

Copperhead

- Subset of Python syntax
  - No classes or metaclasses
  - Strong typing
- Can works with numpy arrays
- Easy switching between CPU and GPU

```
from copperhead import *
import numpy as np

@cu
def axpy(a, x, y):
    return [ a * xi + yi for xi, yi in zip
(x, y) ]

x = numpy.ones(1000)
y = numpy.arange(1000)

gpu = axpy(2.0, x, y)
```

Copperhead

- Subset of Python syntax
  - No classes or metaclasses
  - Strong typing
- Can works with numpy arrays
- Easy switching between CPU and GPU

```python
with places.here:
    # Executed on CPU
    cpu = axpy(2.0, x, y)

with places.gpu0:
    # Executed on GPU
    gpu = axpy(2.0, x, y)
```

Copperhead

- Lacks of an extensive documentation
- Aims to supports other platforms than CUDA
- Yet not ready for production environments

Some documentation on

code.google.com/p/copperhead

Copperhead

- Lacks of an extensive documentation
- Aims to supports other platforms than CUDA
- Yet not ready for production environments

"Copperhead is a project to bring data parallelism to Python"

from copperhead project home page

Copperhead

- Lacks of an extensive documentation
- Aims to supports other platforms than CUDA
- Yet not ready for production environments

Theano

CPU and GPU Math Expression Compiler

Theano

```
from theano import function, tensor as T

a = T.fscalar('a')
x = T.fvector('x')
y = T.fvector('y')
axpy = function([a, x, y], a * x + y)

v0 = [1.0, 1.0, 1.0, 1.0]
v1 = [1.0, 2.0, 3.0, 4.0]

res = axpy(2.0, v0, v1)
```

Theano

- Transparent GPU computing
- Integration with numpy
- Exaustive documentation
- Ready for production environment

```
# export THEANO_FLAGS="device=gpu"
# python theano_program.py
```

Theano

- Transparent GPU computing
- Integration with numpy
- Exaustive documentation
- Ready for production environment

```
from theano import function,
tensor as T
import numpy as np

a = T.fscalar('a')
x = T.fvector('x')
y = T.fvector('y')
axpy = function([a, x, y], a * x + y)

v0 = np.ones(1000)
v1 = np.arange(1000)

res = axpy(2.0, v0, v1)
```

Theano

- Transparent GPU computing
- Integration with numpy
- Exaustive documentation
- Ready for production environment

documentation on

deeplearning.net/software/theano/

Theano

- Transparent GPU computing
- Integration with numpy
- Exaustive documentation
- Ready for production environment

BSD Licence

Active community

Bug tracking system

Theano

- No full performace portability
- Not all functions run faster on GPU

Use of shared variables to reduce memory transfers

Theano

- No performace portability
- Not all functions run faster on GPU

High arithmetic intensity needed for high speedups