resophonic::kamasu

Computing on the GPU with CUDA and boost::proto

Troy D. Straszheim
Resophonic Systems, Inc.
Washington, DC

resophonic::kamasu
Computing on the GPU with CUDA and boost::proto

Troy D. Straszheim
Resophonic Systems, Inc.
Washington, DC

honored to be here,
First read modern c++ design in about 2002 or 3, wrote everything with policy classes for awhile, still get a thrill out of it
love this kind of thing as it gives me an opportunity to look in to the minds of people that are smarter than I am, and not in real-time, and in ascii, so there's little danger that I'll need eyebleach
last year at boostcon, hartmut was encouraging
i thought great, opportunity to learn proto, look into eric niebler's mind
CUDA was getting buzz
opportunity to do both, and for the next hour or so I'm going to tell you what happened. these are the proceeds thus far.
haven't really discussed what I'm doing with anybody yet, so I expect you to see lots of things I'm overlooking, looking forward to what happens after this talk, lots of time for questions and discussion.

Not a finished product.

The Underpants Gnomes' Business Plan

1. Collect Underpants
2. ???
3. Profit

The Underpants Gnomes' Business Plan

1. Collect Underpants
2. ???
3. Profit

The Kamasu Business Plan

1. Learn boost::proto
2. Learn CUDA
3. Combine
4. ???
5. Profit

The Kamasu Business Plan

1. Learn boost::proto
2. Learn CUDA
3. Combine
4. ???
5. Profit

The good news is that the problem-hungry amongst you are about to be well fed

The Kamasu Business Plan

1. Learn boost::proto
2. Learn CUDA
3. Combine
4. ??? **<- You are here**
5. Profit

The Kamasu Business Plan

1. Learn boost::proto
2. Learn CUDA
3. Combine
4. ??? **<- You are here**
5. Profit

The good news is that the problem-hungry amongst you are about to be well fed

Outline

CUDA

Related efforts

boost::proto
    transforms

resophonic::kamasu
    benchmarks

Outline

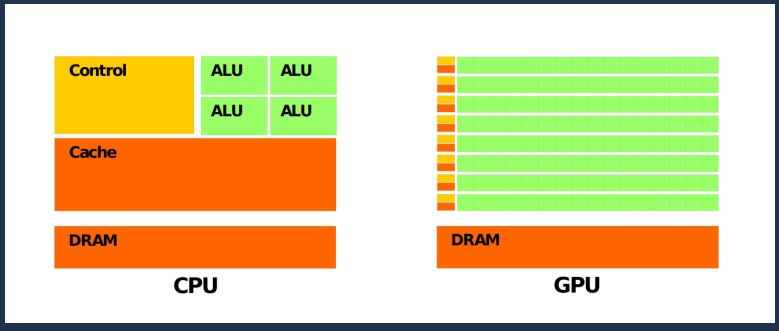
CUDA

Related efforts

boost::proto
    transforms

resophonic::kamasu
    benchmarks

The GPU vs the CPU

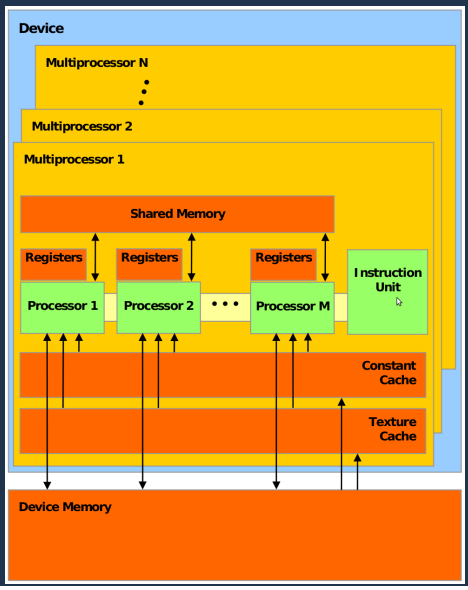


The GPU vs the CPU

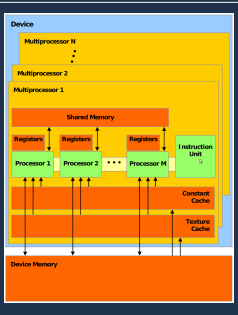
gpu specialized for compute-intensive, highly parallel computation, which is what grahics is all about, so more of it is dedicated to doing math.
it has a lot less sophisticated flow control
a lot smaller cache
the green bits are

presumably that's not to scale

The GPU



The GPU

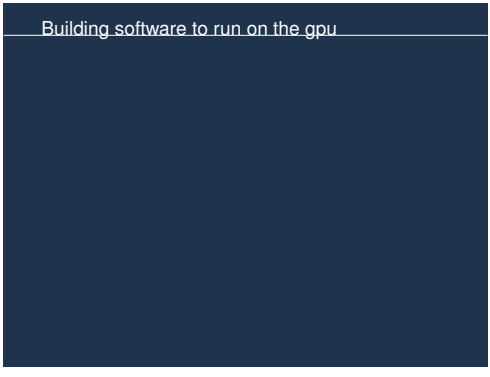


This is Single Instruction Multi Thread, handles hundreds of threads running different programs.
Broken up into Streaming Multiprocessors, seen here, which have eight Scalar Processor cores, an instruction unit, shard memory, some cache. This thing creates and manages threads and implements __syncthreads() which is for synchronization.
Constant cache is fast and constant
Texture cache is fast and constant and has some extra capabilities for filtering and whatnot.
video card GTX 280 in my machine has 30 of these multiprocessors, 8 cores on each for a total of 240 cores, clock is 1.35GHz
Each multiprocessor is composed of eight processors, so that a multiprocessor is able to process the 32 threads of a warp in four clock cycles.

There is an organizational scheme that they use involving warps, blocks and grids, not very important for the sake of this talk.

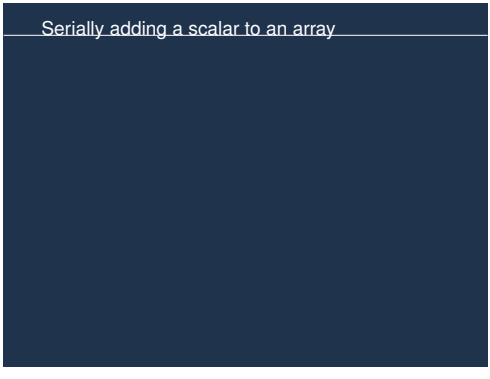
Building software to run on the gpu

nvcc and the like functions that run on the host only, on the device only, and callable from the host that run on the device



Serially adding a scalar to an array

```
void add(float* data, unsigned size, float scalar)
{
  for(unsigned i=0; i<size; i++)
    data[i] += scalar;
}
```

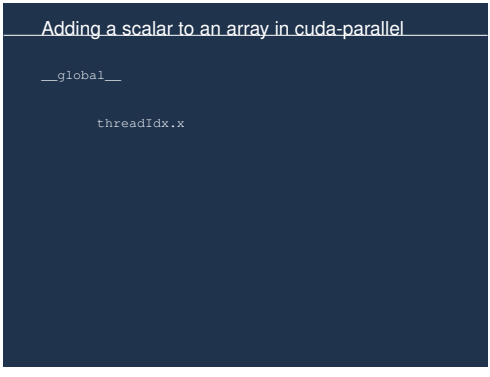



looking at how to add a scalar to a vector.

this is the sequential C version we're all familiar with.

Adding a scalar to an array in cuda-parallel

```
__global__ void
add(float *data, float scalar)
{
  data[threadIdx.x] += scalar;
}
```

This is a 'kernel' or a function that executes on the device, but is called by the host.
the __global__ specifier makes it so. this function is compiled by nvcc into code that runs on the gpu.
this threadidx is a built in variable that the nvcc compiler puts into every kernel function. it is actually has 3 dimensional structure, a 2 dimensional 'grid' of 'blocks', and a thread index within the block.
Here's how we launch it. First we somehow make an array of N elements out on the GPU, we'll get to that, then we use this magic anglebracket syntax that NVCC preprocesses into a bunch of function calls to an nvidia written library that fetches the compiled code for the kernel from wherever its been stored, ships it out to the video card, and executes it however many times is specified with this grid/block stuff. From our perspective they run simultaneously, though they might be on different multiprocessors and whatever. When threads have to communicate with each other or synchronize access to shared memory things get more complicated but that's all outside the scope of this talk.

Adding a scalar to an array in cuda-parallel

```
__global__ void
add(float *data, float scalar)
{
  data[threadIdx.x] += scalar;
}
```


Adding a scalar to an array in cuda-parallel

```
__global__

      threadIdx.x
```

This is a 'kernel' or a function that executes on the device, but is called by the host.
the __global__ specifier makes it so. this function is compiled by nvcc into code that runs on the gpu.
this threadidx is a built in variable that the nvcc compiler puts into every kernel function. it is actually has 3 dimensional structure, a 2 dimensional 'grid' of 'blocks', and a thread index within the block.
Here's how we launch it. First we somehow make an array of N elements out on the GPU, we'll get to that, then we use this magic anglebracket syntax that NVCC preprocesses into a bunch of function calls to an nvidia written library that fetches the compiled code for the kernel from wherever its been stored, ships it out to the video card, and executes it however many times is specified with this grid/block stuff. From our perspective they run simultaneously, though they might be on different multiprocessors and whatever. When threads have to communicate with each other or synchronize access to shared memory things get more complicated but that's all outside the scope of this talk.

Adding a scalar to an array in cuda-parallel

```
__global__ void
add(float *data, float scalar)
{
  data[threadIdx.x] += scalar;
}
```


Adding a scalar to an array in cuda-parallel

`__global__`

`threadIdx.x`

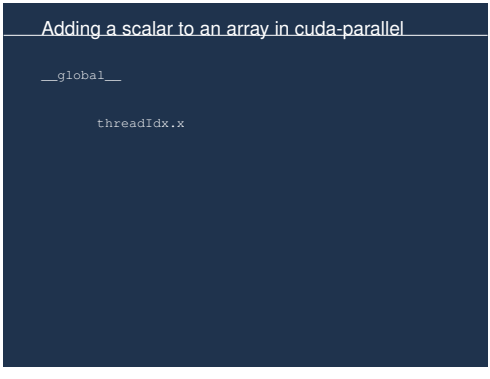
This is a 'kernel' or a function that executes on the device, but is called by the host.
the __global__ specifier makes it so. this function is compiled by nvcc into code that runs on the gpu.
this threadidx is a built in variable that the nvcc compiler puts into every kernel function. it is actually has 3 dimensional structure, a 2 dimensional 'grid' of 'blocks', and a thread index within the block.
Here's how we launch it. First we somehow make an array of N elements out on the GPU, we'll get to that, then we use this magic anglebracket syntax that NVCC preprocesses into a bunch of function calls to an nvidia written library that fetches the compiled code for the kernel from wherever its been stored, ships it out to the video card, and executes it however many times is specified with this grid/block stuff. From our perspective they run simultaneously, though they might be on different multiprocessors and whatever. When threads have to communicate with each other or synchronize access to shared memory things get more complicated but that's all outside the scope of this talk.

Adding a scalar to an array in cuda-parallel

```
__global__ void
add(float *data, float scalar)
{
  data[threadIdx.x] += scalar;
}

int main() {
  int N = 1024;
  float *arr = make_vector_on_gpu(N);

  add<<<1, N>>>(arr, 3.14159);
}
```

This is a 'kernel' or a function that executes on the device, but is called by the host.
the __global__ specifier makes it so. this function is compiled by nvcc into code that runs on the gpu.
this threadidx is a built in variable that the nvcc compiler puts into every kernel function. it is actually has 3 dimensional structure, a 2 dimensional 'grid' of 'blocks', and a thread index within the block.
Here's how we launch it. First we somehow make an array of N elements out on the GPU, we'll get to that, then we use this magic anglebracket syntax that NVCC preprocesses into a bunch of function calls to an nvidia written library that fetches the compiled code for the kernel from wherever its been stored, ships it out to the video card, and executes it however many times is specified with this grid/block stuff. From our perspective they run simultaneously, though they might be on different multiprocessors and whatever. When threads have to communicate with each other or synchronize access to shared memory things get more complicated but that's all outside the scope of this talk.

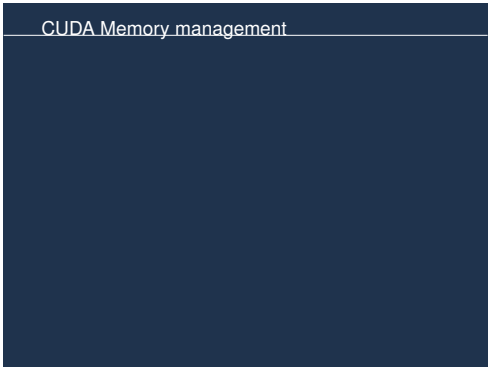
CUDA Memory management

```
cudaError_t cudaMalloc(void** devPtr, size_t count );
cudaError_t cudaFree(void* devPtr);

cudaError_t cudaMemcpy(void* dst, const void* src,
                       size_t count,
                       enum cudaMemcpyKind kind);

cudaMemcpyHostToHost
cudaMemcpyHostToDevice
cudaMemcpyDeviceToHost
cudaMemcpyDeviceToDevice

cudaError_t cudaMemset(void* devPtr, int value,
                       size_t count);
```

A holder class

```
template <typename T>
class holder
{
    T* devmem;
    std::size_T size_;

  public:

    holder();
    holder(std::size_t n);
    ~holder();
    boost::shared_ptr<holder> clone();
    void resize(std::size_t size);
    T* data() { return devmem; }
    std::size_t size() { return size_; }
};
```


A holder class

it isn't an array, it could be the underlying data used by an a ndimensional array

holder<T> implementations

```
template <typename T>
holder<T>::holder(std::size_t s) : size_(s)
{
   cudaMalloc(reinterpret_cast<void**>(&devmem),
              size_ * sizeof(T));
}

template <typename T>
holder<T>::~holder()
{
  if (devmem)
    cudaFree(devmem);
}
```


holder<T> implementations

```
template <typename T>
boost::shared_ptr<holder<T> >
holder<T>::clone()
{
   boost::shared_ptr<holder> nh(new holder(size_));
   cudaMemcpy(devmem, nh->devmem,
              sizeof(T) * size_,
              cudaMemcpyDeviceToDevice);
   return nh;
}
```


holder<T> implementations

```
template <typename T>
void
holder<T>::put(const T* hostmem, std::size_t s)
{
  if (s != size_)
    resize(s);
  cudaMemcpy(devmem, hostmem, s * sizeof(T),
             cudaMemcpyHostToDevice);
}

template <typename T>
void
holder<T>::get(T* hostmem)
{
  cudaMemcpy(hostmem, devmem, size_ * sizeof(T),
             cudaMemcpyDeviceToHost);
}
```


holder<T> implementations

now we just pass these things around at the end of shared pointers and we're good when for instance one array is a slice of another array

numpy arrays

```
>>> a
array([[  1.,   2.,   3.,   4.,   5.],
       [  6.,   7.,   8.,   9.,  10.],
       [ 11.,  12.,  13.,  14.,  15.],
       [ 16.,  17.,  18.,  19.,  20.]])
```


numpy arrays

```
          2.
[  6.,   7.,   8.,   9.,  10.],
         12.,  13.,  14.,
[ 16.,  17.,  18.,  19.,  20.]
```

numpy arrays

```
>>> a
array([[  1.,   2.,   3.,   4.,   5.],
       [  6.,   7.,   8.,   9.,  10.],
       [ 11.,  12.,  13.,  14.,  15.],
       [ 16.,  17.,  18.,  19.,  20.]])


>>> a[1,:]
array([  6.,   7.,   8.,   9.,  10.])

>>> a[3,::-1]
array([ 20.,  19.,  18.,  17.,  16.])

>>> a[:,1]
array([  2.,   7.,  12.,  17.])
```


numpy arrays

2.
[6., 7., 8., 9., 10.],
12., 13., 14.,
[16., 17., 18., 19., 20.]

array([6., 7., 8., 9., 10.])

array([20., 19., 18., 17., 16.])

array([2., 7., 12., 17.])

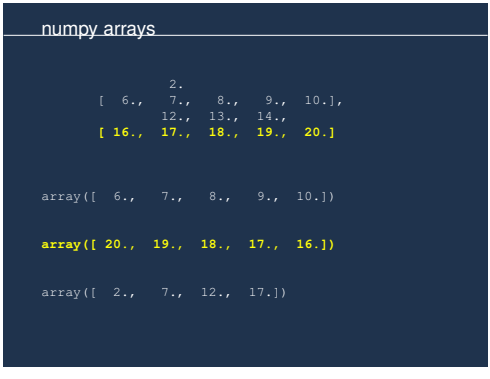
numpy arrays

```
>>> a
array([[  1.,   2.,   3.,   4.,   5.],
       [  6.,   7.,   8.,   9.,  10.],
       [ 11.,  12.,  13.,  14.,  15.],
       [ 16.,  17.,  18.,  19.,  20.]])


>>> a[1,:]
array([  6.,   7.,   8.,   9.,  10.])

>>> a[3,::-1]
array([ 20.,  19.,  18.,  17.,  16.])

>>> a[:,1]
array([  2.,   7.,  12.,  17.])
```

numpy arrays

```
>>> a
array([[  1.,   2.,   3.,   4.,   5.],
       [  6.,   7.,   8.,   9.,  10.],
       [ 11.,  12.,  13.,  14.,  15.],
       [ 16.,  17.,  18.,  19.,  20.]])


>>> a[1,:]
array([  6.,   7.,   8.,   9.,  10.])

>>> a[3,::-1]
array([ 20.,  19.,  18.,  17.,  16.])

>>> a[:,1]
array([  2.,   7.,  12.,  17.])
```


numpy arrays

```
              2.
       [  6.,   7.,   8.,   9.,  10.],
              12.,  13.,  14.,
       [ 16.,  17.,  18.,  19.,  20.]


array([  6.,   7.,   8.,   9.,  10.])

array([ 20.,  19.,  18.,  17.,  16.])

array([  2.,   7.,  12.,  17.])
```

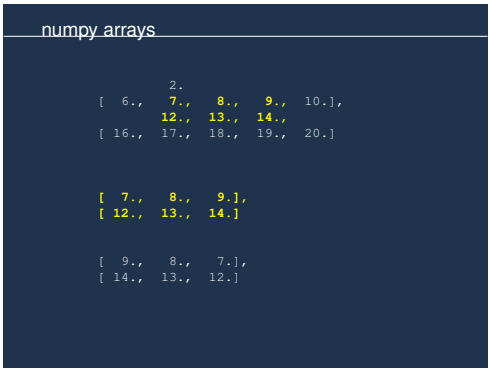
numpy arrays

```
>>> a
array([[  1.,   2.,   3.,   4.,   5.],
       [  6.,   7.,   8.,   9.,  10.],
       [ 11.,  12.,  13.,  14.,  15.],
       [ 16.,  17.,  18.,  19.,  20.]])


>>> a[1:3, 1:4]
array([[  7.,   8.,   9.],
       [ 12.,  13.,  14.]])

>>> a[2:0:-1, 3:0:-1]
array([[  9.,   8.,   7.],
       [ 14.,  13.,  12.]])
```


numpy arrays



numpy arrays

```
>>> a
array([[  1.,   2.,   3.,   4.,   5.],
       [  6.,   7.,   8.,   9.,  10.],
       [ 11.,  12.,  13.,  14.,  15.],
       [ 16.,  17.,  18.,  19.,  20.]])


>>> a[1:3, 1:4]
array([[  7.,   8.,   9.],
       [ 12.,  13.,  14.]])

>>> a[2:0:-1, 3:0:-1]
array([[ 9.,  8.,  7.],
       [ 14.,  13.,  12.]])
```


numpy arrays

```
                2.
[  6.,   7.,   8.,   9.,  10.],
          12.,  13.,  14.,
[ 16.,  17.,  18.,  19.,  20.]


[  7.,   8.,   9.],
[ 12.,  13.,  14.]


[  9.,   8.,   7.],
[ 14.,  13.,  12.]
```

PyCuda (Andreas Klöckner)

```
import pycuda.autoinit, pycuda.driver as drv, numpy

mod = drv.SourceModule("""
__global__ void multiply_them(float *dest, float *a, flo
{
  const int i = threadIdx.x;
  dest[i] = a[i] * b[i];
}
""")

multiply_them = mod.get_function("multiply_them")
a = numpy.random.randn(400).astype(numpy.float32)
b = numpy.random.randn(400).astype(numpy.float32)
dest = numpy.zeros_like(a)
multiply_them(
    drv.Out(dest), drv.In(a), drv.In(b),
    block=(400,1,1))
print dest-a*b
```


PyCuda (Andreas Klöckner)

```
at *b)
```

the pycuda approach uses cuda's jit engine.

you can use any of various templating engines to 'metaprogram'

Outline

CUDA

Related efforts

boost::proto
transforms

resophonic::kamasu
benchmarks

Outline

CUDA

Related efforts

boost::proto
transforms

resophonic::kamasu
benchmarks

Outline

CUDA

Related efforts

boost::proto
transforms

resophonic::kamasu
benchmarks

Outline

CUDA

Related efforts

boost::proto
transforms

resophonic::kamasu
benchmarks