cuda-on-cl

A compiler and runtime for running NVIDIA® CUDA™ C++11 applications on OpenCL™ 1.2 devices

Hugh Perkins (ASAPP)

Demo: CUDA™ on Intel HD5500

```
__global__ void setValue(float *data, int idx, float value) {
        if(threadIdx.x == 0) {
        data[idx] = value;
        }
}

int main(int argc, char *argv[]) {
    ...
    cudaMalloc((void**)(&gpuFloats), N * sizeof(float));
    setValue<<<dim3(32, 1, 1), dim3(32, 1, 1)>>>(gpuFloats, 2, 123.0f);
    cudaMemcpy(hostFloats, gpuFloats, 4 * sizeof(float), cudaMemcpyDeviceToHost);
    cout << "hostFloats[2] " << hostFloats[2] << endl;
    ...
}
```

```
/tmp/foo$ ./simple  2>/dev/null
Using Intel , OpenCL platform: Intel Gen OCL Driver
Using OpenCL device: Intel(R) HD Graphics 5500 BroadWell U-Processor GT2
building kernel _Z8setValuePfif
F name _Z8setValuePfif
 running generation on _Z8setValuePfif
building kernel _Z8setValuePfif
 ... built
hostFloats[2] 123
/tmp/foo$
```

Background: why?

- NVIDIA® CUDA™ is the language of choice for machine learning libraries:
  - Tensorflow
  - Caffe
  - Torch
  - Theano
  - …

- Ports to OpenCL by hand include
  - Caffe (Tschopp; Gu et al; Engel)
  - Torch (Perkins) <= me :-)

- Dedicated OpenCL libraries are few:
  - DeepCL (Perkins)

Why not port by hand?

- Maintenance nightmare
- Need to fork the code
    - The Caffe forks are separate from core CUDA codebase
    - The Torch fork is a separate repo from core CUDA Torch codebase
        - Feature incomplete
        - Frozen at February 2016

Concept: leave the code in NVIDIA® CUDA™

- Leave the code in NVIDIA® CUDA™
- Compile into OpenCL

NVIDIA® CUDA™ compiler ecosystem

What	Who	Input	Portable?	Compile/run NVIDIA® CUDA™?
HIP	AMD	HIP	No (AMD only)	Almost (rename API calls)
ComputeCpp	Codeplay	SYCL	Yes (SPIR)	No (different api)
triSYCL	Keryell	SYCL	Yes (SPIR)	No (different api)
NVIDIA CUDA	NVIDIA	NVIDIA® CUDA™	No (NVIDIA only)	Yes

NVIDIA® CUDA™ compiler ecosystem

What	Who	Input	Portable?	Compile/run NVIDIA® CUDA™?
HIP	AMD	HIP	No (AMD only)	Almost (rename API calls)
ComputeCpp	Codeplay	SYCL	Yes (SPIR)	No (different api)
triSYCL	Keryell	SYCL	Yes (SPIR)	No (different api)
NVIDIA CUDA	NVIDIA	NVIDIA® CUDA™	No (NVIDIA only)	Yes
cuda-on-cl	**Perkins**	**NVIDIA® CUDA™**	**Yes (OpenCL 1.2)**	**Yes**

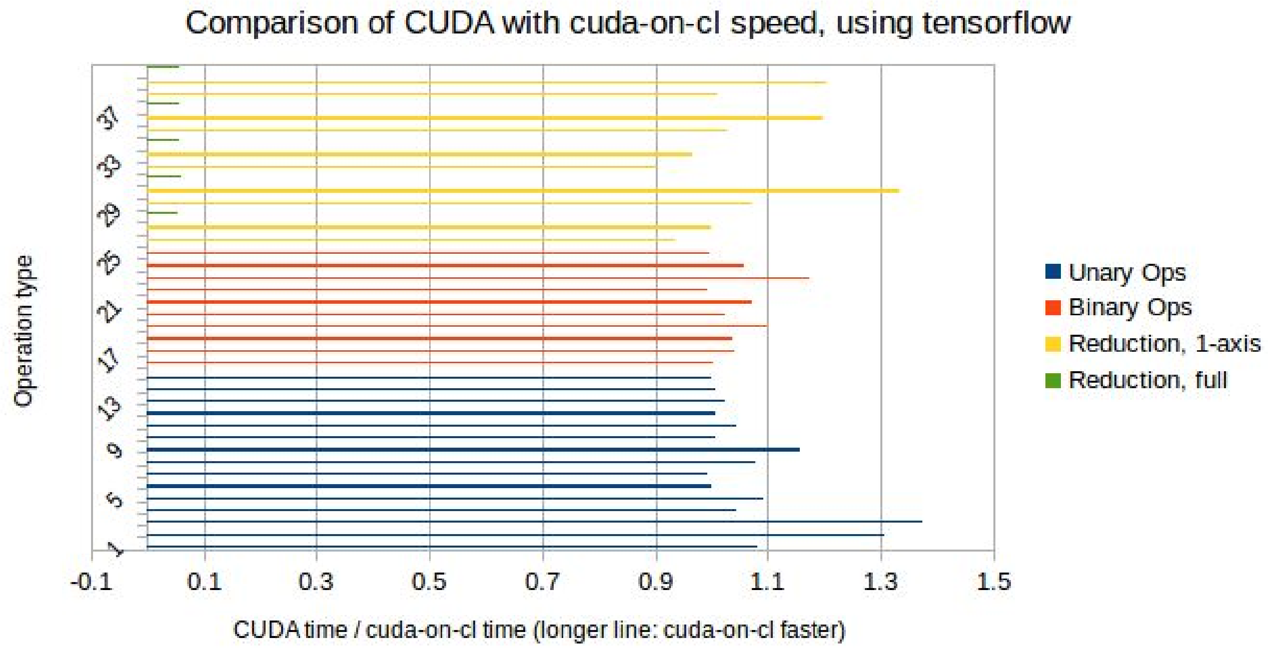
Portability vs speed

Speed vs portability: pick one

Portability ⟷ Speed

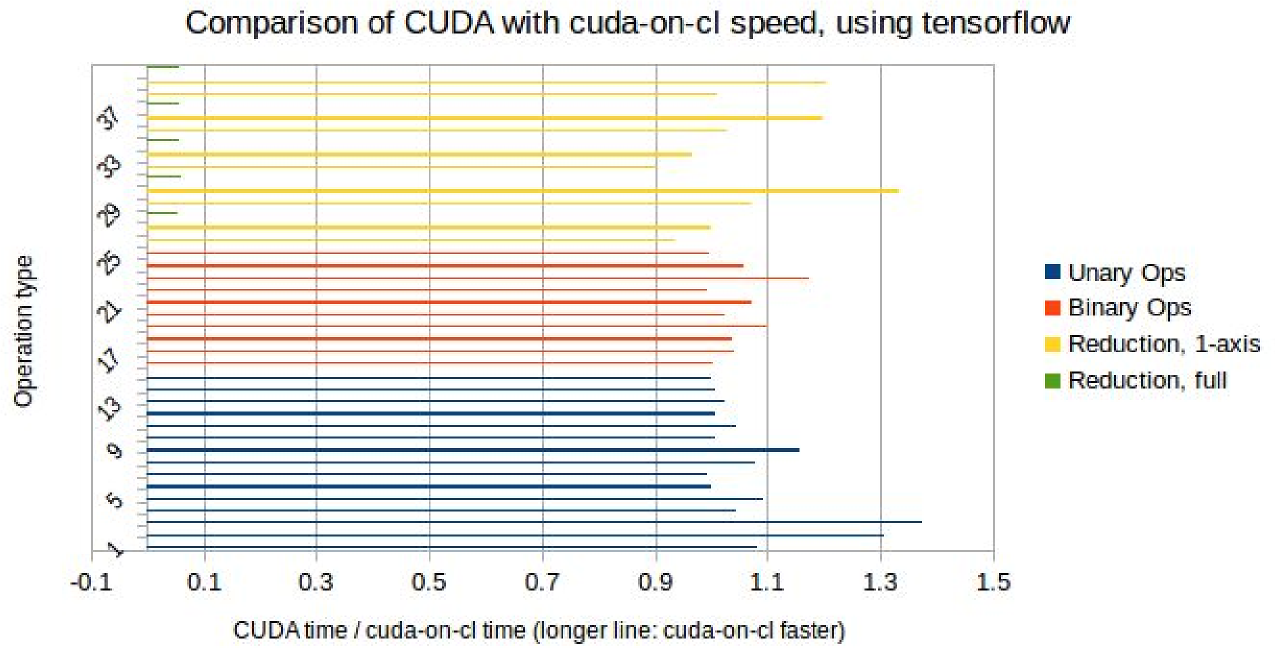
CUDA-on-CL	triSYCL / ComputeCpp	HIP	NVIDIA CUDA
OpenCL 1.2 is widely supported	SPIR-1.2 not widely supported	One single hardware type, so fast, but not portable	One single hardware type, And its NVIDIA :-)

So how fast is cuda-on-cl?



NVIDIA K80 GPU
Batch size: ~500MB

So how fast is cuda-on-cl?



Execution times comparable for:

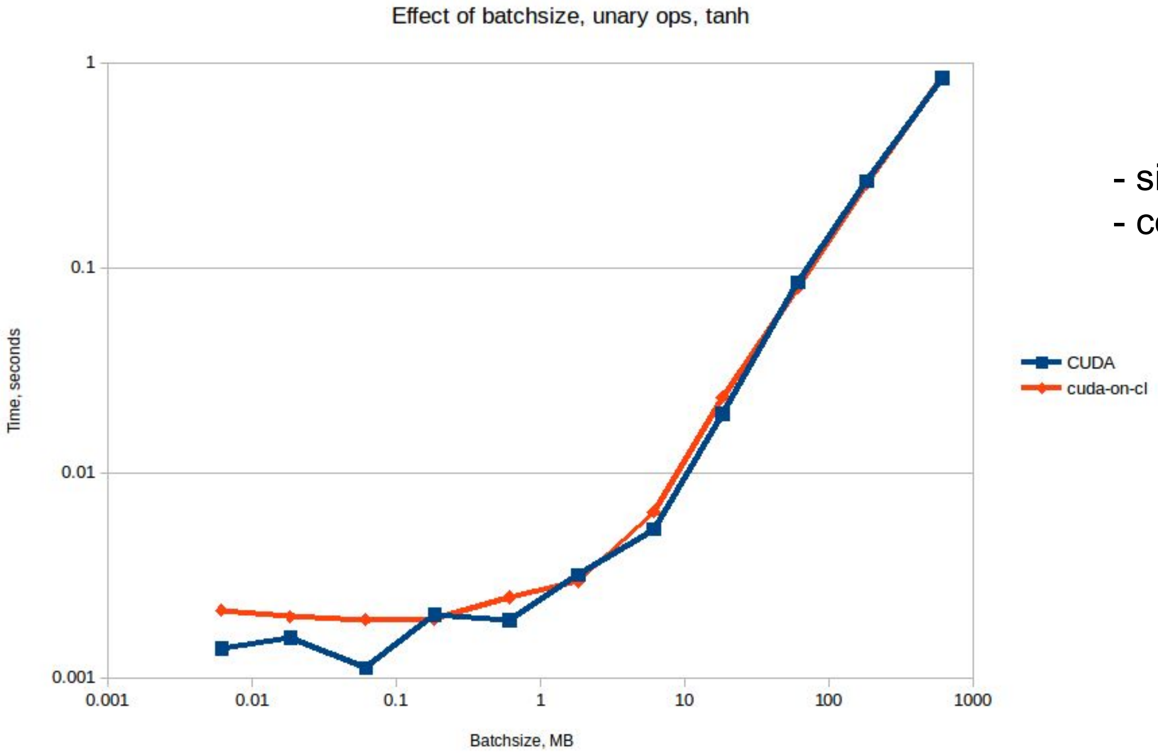
- unary ops
- binary ops
- single-axis reduction

But:

- full reduction slow
- large batchsizes

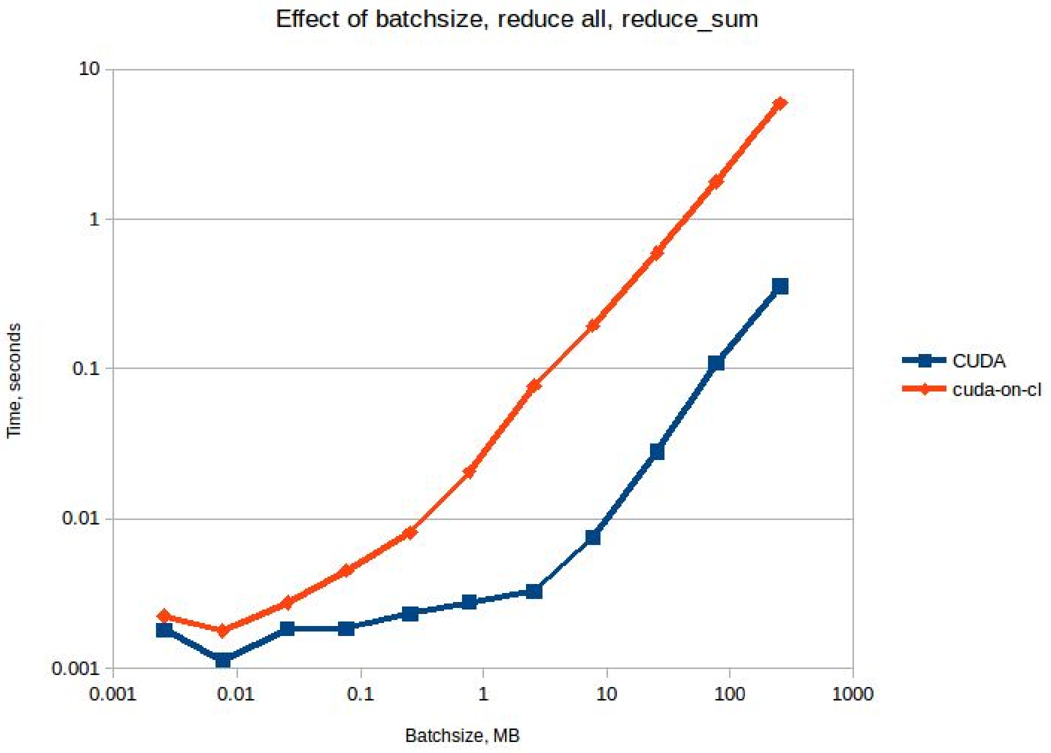
NVIDIA K80 GPU
Batch size: ~500MB

Effect of batchsize on execution time, unary ops



- similar for batchsize >= 1MB
- constant per-batch overhead higher

Effect of batchsize on execution time, full reduction



- full reduction 14 times slower
- open opportunity to analyze why

Key design decisions

- We want to compile C++11 kernels. How?

  Use CLANG C++11 parser

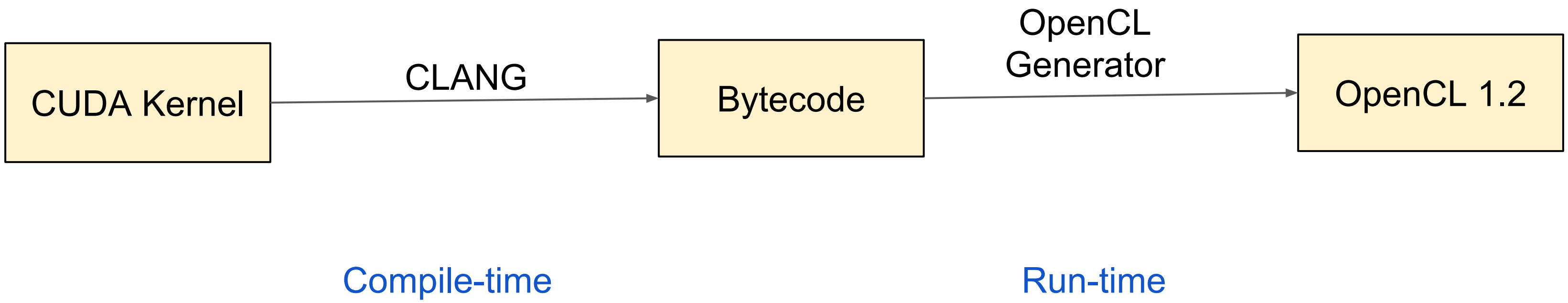
- How to feed bytecode to the GPU driver?

  Convert to OpenCL 1.2

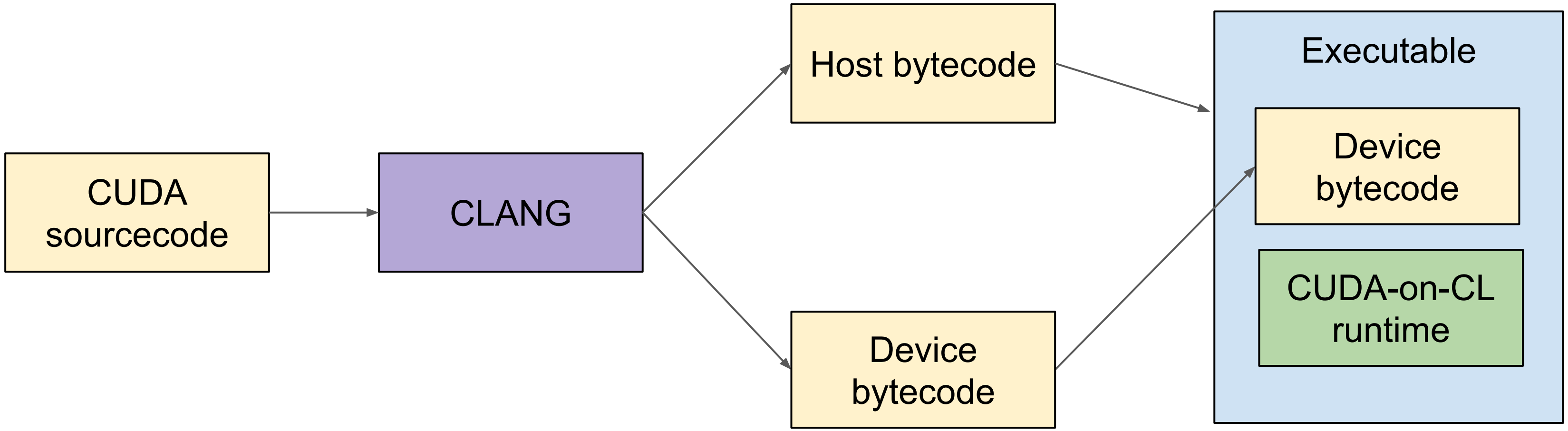
- How to handle NVIDIA® CUDA™ API calls?

  Implement NVIDIA® CUDA™ API, in OpenCL

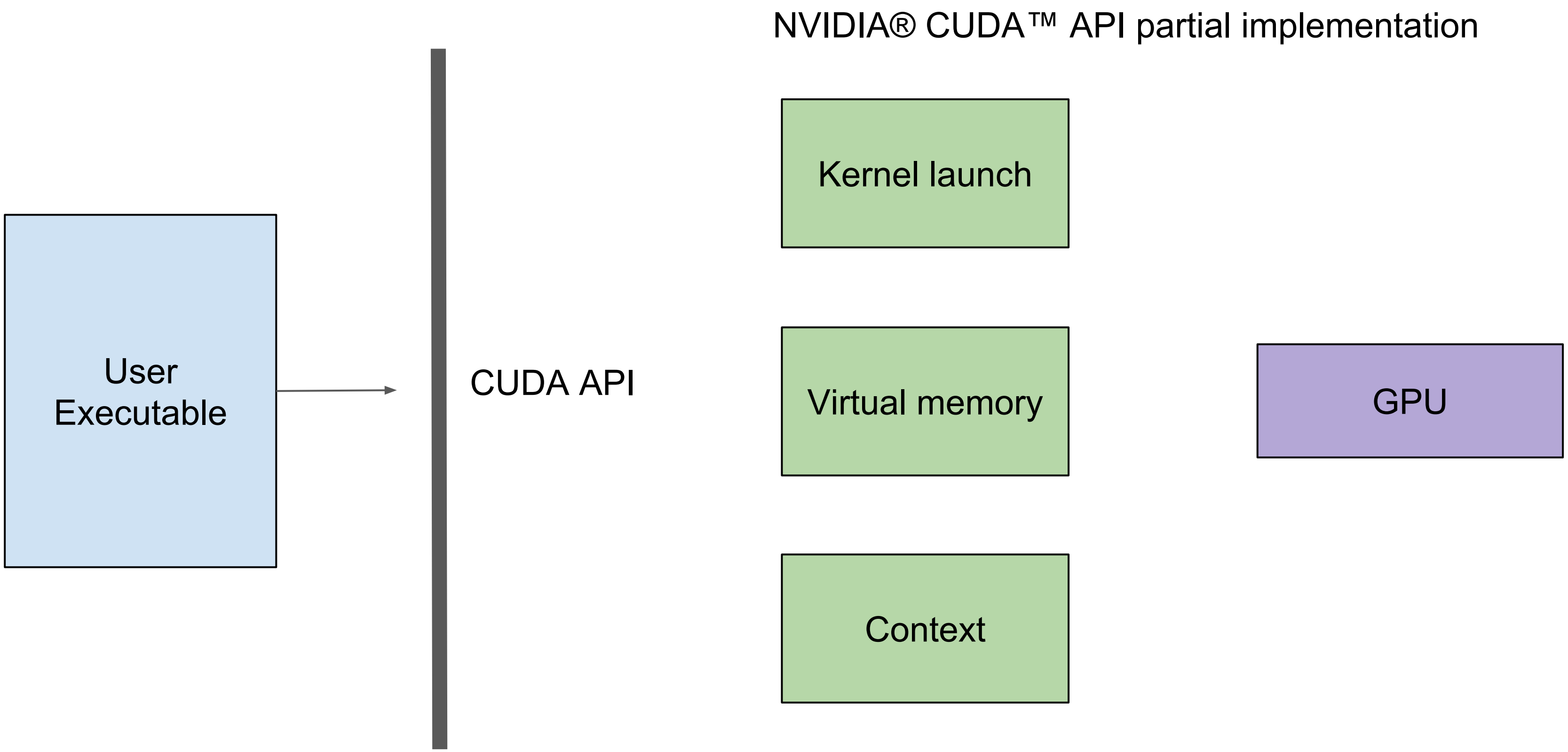
Kernel compilation



Compile-time



Run-time


NVIDIA® CUDA™ API partial implementation
User Executable
CUDA API
Kernel launch
Virtual memory
Context
GPU

Edge/not-so-edge cases

OpenCL 1.2:

- **does not allow hostside GPU buffer offsets <= we will look at this**
- requires address-spaces to be statically declared (global/local/private…)
  - … including function parameters
  - … which might be called with diverse address-space combinations
- forbids by-value structs as kernel parameters
- forbids pointers in kernel parameter structs
- lacks many hardware operations, eg `__shfl__`

CUDA-on-CL handles all of the above

Case-study: hostside GPU buffer offsets

CUDA lets you do things like:

```
float *buf = (float *)cudaMalloc(1024);
someKernel<<<... >>>(buf + 128);
```

Case-study: hostside GPU buffer offsets

OpenCL 1.2 doesn't allow this:

```
cl_mem buf = clCreateBuffer(..., 1024, ...);
clEnqueueNDRangeKernel(... , buf + 128, ...);
```

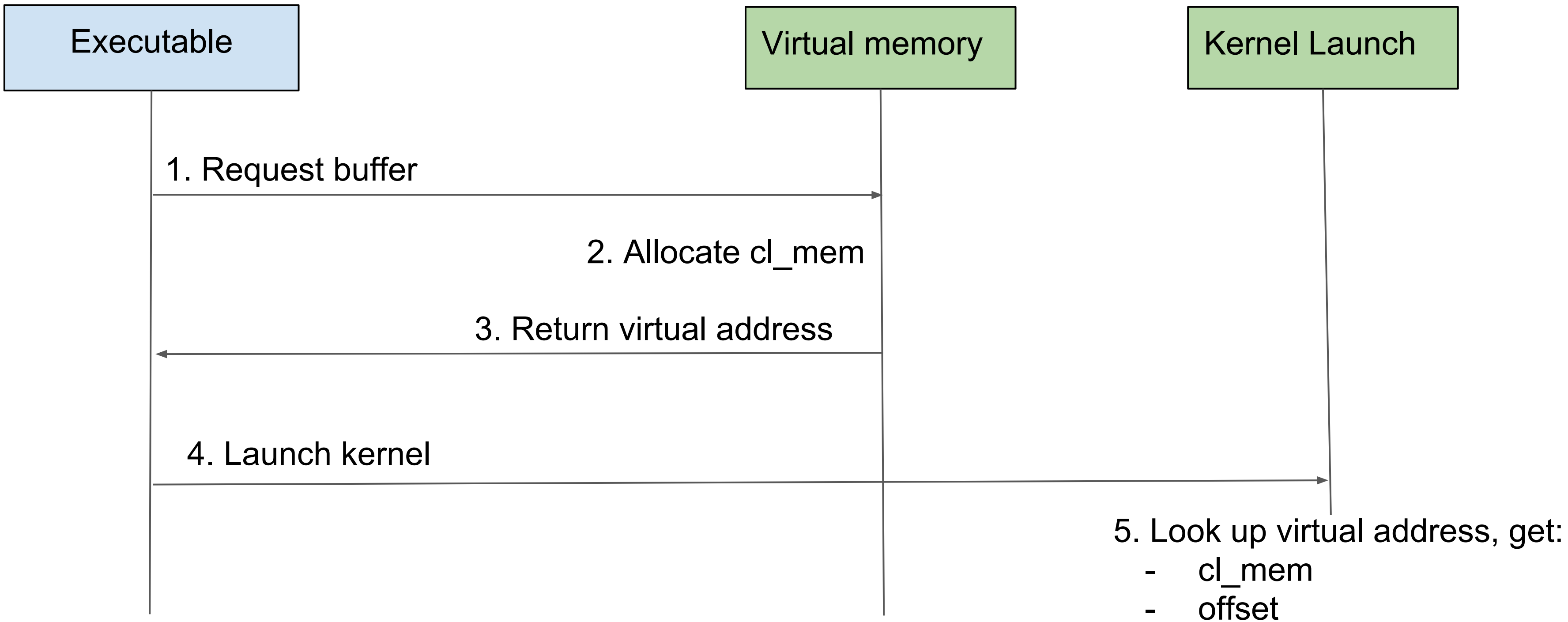
Not allowed

Case-study: hostside GPU buffer offsets

CUDA-on-CL solution:

1. Implement virtual memory, and
2. Rewrite the kernel

Part 1: virtual memory



Part 2: rewrite kernel

Before:

```
kernel void someKernel(global float *buf, …) {
    ...
}
```

After:

```
kernel void someKernel(global float *buf_data, int buf_offset, …) {
    global float *buf = buf_data + buf_offset;
    ...
}
```

Transparent: no changes required to user source-code

Open issues

- Execution speed:
  - NVIDIA compiler optimizations really good
  - OpenCL 1.2 compatibility boilerplate increases launch overhead
  - OpenCL intrinsic kernel launch time high
  - Missing hardware implementations (`shfl` etc)

- Portability:
  - Each vendor driver has different quirks
    - Need to test case-by-case
    - CUDA-on-CL stresses the drivers in unusual ways

Overall

- CUDA-on-CL actually works, on some fairly complex kernels
- Runs on multiple vendors' GPUs
- Execution speed can be at parity with native NVIDIA® CUDA™
- Much more general solution than porting by hand

https://github.com/hughperkins/cuda-on-cl

Thank you to Andy Maheshwari (ASAPP) for awesome help reviewing the presentation.