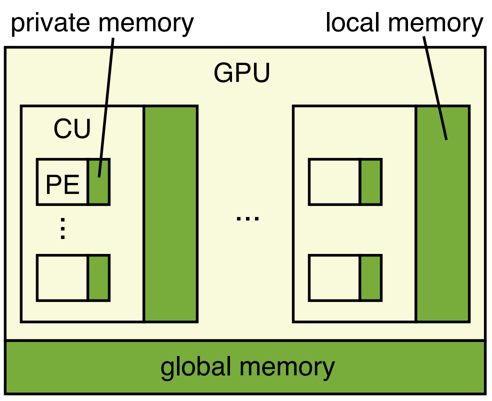
Formalising GPU memory models

John Wickerson [1] / Tyler Sorensen [2,3] / Daniel Poetzl [2,4] / Mark Batty [5] / Jade Alglave [2] / Alastair Donaldson [1]

[1] Imperial College London / [2] University College London / [3] University of Utah / [4] University of Oxford / [5] University of Cambridge

GPUs are great. Graphics processing units (GPUs) pack hundreds of processing elements onto a single chip. They offer tremendous compute power at a low cost. Originally intended only for graphics computations, they also excel at such tasks as medical imaging, computational finance, and molecular simulation - often beating CPU performance by orders of magnitude.

GPUs are complex. Modern CPUs run several threads concurrently, and give each access to a shared memory. GPUs are more structured. Each thread is run on a processing element (PE), each of which has its own *private memory*. These PEs are organised into several *compute units* (CUs), each of which has its own *local memory*. The collection of threads running on one CU is called a *workgroup*. The device also maintains a *global memory*.



Programming GPUs is hard.

Programming models such as OpenCL and CUDA allow programmers to harness the power of GPUs. Besides the complexities of GPU hardware discussed above, these languages also provide a variety of instructions – such as *barriers*, *atomic* operations, and *fences* – that enable expert programmers to ensure that high-performance code behaves correctly. The programmer must choose, for each instruction, whether to apply it to the local memory or the global memory, and whether to propagate its effects to the whole device or only within the current CU. The opportunities for confusion are almost endless.

The memory model is supposed to help... GPU programming languages such as OpenCL and CUDA provide a *memory model*. This is intended to be a precise description of how the GPU's various memories store and load values, and how the programmer can write 'good' programs that avoid certain problems like *data races*.

...but it's not good enough.

These memory models are described in prose. Although they are very detailed, they ultimately suffer from ambiguities, contradictions and omissions, like all natural language texts. How can programmers be confident about the behaviour of their programs when the standard is so vague?

Our aim is to translate several GPU memory models into the rigorous language of *mathematics*.

We will...

- develop a *generic framework* for GPU memory models, and build two instantiations of it: one for NVIDIA's low-level PTX language and one for the higher-level OpenCL 2.0 language;
- write our models using a modelling language such as *Herd* or *Lem*;
- test our models by simulating the execution of some small programs, and then refine the models if the results do not agree with the prose;
- ask the authors of the standards for clarification when ambiguities arise;
- when designing our PTX model, run experiments to see how the real NVIDIA hardware behaves; and
- evaluate the soundness of our OpenCL model by devising an OpenCL-to-PTX translator, and then confirming that OpenCL programs deemed 'good' in our OpenCL model are always translated to PTX programs deemed 'good' in our PTX model.

What do these programs do?

When designing our mathematical models, we study the behaviour of small programs called *litmus tests*. Here are two. In the first, which final values of `x` and `y` are possible?

```
global atomic x=0, local atomic y=0;

// THREAD 1:
int t = load(&x,acquire);
store(&y,t,release);

// THREAD 2 (in same workgroup):
int u = load(&y, acquire);
store(&x,u,release);
```

In the second, can `t` end up 0?

```
global atomic x=0, global atomic y=0;

// THREAD 1:
store(&x,1,relaxed,workgroup);
store(&y,1,release,device);

// THREAD 2 (in different workgroup):
if (load(&y,acquire,device)==1)
 int t=load(&x,relaxed,workgroup);
```

We acknowledge financial support from the EPSRC and the CARP FP7 project.