CMP SCI 635: Modern Computer Architecture

Dynamic Thread Block Launch: A Lightweight Execution Mechanism to Support Irregular Applications on GPUs

Name: Kunjal Panchal Date: 19th Nov, 2019 Student ID: 32126469

Paper: 20 – Data Parallel [Wang15]

Strengths:

1. CUDA devices are fine at rather random access – for example, parallel random access to data small enough to fit in, and explicitly allocated at, what Nvidia calls "shared memory". We get problems if you want random access to swaths of data (though I think highest-end devices come with caches for that, so they'd be tied with CPUs at some point perhaps). In this sense, the accelerators are not necessarily better. We get fine random access when you hit the per-core/per-cluster explicitly managed local memory, worse latency for nearby local memories, and no data cache at all to speed up access to DRAM.
2. The possibility to coalesce work-items into single (vectorized) threads is one of the powerful aspects of OpenCL. Even if the hardware isn't vector-capable, there's a few workloads for which vectorization helps even on the scalar GPUs, since it gives better bandwidth utilization and thus improves the computation to memory transaction ratios. [But it's probably not worth the effort at the compiler level.]

Weaknesses:

1. Suppose I have two OpenCL kernels trying to run at the same time. One kernel does my post-processing and needs to be run per frame. Another kernel number crunches and takes half a second to complete. It doesn't have to be complete every frame, but it will always be running. I also have 3d graphics being rendered via OpenGL. If OpenGL is fighting these kernels and is waiting for the second kernel to complete before displaying anything, us trying to break up the second kernel into many more blocks hoping that the OpenGL will man-up, starvation can happen.
2. If we want in-GPU fake memory allocations, a simple atomic integer counter is enough to get new fake allocated memory parts to each work item. But object size in GPU memory is not well defined so we have to pack each of them by the next power-of-2 size and get biggest members on top of its struct and smallest to the bottom just for efficiency. Wouldn't that incur a high overhead in dynamic parallelism?

Questions/Assertions:

1. GPGPU means doing your general-purpose stuff under the constraints imposed by the GPU architecture. An OpenCL accelerator lifting some of these constraints could be very welcome. This is about making something that's more efficient than CPUs, but as nice and flexible as possible, and more flexible than GPUs.
2. Such a faster than CPUs and more flexible than GPUs accelerator can give us and advantage that GPUs don't: One important feature is divergent flow. GPUs are SIMD or SIMT hardware; either way, they can't efficiently support something like this:

```
if(cond(i)) {
  out[i] = f(i);
}
else {
  out[i] = g(i);
}
```

   What they'll end up doing is, essentially, compute f(i) and g(i) for all values of i, and then throw away some of the results. For deeply nested conditionals, the cost of wasted computations can make the entire exercise of porting to a GPU pointless.
3. Most GPUs can only run a single kernel at a time. This, by the way, includes both display and compute. This is actually the reason why a watchdog is enabled when the GPU is attached to a display: long-running compute kernels get terminated if the display stuff can't run for too long. Some modern GPUs can run concurrent kernels, but the details on how this works depends on the vendor.
4. The safest bet if we want to run concurrent kernels and we know that our device can (and the driver supports it) is to ensure our launch grid is composed by “few” work-groups only. This of course means that we might need to redesign our kernel to use a fixed number of work-groups (and processing our data piece-wise) rather than the traditional approaches of one work-item per data element.
5. Is there a way to compile your kernels before you load them during execution? I see there's this CLCC compiler but it hasn't been updated in a year and doesn't even support 1.2. AMD has one but that only works on AMD devices. If there isn't any dedicated compiler for kernels, what happens if I have a syntax error in my kernel? By forcing a syntax error in the kernel, won’t I still be able to run the program without anything complaining. I would just get junk output obviously. Is there a way to catch stuff like that before running? Is it possible to run a debugger through my kernel?