Generating OpenCL Kernels using Decompilation

Summary

- OpenCL is a framework for developing applications that execute across a range of device types (multicore x86, GPUs, FPGAs).
- It is gathering a lot of interest/momentum as a parallel programming platform.
- It uses a string representation for the kernels describing the OpenCL computations.
- This has a number of disadvantages, including difficulty of use and inefficiencies.
- An approach based on program decompilation might provide an alternative route to developing some kernels and does not suffer from these disadvantages.

OpenCL

- The Open Computing Language (OpenCL) is a framework for developing parallel applications that can be mapped to a range of target architectures.
- OpenCL includes a language (based on C99) for writing kernels (functions that execute on OpenCL devices), plus APIs that are used to define and then control the platforms.
- To allow kernels to be optimally compiled to the resources available at runtime they are represented as strings in the application.
- Has the added advantage that the host language, e.g. C++, does not need to know anything about the kernel language.
- An OpenCL device driver compiles these strings into platform-specific code at runtime. This distinguishes OpenCL from other more common approaches that strive for platform independence by using virtual machine code and just-in-time compilers.
- Here is an example kernel for vector addition:

```
const char* programSource =
  "__kernel                          \n"
  "void vecadd(__global int *A, \n"
  "            __global int *B, \n"
  "            __global int *C) \n"
  "{                                 \n"
  "  int idx = get_global_id(0; \n"
  "  C[idx] = A[idx] + B[idx];  \n"
  "}                                 \n"
;
```

Deficiencies

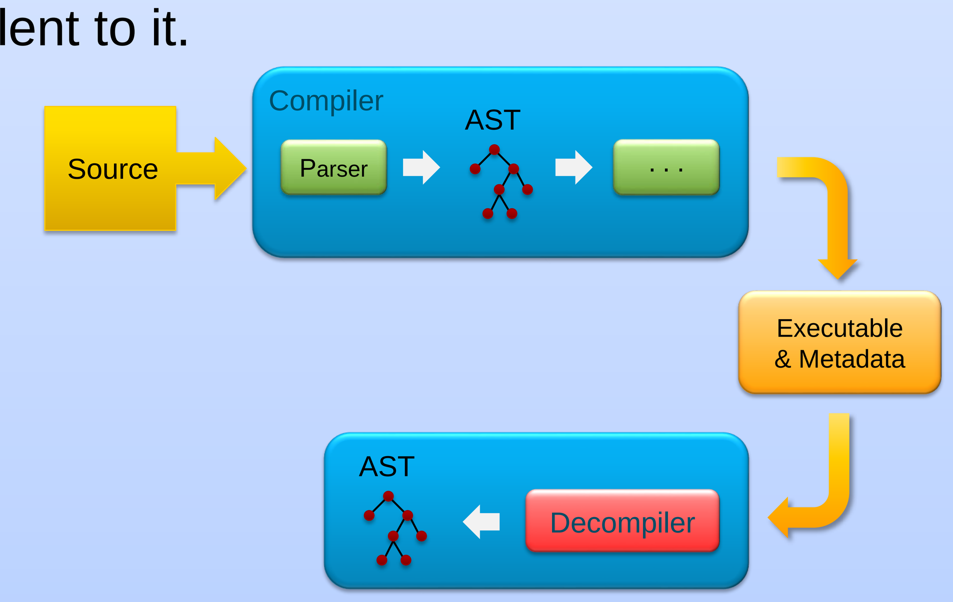
- Describing multiline string constants is tedious in many languages. Developers often write the kernels in separate files and then read the contents into strings at runtime.
- IDE support no longer available to us when developing kernels, including compile-time type-checking, syntax highlighting, and autocompletion.
- May decrease performance in some cases due to overheads.
- Not really platform-independent; even something simple like a parallel reduce may need to choose between different kernels depending on platform.

Parallel Patterns

- A lot of algorithms can be described by combining a set of common patterns, for example
  - Map
  - Reduce
  - Map-reduce
  - Adjacent difference
  - Scan (a.k.a. parallel prefix)
  - Searching
- We must parameterise each pattern with one or more user-defined functions.
- In OpenCL we must combine a string-based representation of the pattern with a string-based representation of the user function to produce the source for the kernel.
- For small problem sizes it may be faster to use a task-based native approach (less overheads).
- So we might need ***two*** versions of the user function, a native version and a "string" version; this is both tedious and error-prone.

Decompilation

- Given an executable, e.g. a .NET assembly, we can use the metadata, plus heuristics, to decompile selected methods back to a high-level abstract syntax tree (AST) representation.
- Not necessarily identical to the original, but semantically equivalent to it.



- Requires reflection/metadata, so not practical for C++.
- We can decompile the function arguments to our patterns to generate kernels and call them when appropriate.
- Use native version when required, e.g. data too small or functions too complex to convert to an OpenCL kernel.

Example

```
Func<float, float, float> f = // Some random function, e.g.
  (float a, float b) =>
    (float)Math.Sqrt(Math.Abs(Math.Sin(a) + Math.Cos(b)));

// Use C# implementation with task-based parallelism if appropriate
Map2(f, arrA, arrB, arrC);

// Use either C# or OpenCL implementation based on heuristics etc.
Map2(f, arrA, arrB, arrC, context);
```

OpenCL context

- We automatically generate the appropriate OpenCL kernel string if required, e.g.

```
kernel void K(
    global /*  read_only */ float* a,
    global /*  read_only */ float* b,
    global /* write_only */ float* c )
{
    int index = get_global_id(0);
    c[index] = sqrt (fabs (sin (a [index]) + cos (b [index])));
}
```

- We can use "lazy" operators to delay the computations until the result is required.
- Allows us to aggregate computations, e.g. map of a map of a …
- Produces more compute-intensive OpenCL kernels.

Status

- Small prototype developed as proof of concept.
- Would need to develop a robust decompiler and translator to OpenCL kernels.
- For each pattern we would need to develop a family of efficient kernels to exploit each platform.
- Would require the development of heuristics, perhaps an autotuner, to choose when to use OpenCL, and when to use native code.
- Currently a "10% project".
- Could potentially be developed further if there was sufficient interest.