FORMALIZING ADDRESS SPACES WITH APPLICATION TO CUDA, OPENCL, AND BEYOND

Benedict R. Gaster* and Lee Howes
AMD

* author is now at Qualcomm
DATA LOCALITY

Data-locality plays an important role in an applications performance, e.g.:

NUMA
Caches (temporal and spatial)
Address Spaces

the subject of this talk
ADDRESS SPACES

Address spaces explicitly manage where data lives during execution

Originally standardized in Embedded C

Popularized in modern GPGPU languages:

CUDA (not formalized as part of the type system)
OpenCL (formalized as part of the type system)
OPENCL 1.X MEMORY HIERARCHY

- Kernel
  - Global Memory
    - Constant Memory
  - Local Memory
    - Work Item
      - Private
    - Work Item
      - Private
    - Work Item
      - Private

Levels:
- Kernel-wide scope
- Work group scope
- Work item scope
kernel void vscale(
    global int * C,
    global int * A,
    const global int * S)
{
    C[get_global_id(0)] = A[get_global_id(0)] * S[get_group_id(0)];
}
OPENCL ADDRESS SPACES

All pointers in an OpenCL program must be assigned an address space
Lacks the ability to parameterize over address spaces.
int scale(global int * A, global int * S);
SCALE VECTOR USING ABSTRACTION

kernel void vscale(
    global int * C,
    global int * A,
    const global int * S)
{
    C[get_global_id(0)] = scale(&A[get_global_id(0)], &S[get_group_id(0)]);
}
OPTIMIZE SCALING CONSTANTS TO ON-CHIP MEMORY

kernel void vscale(
    global int * C,
    global int * A,
    constant int * S)
{
    C[get_global_id(0)] = scale(&A[get_global_id(0)], &S[get_group_id(0)]);
}
OPTIMIZE SCALING CONSTANTS TO ON-CHIP MEMORY

kernel void vscale(
    global int * C,
    global int * A,
    constant int * S)
{
    C[get_global_id(0)] = scale(&A[get_global_id(0)], &S[get_group_id(0)]);
}

No longer type checks, i.e.

constant ∪ global

is not valid…
Introduce an address space, *generic*, that subsumes all others
GENERIC ADDRESS SPACE
int scale(generic int * A, generic int * B);
int scale(int * A, int * B);
DOES GENERIC REQUIRE HARDWARE SUPPORT?

OpenCL C + generic

```c
global int * g_ptr;
int x = *g_ptr;

int * ptr = g_ptr;
x = *ptr;

local int * l_ptr;
x = *l_ptr;

g_ptr = l_ptr;
x = *g_ptr;
```

Pseudo IR + generic

```c
int * g_ptr __attribute__((global))
int * ptr __attribute__((generic))
int * l_ptr __attribute__((local))
int x;

x = load_global(g_ptr);
ptr = g_ptr;
x = load_generic(g_ptr); // global mem load
x = load_local(l_ptr);
g_ptr = l_ptr;
x = load_generic(g_ptr); // local mem load
```
CAN’T THE COMPILER DEDUCE WHAT TYPE OF LOAD GENERIC IS PERFORMING?

Maybe using Hidley-Milner type inference [1,2]?
SADLY

In general it is not possible!
EXAMPLE WHY HIDLEY-MILNER FAILS

```c
void foo(int *);
kernel void bar(global int *g, local int *l)
{
    generic int * tmp;
    if (get_global_id(0) % 2) {
        tmp = g;
    } else {
        tmp = l;
    }
    foo(tmp);
}
```
tmp is **global** and **local** for different work-items at the point `foo(tmp)`
GENERICS ARE VARIANT (OR SUM) TYPES

global + local int *
A pointer instance within the generic address space can only point to one disjoint address space:

- global
- constant
- local
- private

at any given time.
THIS PAPER

Describes a type system that:

- combines the parametric polymorphism of generics with variant address spaces
- defines a type-inference algorithm that can infer parametric polymorphic variant address spaces types, for all valid programs, or fails
- defines a runtime implementation for generic address:
  - zero overhead for targets with hardware support for generic
  - overhead only in the presence of indirect functions with generic arguments
Our system is based on the general theory of qualified types [3] Extended with the notion of variants [4]

Originally developed in the context of Haskell

class Eq a where
  (==) :: a -> a -> a

instance Eq Int where
  x == y = eqInt x y

(==) : forall a . Eq a => a -> a -> a

eqInt : Int -> Int -> Int
ADDRESS SPACE ARE DEFINED IN TERMS OF ROWS AND A CONSTRUCTOR

\{a_1, \ldots, a_n \mid r\} = \{a_1 \mid \ldots \{a_n \mid r\} \ldots \}
\{a_1, \ldots, a_n \} = \{a_1 \mid \ldots \{a_n \mid \} \ldots \}

A pointer of type τ in some address space a and some yet to be determined address spaces ranged over by r, is represented by the type:

\text{ASpace} \{a \mid r\} τ *
DEFINITION (INJECTION) WITH INITIALIZER

Generic address space:

\[ \tau \times x :: r \Rightarrow \text{size}_t \rightarrow \text{Aspace } r \ \tau \times \]

\[ \text{int} \times x = 0xffffffff; \]

Disjoint address space a:

\[ a \ \tau \times :: (r \setminus a) \Rightarrow \text{size}_t \rightarrow \text{Aspace } \{a \ | \ r\} \ \tau \times \]

\[ \text{global int} \times x = \text{NULL}; \]
ASSIGNMENT (INJECTION)

_ = _ :: (r \ a) \Rightarrow A\{ a | r \} \tau^* \\
  \rightarrow A\{ a | r \} \tau^* \\
  \rightarrow A\{ a | r \} \tau^* \\

global int * g_ptr; // disjoint definition (injection)
int * g;            // generic definition (injection)
int * ptr = g_ptr;  // assignment (injection)
ASSIGNMENT (EMBEDDING)

_ = _ :: (r \ a ) ⇒ Aspace r τ *
→ ASpace{a | r } τ *
→ ASpace{ a | r } τ *

global int * g_ptr; // disjoint definition (injection)
local int * l_ptr;  // disjoint definition (injection)
int * ptr;          // generic definition (injection)

if (…) {
    ptr = g_ptr;    // assignment (embedding)
} else {
    ptr = l_ptr;   // assignment (embedding)
}
LOAD (STORE IS SIMILAR)

\[ \text{ld}(\_ ) :: (\{} \setminus a) \Rightarrow \text{ASpace}\{ a \} \tau^* \rightarrow \tau \]

\[ \text{ld}_a(\_,\_) :: (r \setminus a) \Rightarrow (\text{ASpace}\{ a \} \tau^* \rightarrow \tau) \rightarrow \text{ASpace}\{ a \mid r \} \tau \rightarrow \tau \]

\[ \text{ld}(\_,\_,\_,\_) :: r \setminus a \Rightarrow (\text{ASpace}\{ \text{global} \} \tau^* \rightarrow \tau) \]
\[ \quad \rightarrow (\text{ASpace}\{ \text{local} \} \tau^* \rightarrow \tau) \rightarrow \]
\[ \quad (\text{ASpace}\{ \text{private} \} \tau^* \rightarrow \tau) \rightarrow \]
\[ \quad \text{ASpace} \{ r \mid a \} \tau^* \rightarrow \tau \]
\[ \quad \rightarrow \tau \]
EXAMPLE

kernel void x(
    global * int g,
    local * int l,
    int value)
{
    int * var = 0;
    if (value % 2) {
        var = g;
    } else {
        var = l;
    }
    *g = *var;
}

kernel void x(
    ASpace { global } * int g,
    ASpace { local } * int l,
    int value)
{
    ASpace r int * var = 0;
    if (value % 2) {
        var = g;
    } else {
        var = l;
    }
    store_global(g,
        ld(var, ld_global, ld_local, ld_private));
}
THE DETAILS

The paper provides details of

1. the type inference algorithm

2. how predicates are used as ‘evidence’ to determine the address for a particular instance of a value within the generic address space domain
CONCLUSION

Formalized the notion of generic address spaces for OpenCL, Cuda, etc.

Naturally extends to languages such as C++
   As seen in the earlier OpenCL C++ paper

Formalizes Embedded C’s notion of generic address space
   Provides the ability to extend embedded C to C++

Type inference algorithm has potentially many other applications:
   e.g. scalar/vector usage of OpenCL C programs
REFERENCES


