Automated Analysis of Complex Data Objects for Supporting Heterogeneous Platforms

Sebastian Dreßler
Thomas Steinke

Zuse Institute Berlin
Motivation

- implementations with dynamic and non-linear data types
  - irregular, e.g. linked lists, multiple inherited classes
  - non-contiguous in memory
  - size not known at compile time

- processes each running on devices with their own physical memory
  - host + accelerator platform = heterogeneous system
  - offload kernel functions to accelerator device
  - data management (allocation, transfers between memory spaces)

→ For data transfers serialization methods are needed!
1: ALLOCATE

2: SERIALIZE
(copy sub-arrays)

gap?
High-Level Approaches

- Compiler directives
  - OpenACC
  - Intel Offload model
  - OpenMP 4

- YES, but ...
  - Still needs manual work
  - Are non-linear objects supported?
Proposal: a 3 step approach for automated data transfers.
1. Memory footprint
   *How many data?*

2. Data direction
   *Transfer direction for kernel call*

**METHODS ARE AUTOMATED**
Methods & Tools

1. LLVM (Low-Level Virtual Machine)
2. Graphs for representing dependencies

- LLVM IR is independent from front-end language (C/C++, Fortran, ...)
- no additional object metadata required
- Information traversal eased
- Extensible
Workflow

1. Annotate Kernel function
2. Generate Data Dependency Graph
3. Extract data direction
4. Generate Type Direction Graph
5. Inject size functions into code
void Foo(std::vector<std::vector<int>> &X, int N) {
    for (int i = 0; i < N; i++) {
        X = TF(X);
    }
}
Intent from Data Dependency Graph

void Foo(std::vector<std::vector<int>>& X, int N) {
    for (int i = 0; i < N; i++) {
        X = TF(X);
    }
}

- Directed
- Object Loads & Stores  
  \( N \) : LD only \( \Rightarrow \) IN
- Explore function calls  
  \( X \) : LD + ST \( \Rightarrow \) IN + OUT
void Foo(std::vector<std::vector<int>> &X, int N) {
    for (int i = 0; i < N; i++) {
        X = TF(X);
    }
}

- Directed
- Object type breakdown
- Recursion with cycles
SIZE Functions

- Directly written in LLVM IR
- Can be statically injected into IR code
- Consider structural type information
- Returned size information at run-time (dynamic info)
\texttt{sizeof(N) of type int is}
\texttt{return sizeof(int)}

\texttt{sizeof(X) of type std::vector<\text{std::vector<int}}> is}
\texttt{s = 0}
\texttt{foreach x in X:}
\quad \texttt{s += count(x) * sizeof(int)}
\texttt{return s}
SIZE Function and for Complex Types

class A contains { int X, Y, Z; string name; }

sizeof(Y) of type std::vector<class A> is
s = 0
foreach y in Y:
    s += sizeof(y)
return s

sizeof(a) of type class A is
return 3 * sizeof(int) + sizeof(a.name)

sizeof(s) of type string is
return count(s) * sizeof(char)
Supported Data Types

- scalars
- arrays
  - Pointers; traceback to malloc / new function calls
  - Loop analysis for multi-dimensional arrays
- classes and structures
  - with templates
- STL containers
  - requires iterator methods, e.g. for map
  - iterators with static & dynamic allocation
- multiple inheritance
Evaluation

Limitations

- Overlapping pointers
- Function pointers
- Re-allocation
Type Dependency Graphs

MINIFE

HPCCG
Type Dependency Graphs

OCTREE
(UNROLLED)
Introduced Overhead

**NEW**

foo (int *a, Y *y)
multiple inheritance in class Y

**STRVEC**

foo (std::vector<std::string> x)
varying element size

**MAP**

foo (std::map<int, std::string> x)
STL container, iterator
Status & Conclusion

- Memory footprint & data direction (intent) can be derived from dependency graphs.
- Approach supports non-linear data objects.
- Implementation of two LLVM passes:
  - IR code used as source of graph construction.
  - Subsequent analysis.
- Injected IR code (SIZE functions) can deal with dynamically varying data objects.
Outlook

- Improving the type compatibility
- Step 2: Analyze the data layout on host & device (serialization schema)
- Step 3: Copy data according the serialization schema
THANK YOU