Task Coarsening Through Polyhedral Compilation for a Macro-Dataflow Programming Model

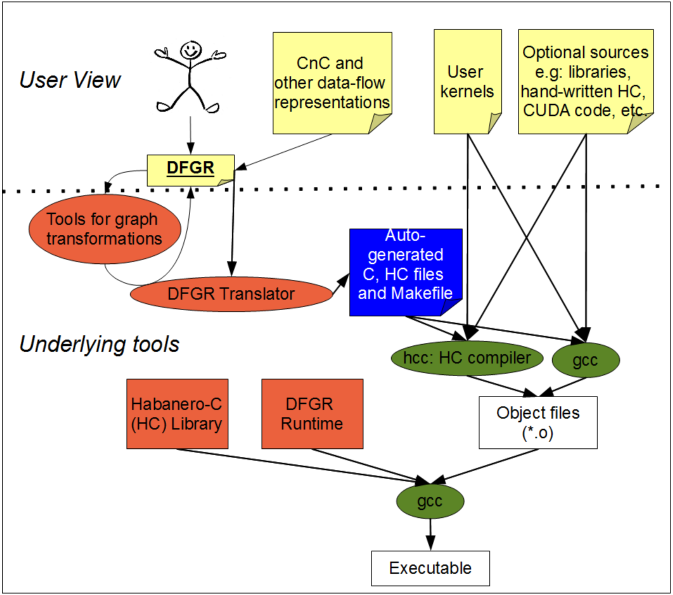
Alina Sbirlea, Louis-Noël Pouchet, Vivek Sarkar

Rice University
Ohio State University

January 19, 2014

IMPACT'15
Amsterdam

DFGR and HC


User View
CnC and other data-flow representations
User kernels
Optional sources e.g: libraries, hand-written HC, CUDA code, etc.
DFGR
Tools for graph transformations
DFGR Translator
Auto-generated C, HC files and Makefile
Underlying tools
hcc: HC compiler
gcc
Habanero-C (HC) Library
DFGR Runtime
Object files (*.o)
gcc
Executable

Poster

IMPACT 2015

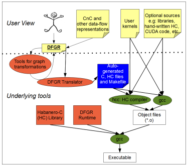
Task Coarsening Through Polyhedral Compilation for a Macro-Dataflow Programming Model

Alina Sbirlea[1], Louis-Noel Pouchet[2], Vivek Sarkar[1]   [1]Rice University, [2]Ohio State University

DFGR: Data-Flow Graph Representation

DFGR

- Has two components:
  - Textual component:
    - high-level view for domain experts
  - IR component:
    - automatic generation from higher-level programming systems
- Uses current software and compilers:
  - Habanero-C provides a parallel task language with extensions for OpenCL code generation
  - OCR for a distributed execution
  - TLDM generation for FPGAs
- Proposes the use optimizations at the IR level.
- See DFM'14 publication by Sbirlea, Pouchet and Sarkar



Textual DFGR Constructs

- Item collection declarations
  - [int* item1]; [float* item2];
- Step collection declarations
  - (step1 : a, b) @CPU=val1, GPU=val2, FPGA=val3;
- Step prescriptions
  - (step1 : i, j) :: (step2 : i+1, j*j);
- Step I/O relations
  - (step2: bar(i, j), j) -> (step1 : i, j);
  - [item1: i-1, j-1] -> (step1 : i, j+1);
  - (step1 : i, j) -> [item1 : i, j], [item2 : i+1, j];
- Ranges and Regions
  - [item1 : {i-1,i+1},{j-1,j+1} -> (step1 : i, j);
  - <region1 : i, j> { 1 <= i, i <= M, 1 <= j, j <= N };
  - env::(step1 : region1);
  - <region2(p, q) : i, j> { p-1 <= i, i <= p+1, q-1 <= j, j <= q+1 };
  - (step1 : i, j) -> [item2 : region2(i,j)];
- Environment
  - env :: (step1 : region1);
  - env -> [item1 : region1]; [item2 : region1] -> env;

DFGR regions as iteration spaces: a hierarchy of concepts

- Ranges: model rectangles, suited for simple regular computations
- Simple polyhedron: affine inequalities; powerful static analysis & transformations
- Union of Z-polyhedra: generalization of polyhedra, analyzable using modern polyhedral compilation frameworks
- Union of arbitrary sets: most general; includes uninterpreted functions (foo(i))

Key Features

- Steps are functional
- Item collections implement Dynamic Single Assignment form
- Data type in collections can be arbitrary (w/ serializers)
- Dependence between steps with step-to-step dependence or via data dependence
- Use tags as unique identifiers for step instances and items in collections
- Tag values may be known only at runtime or at compile-time
- Natively represent task-level, pipeline and stream parallelism

Transforming DFGR graphs for task+data coarsening

DFGR to Polyhedra

- Support the subset of DFGR programs without non-affine expressions, uninterpreted functions, nor data-dependent get/puts (e.g., [A : [B : i]])
- Conversion to polyhedral representation (SCopLib)
  - Create iteration domains by propagating the tag functions in step prescriptions
  - Create access functions directly from item tag functions
  - No schedule created
- Extract dependence polyhedra: DSA form ensures only flow dependences: no need for any schedule to determine which instance is the producer or consumer for RAW

Polyhedra to Polyhedra

- Transformation objective for DFGR on CPU: increase task granularity to have less tasks computing on more data and reduce communication.
- Use iteration space tiling on the polyhedral representation with the PLuTo algorithm [Bondhugula et al,2008]
- Input is polyhedral representation + dependence polyhedra, run PLuTo as-is and obtain a schedule for the transformed program as well as tiled iteration domains

Polyhedra to DFGR

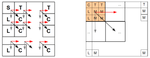
- Generate C code implementing the tiled schedule using CLooG [Bastoul,2004]
- New DFGR tasks are created for each tile body generated
- Dependence between tiles are modeled by describing the data flowing between tiles (read/written)
- Data flow of the transformed program extracted by polyhedral analysis, after updating also the data layout with tiling of data in item collections
- DSA on data tiles may not be preserved but the transformed code is still DSA: use "fake" item collections to make the DFGR graph DSA if multiple tags write to the same tile

Smith-Waterman example

C code

```
A[0][0] = corner();
for(j=1; j<NW; j++)
  A[0][j] = top(j);
for(i=1; i<NH; i++) \{
  A[i][0] = left(i);
  for(j=1; j<NW; j++)
     A[i][j] = center(i, j, A[i-1][j-1],
                  A[i-1][j],A[i][j-1];
\}
```

Dependences



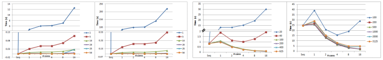
Input DFGR

```
<int A>;
(corner:i,j) -> [A:i,j];
[A:i,j-1] -> (top:i,j) -> [A:i,j];
[A:i-1,j] -> (left:i,j) -> [A:i,j];
[A:i-1,j-1], [A:i-1,j], [A:i,j-1] ->
       -> (center:i,j) -> [A:i,j];
env::(corner:0,0);
env::(top:0,{1 .. NW});
env::(left:{1 .. NH},0);
env::(center:{1 .. NH},{1 .. NW});
[A:NH,NW] -> env;
```

Transformed DFGR

```
<int** A >;
(newStmt1 : c1, c2) -> [ A : c1, c2];
[ A : c1, c2-1 ] -> (newStmt3 : c1, c2) -> [ A : c1, c2 ];
[ A : c1-1, c2 ] -> (newStmt2 : c1, c2) -> [ A : c1, c2 ];
[ A : c1-1, c2 ], [ A : c1, c2-1 ], [ A : c1-1, c2-1 ] ->
          (newStmt4 : c1, c2) -> [ A : c1, c2 ];
< regnewStmt2 : c1> { max(1,0)<= c1 <= floord(NH, 32) };
< regnewStmt3 : c2> { 1<=c2<=floord(NW, 32) };
< regnewStmt4 : c1, c2> { max(1,0)<= c1 <= floord(NH, 32);
                          1<= c2 <= floord(NW, 32) };
env :: (newStmt1 : 0, 0);
env :: (newStmt2 : regnewStmt2, 0);
env :: (newStmt3 : 0, regnewStmt3);
env :: (newStmt4 : regnewStmt4);
```

Performance results on 16-core Intel E7330 @ 2.4 GHz



(a) Input sequence sizes: 400×400. (b) Input sequence sizes: 800×800. (a) Input sequences: 10k×10k. (b) Input sequences: 50k×50k.