CARP: Correct and Efficient Accelerator Programming

Presenter: Javed Absar

ARM CARP Team: Alexey Kravets, Anton Lokhmotov, David Tweed, Georgios Pinitas, Sven Van Haastregt, Ulysse Beaugnon

Media Processing Division (MPD)

Contents

- Introduction
  - CARP
  - Motivation
  - Compilation Framework

- **PENCIL Language**
  - Features
  - Language Examples

- **VOBLA Language**
  - Control flow, expressions
  - Function templates
  - Storage, layout, views

- **BLAS Implementation**

- Results

- Conclusion
Introduction - CARP

- Objective
  - To design compilation and verification techniques and tools for Correct and Efficient AcceleratoR Programming

- European Commission funded project
  - Funded through FP7 Scheme
  - Started on 1 Dec, 2011
  - European Commission contribution: 2.8 million Euros
  - Partners: ARM, ENS, Imperial College London, Realeyes, RWTH Aachen, Monoidics, Twente University, Rightware
Introduction - Motivation

- **Native Accelerator programming is low-level**
  - Hard to program
  - Lack of performance portability

- **Many researchers focusing on ease of programming**
  - Domain-specific languages (DSLs) – Halide, OptiML
  - High-level programming models

- **Compiling directly to low-level code is inadvisable**
  - Implementers duplicate work and compromise on quality

- **Need an intermediate language**
  - For $n$ input languages and $m$ architectures, build $n+m$ rather than $n*m$ compilers.
  - The right approach to achieve performance portability

- **Wish to leverage Polyhedral Magic for GPUs**
  - Polyhedral techniques for arrays and loop transformation sync well with GPU architecture and GPU targeting applications
Compilation Framework

- Domain Specific Language
- Domain Specific Language
- Domain Specific Language

DSL Compilers

Intermediate Language

IL-2-Native Compiler

Native Accelerator API (OpenCL, CUDA)

- NVIDIA GeForce
- AMD Fusion
- ARM Mali
- Qualcomm Adreno
1. Design VOBLA, a domain-specific language for linear algebra

2. Design PENCIL, a platform-neutral computer intermediate language, to be used either by DSL compilers as their target language or by expert programmers

3. Contribute to designing and implementing advanced compilation techniques (based on the polyhedral model) for translating platform-independent PENCIL code into efficient platform-specific OpenCL code

4. Develop a performance-portable implementation of Basic Linear Algebra Subroutines (BLAS) based on the DSL-to-PENCIL-to-OpenCL flow

5. Devise a performance and energy model of the Mali-T600. Validate that it allows the compiler to make useful predictions for optimizing OpenCL code.
CARP Compilation Framework

Hand-Written PENCIL

Domain Specific Language

Vehicle for Optimized Linear Algebra (VOBLA)

DSL Compilers

Platform-Neutral Compute Intermediate Language (PENCIL)

Parallel Polyhedral Code Generation (PPCG)

Native Accelerator API (OpenCL)

NVIDIA GeForce

AMD Fusion

ARM Mali

Qualcomm Adreno

Mali OpenCL Compiler

MODEL
PENCIL: Platform-neutral Compute Intermediate Language
PENCIL Features

- **Facilitate Parallelism: Declarative rather than Imperative**
  - Convey information on parallelism in the code. Not specify how exactly to parallelise and schedule

- **PENCIL code should be source compatible with C99 compiler**
  - Important for checking sequential correctness

- **Key Directives**
  - `__pencil_assume`, `__pencil_assert`
  - pragmas – pencil independent (reduction), ivdep
  - Summary function – DEF, USE

- **Other PENCIL Features**
  - Restricted use of pointers
  - Use of `const restrict static`
  - Canonical forms for loops
PENCIL Language

```c
void foo1 (int n, int m, int S, int D[const restrict static S]){
  __pencil_assume ( m > n);
  for (int k = 0; k < n; k++) { D[i] = D[i+m]; }
}

void foo1 (int m, int S, int D[const restrict static S],
           int E[const restrict static S]){  
  #pragma pencil independent
  for (int k = 0; k < m; k++)  D[E[i]] = i*i;
}

int foo1 ( int m, int S, int D[const restrict static S],
           int E[const restrict static S]){  
  double ret = 0.0;
  #pragma pencil independent reduction (+:ret)
  for (int k = 0; k < m; k++)  ret += D[i]*E[i];
    return ret;
}
```
PENCIL Language

- SUMMARY FUNCTION
  - To capture complex or unknown memory access information

```c
int foo( int N, int A[const restrict static N]) ACCESS(foo_summary);

int foo_summary( int N, int A[const restrict static N]){
    USE(A);
}

void bar( int N, int A[const restrict static N]){
    foo(N,A);
    for (int k = 0; k < m; k++) A[i]++;
}
```
VOBLA - Introduction

- VOBLA: Vehicle for Optimizing Basic Linear Algebra
  - Domain Specific Language (DSL) to handle dense and sparse matrices
  - Enables concise specification of linear algebra codes
    - LAPACK and BLAS
  - Generate rich PENCIL code

- VOBLA Features
  - Compact code when describing complex matrix operations
  - Array shape support
    - Single function can handle various access pattern (transpose, conjugate)
  - VOBLA functions are generic which can be exported to different types
  - Easily iterate over different data layout (e.g. sparse matrices)
VOBLA Language

- Control flow operators - for, forall, while, if
  - each iteration of forall is independent
  - for/forall specify multi-dimensional iteration space
  - Compact representation for loops over assignment

    \[
    x[i] = i \text{ forall } i \text{ in } 0:n-1;
    \]

- Expressions
  - Basic operators are available for vectors and vector-scalar operands
  - \text{len}: returns size of different dimensions of an array
  - \text{sum}: operator to sum over sequence of scalars

    \[
    \text{let } \text{norm2} = \text{sum}(x*x \text{ forall } _, x \text{ in } X);
    \]

- Re, Im and Conjugate built-in function
VOBLA Language

- **Template Functions**
  - To manage different versions – floating (floating point precision, array storage) of same function
  - To improve ease of programming and compactness by exposing only the algorithm part of code
  - Help compiler by preventing obfuscation through implementation details

```plaintext
function scal(a: Value, out X: Value[]) { ... }

export scal<Complex Double>(X is Column) as zscal;

export scal<Float>(X is Reversed Column) as sscal;
```
VOBLA Language

- **Array Access Pattern**
  - Hide complexity of storage, provide generality and simplify code generation

- **Iterate access pattern**
  - Enumerate the elements in an undefined pattern
    
    \[
    A_{ij} = i \times j \quad \text{forall} \quad i, j, A_{ij} \quad \text{in} \quad A;
    \]

- **Sparse iterate**
  - Skips elements that are zero
    
    \[
    \text{let } \text{norm2} = 0; \\
    \text{norm2} += X_i \times X_i \quad \text{for} \quad _, X_i \quad \text{in} \quad X.\text{sparse};
    \]

- **Indexed**
  - Cannot be implemented for every storage format
    
    \[
    \text{let } \text{dot} = 0; \\
    \text{dot} += x_i \times y[i] \quad \text{forall} \quad i, x_i \quad \text{in} \quad X.\text{sparse};
    \]
VOBLA Language

- **Array Operators**
  - Allow compact representation of array operations

    ```
    //X += 2*Y
    X[i] += 2*Yi  forall i, Yi in Y.sparse;
    ```

- **Array Views**
  - Help decouple algorithm from the way array is stored
  - Avoids copying of data

    ```
    //VOBLA
    ..X[2:4]..  // Take elements of X between 2 and 4
    ..A[2..]..  // Take the third column of A
    let a = Transpose(A)[i][j];  //float a = A[j][i];
    ```
VOBLA Language

- User Defined Access Patterns
  - In addition to iterate, sparse iterate and indexed

- Four objects to define access pattern
  - Interface
    - Specifies which access pattern is implemented by the array
    - During compilation each interface will have a layout implementing its interface specifying how to access the data
    - To access data, a method defined in by Interface object is called
  - Storage
    - Storage object contains the storage layout
  - Layout
    - Specifies how to access data for a given storage
  - View
    - Create new layouts from existing ones using inheritance
**VOBLA Language**

- **Storages**
  - Storage object represents the part of an array that is independent of the way it is accessed

```cpp
//storage object- coordinate list(COO) sparse format
storage CooStorage {
    nRows: Index;
    nCols: Index;
    nNonZeros: Index;
    rowIdx: Index[nNonZeros];
    colIdx: Index[nNonZeros];
    data: Value[nNonZeros];
}
```

```
[5 0 0]
[9 0 7]
[0 0 3]
```

<table>
<thead>
<tr>
<th>ROW</th>
<th>0</th>
<th>1</th>
<th>1</th>
<th>2</th>
</tr>
</thead>
<tbody>
<tr>
<td>COL</td>
<td>0</td>
<td>0</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>VALUE</td>
<td>5</td>
<td>9</td>
<td>7</td>
<td>3</td>
</tr>
</tbody>
</table>
VOBLA Language

- **Layout** definition contains
  - Name of storage object
  - List of interfaces implemented by the layout
  - An implementation for each of the methods defined in the interfaces

```c++
layout Coo: CooStorage implements SparseIterable<Value>[][] {
  parameter:
  interface:
    getLen1(): Index { return nRows; }
    getLen2(): Index { return nCols; }

    sparseIterate(): range<Index, Index, &Value> {
      yield rowIdx[k], colIdx[k], data[k]
      forall k in 0:nNonZeros;
    }
}
```
VOBLA Language

- **Views**
  - Enables layouts from existing layouts
  - Acts as wrapper around a layout

- **TransposeMat on base interface Accessible**

  ```
  view TransposedMat: Accessible<Value>[][]
  implements Accessible<Value>[][] {
    access (i: Index, j: Index): &Value {
      return base.access(j, i);
    }
  }
  ```

- **To access sparse transpose matrix in COO format**

  ```
  export gemm<Float>(A is TransposedMat(COO))
  ```
BLAS Implementation

- Basic Linear Algebra Subprograms library
  - Level 1: vector-vector operations
  - Level 2: matrix-vector operations
  - Level 3: matrix-matrix operations

- BLAS in VOBLA
  - Data types
    - Most BLAS functions are defined on 4 data types – single and double precision real numbers, single and double precision complex numbers
  - Matrix view
    - For each matrix BLAS takes in an additional flag indicating if matrix is to be interpreted as normal, transposed, or conjugate transpose
  - Storage layout
    - Level 2 and 3 BLAS functions support dense matrix storage layouts and pre-defined compressed or triangular layouts
BLAS Implementation

- **GEMM**: $C \leftarrow \alpha A B + \beta C$

```java
function gemm(alpha: Value,
in A: SparseIterable<Value>[m][k],
in B: Value[k][n],
beta: Value,
out C: Value[m][n])
{
    Cij *= beta forall _, _, Cij in C.sparse;
    C[i][j] += alpha*Ail*B[l][j]
        for i, l, Ail in A.sparse, j in 0:n-1;
}
```

- 50 export statements to obtain all BLAS variants
Results
Conclusion

- VOBLA – A DSL for efficient linear algebra programming
- PENCIL – special constructs to carry meta-data for polyhedral code generator
- VOBLA based BLAS on average 2.5x better than basic OpenCL implementation
- A step closer to solving programmer productivity and performance portability
References


