Vobla: A Vehicle for Optimized Basic Linear Algebra

Ulysse Beaugnon\textsuperscript{1,2} Alexey Kravets\textsuperscript{1} Sven van Haastregt\textsuperscript{1}
Riyadh Baghdadi\textsuperscript{2} David Tweed\textsuperscript{1} Javed Absar\textsuperscript{1}
Anton Lokhmotov\textsuperscript{1}

\textsuperscript{1}ARM Ltd., Cambridge, United Kingdom
\textsuperscript{2}INRIA and École Normale Supérieure, Paris, France

LCTES, 12–13 June 2014
OpenCL on embedded/mobile devices

Performance improvements

- Power consumption
- Execution time

Example of applications

- Image and video processing
- Signal processing
- Physics engines
Programmer Productivity

- Optimized accelerator code is tedious to write.
- Algorithm becomes tightly coupled with implementation.
- Poor opportunities for code reuse.

Example OpenCL code for cgemm:

```c
float4 accum_real = (float4)(0, 0, 0, 0);
float4 accum_img = (float4)(0, 0, 0, 0);
for (int c4 = 0; c4 < k; c4 += 2) {
    float4 aa = vload4(0, (global float *)&A[i * n + c4]);
    float4 bb;
    bb.s01 = vload2(0, (global float *)&B[c4*k + j]);
    bb.s23 = vload2(0, (global float *)&B[(c4 + 1)*k + j]);
    accum_real += aa * bb;
    accum_img += aa * bb.s1032;
}
```
Programmer Productivity

- Optimized accelerator code is tedious to write.
- Algorithm becomes tightly coupled with implementation.
- Poor opportunities for code reuse.

Example OpenCL code for cgemm:

```c
float4 accum_real = (float4)(0,0,0,0);
float4 accum_img = (float4)(0,0,0,0);
for (int c4 = 0; c4 < k; c4 += 2) {
    float4 aa = vload4(0, (global float*)&A[i * n + c4]);
    float4 bb;
    bb.s01 = vload2(0, (global float*)&B[c4*k + j]);
    bb.s23 = vload2(0, (global float*)&B[(c4 + 1)*k + j]);
    accum_real += aa * bb;
    accum_img += aa * bb.s1032;
}
```
Performance Portability

- OpenCL is functionally portable, but not *performance portable*.
- Code optimized for device A performs poorly on device B.
- In fact, some code may not run at all on a different device (e.g. out of resources).
OpenCL is functionally portable, but not *performance portable*.

Code optimized for device A performs poorly on device B.

In fact, some code may not run at all on a different device (e.g. out of resources).
Solution

- **VOBLA**
  - `voblac`
  - DSL compiler

- **other DSL**
  - DSL compiler

- **PENCIL**
  - `voblac`

- **PPCG**

- **OpenCL**

**Domain specific**
- **Easy to write**

**Target and domain independent**
- **Harder to write**

**Target specific optimized code**
- **Very hard to write**
DSL Level: Extracting the Algorithm

Algorithmic concerns

\[ Y \leftarrow \alpha AX + \beta Y \]
DSL Level: Extracting the Algorithm

Algorithmic concerns

\[ Y \leftarrow \alpha AX + \beta Y \]

Non-algorithmic concerns

- Dense or sparse matrix
- Single or double precision
- Real or complex
- Storage format
- Loop tiling size
- Vectorization factor
- ...

Export statements

Generic algorithm

// In function foo
Yi *= beta forall _, Yi in Y.sparse;
Y[i] += alpha*Aij*X[j] for i, j, Aij in A.sparse;

Specialization

export foo<Double>(A is Array) as dfoo_dense;
export foo<Complex Float>(A is Csr) as cfoo_csr;
## Access patterns

<p>| | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>6</td>
<td>0</td>
<td>3</td>
<td>2</td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>4</td>
<td>7</td>
<td>2</td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>3</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>7</td>
<td></td>
</tr>
</tbody>
</table>
Access patterns

Iterate

- Enumerate all the elements

\[ A_{ij} = i \times j \text{ forall } i, j, A_{ij} \text{ in } A; \]

\[
\begin{array}{cccc}
6 & 0 & 3 & 2 \\
0 & 4 & 7 & 2 \\
0 & 0 & 3 & 1 \\
0 & 0 & 0 & 7 \\
\end{array}
\]
Access patterns

**Iterate**
- Enumerate all the elements

\[ A_{ij} = i \times j \text{ for all } i, j, A_{ij} \text{ in } A; \]

**Sparse Iterate**
- Enumerate the elements
- Skip zeros when possible

\[ \text{norm2 } += X_i \times X_i \text{ for } _, X_i \text{ in } X.\text{sparse}; \]
**Access patterns**

### Iterate
- Enumerate all the elements

\[ A_{ij} = i \times j \text{ forall } i, j, A_{ij} \text{ in } A; \]

### Sparse Iterate
- Enumerate the elements
- Skip zeros when possible

\[ \text{norm2} += X_i * X_i \text{ for } _, X_i \text{ in } X\text{.sparse}; \]

### Random Access
- Access by element index

\[ \text{dot} += x_i * y[i] \text{ forall } i, x_i \text{ in } X\text{.sparse}; \]
Array operators

- =, +, -, *, /, sum operators available
- Use the best access pattern available

Example

// Original code
X *= 2;

// Compiled code
Xi *= 2 forall _, Xi in X.sparse;
Data views

Matrix transposition for free

// Original code
x = Transpose(B)[i][j]

// Compiled code
x = B[j][i]

Features

- Also handle submatrices, row extraction, conjugate
- Views can be combined
- User can define new views
Code generation

- **VOBLA**
  - voblac
  - DSL compiler
  - PENCIL
  - PPCG
  - OpenCL

- **other DSL**
  - DSL compiler
  - PPCG
  - OpenCL

**Domain specific**
- Easy to write

**Target and domain independent**
- Harder to write

**Target specific optimized code**
- Very hard to write
Expected features of the Pencil compiler

- Generates both GPU and CPU code
- Handle data transfers between the GPU and the CPU
- Generates efficient parallel code
Pencil to OpenCL

**Expected features of the Pencil compiler**
- Generates both GPU and CPU code
- Handle data transfers between the GPU and the CPU
- Generates efficient parallel code

**Polyhedral compilation**
- Automatic parallelization
- Well suited for massive data parallelism
- Only works for regular access patterns
Generates polyhedral-friendly code

- Array iterators and operators
- Duplicate code instead of reshaping arrays
Pencil code generation from Vobla

Generates polyhedral-friendly code
- Array iterators and operators
- Duplicate code instead of reshaping arrays

Expose information on arrays
- Keep track of array size to copy them to the GPU memory
- Infer relations between arrays size to enable optimizations
- Track aliasing
Pencil code generation from Vobla

Generates polyhedral-friendly code
- Array iterators and operators
- Duplicate code instead of reshaping arrays

Expose information on arrays
- Keep track of array size to copy them to the GPU memory
- Infer relations between arrays size to enable optimizations
- Track aliasing

Explicit parallelization by the programmer
- Use `for` to let the compiler find parallelism
- Use `forall` to indicate parallelism of irregular code
Example: Gemv

Generic Matrix-Vector multiplication:

\[ Y \leftarrow \alpha \times A \times X + \beta \times Y \]
import sparse csr;

function gemv(
    alpha: Value,
    in A: SparseIterable<Value>[m][n],
    in X: Value[n],
    beta: Value,
    out Y: Value[m]) {
    // Y = beta * Y
    Y *= beta;
    // Y += A * X
    Y[i] += alpha*Aij*X[j] for i, j, Aij in A.sparse;
}

export gemv<Complex Float>(A is Csr) as gemv_csr;
Example: Gemv

```c
void cgemv_csr(
    const struct ComplexFloat alpha,
    const struct Csr_view A_view,
    const int A_colIdx[restrict const static A_view.storage.nNonZero],
    const int A_rowPtr[restrict const static A_view.storage.nRows+1],
    const struct ComplexFloat A_data[restrict const static A_view.storage.nNonZero],
    const struct VectorView X_view,
    const struct ComplexFloat X[restrict const static X_view.storage.base_size0],
    const struct ComplexFloat beta,
    const struct VectorView C_view,
    struct ComplexFloat Y[restrict const static C_view.storage.base_size0]) {
    __pencil_assume((X_view.view_size0==A_view.storage.nCols));
    __pencil_assume((C_view.view_size0==A_view.storage.nRows));
    #pragma pencil independent
    for (int i = 0; i <= C_view.view_size0-1; i += 1) {
        struct ComplexFloat tmp;
        tmp.Re = Y[C_view.offset0+i].Re * beta.Re - Y[C_view.offset0+i].Im*beta.Im;
        tmp.Im = Y[C_view.offset0+i].Im * beta.Re + Y[C_view.offset0+i].Re*beta.Im;
        Y[C_view.offset0+i] = tmp;
    }
    for (int i = 0; i <= A_view.storage.nRows-1; i += 1) {
        for (int j = A_rowPtr[i]; j <= A_rowPtr[i+1]-1; j += 1) {
            struct ComplexFloat AXpY;
            struct ComplexFloat alphaA;
            struct ComplexFloat alphaAX;
            alphaA.Re = alpha.Re*A_data[j].Re - alpha.Im*A_data[j].Im;
            alphaA.Im = alpha.Re*A_data[j].Re + alpha.Im*A_data[j].Im;
            alphaAX.Re = alphaA.Re*X[X_view.offset0+A_colIdx[j]].Re - alphaA.Im*X[X_view.offset0+A_colIdx[j]].Im;
            alphaAX.Im = alphaA.Re*X[X_view.offset0+A_colIdx[j]].Im + alphaA.Im*X[X_view.offset0+A_colIdx[j]].Re;
            AXpY.Re = Y[C_view.offset0+i].Re + alphaAX.Re;
            AXpY.Im = Y[C_view.offset0+i].Im + alphaAX.Im;
            Y[C_view.offset0+i] = AXpY;
        }
    }
}
```
Results on Mali-T604 GPU

Execution time (normalized)

- Naive
- Hand-optimized
- Vobla
Contributions

Vobla DSL

- Generic code that can be specialized into many versions
- Concise code that only exposes algorithmic concerns
- Support for user-defined storage formats

BLAS implementation

- $8 \times$ improvement over straightforward OpenCL code
- Compliant with the original implementation

Pencil compilation flow validation

- Working toolchain from Vobla to OpenCL
Thank you!

Get the source code:

http://github.com/carpiproject

Learn more about the CARP project:

http://carpiproject.eu