Vobla: A Vehicle for Optimized Basic Linear Algebra

Ulysse Beaugnon[1,2] Alexey Kravets[1] Sven van Haastregt[1]
Riyadh Baghdadi[2] David Tweed[1] Javed Absar[1]
Anton Lokhmotov[1]

[1]ARM Ltd., Cambridge, United Kingdom
[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.

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:

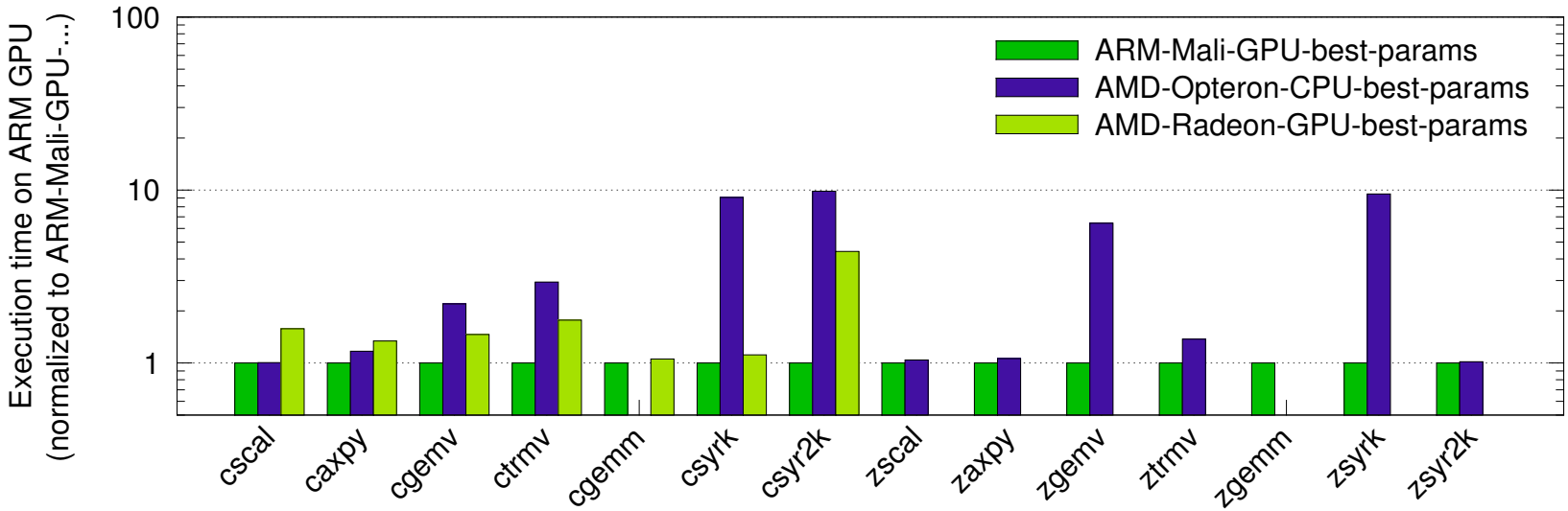
```
...
float4 accum_real = (float4)(0,0,0,0);
float4 accum_img = (float4)(0,0,0,0);
for (int c4 = 0; c4 < k; c4 += 2) {
  float4 aa = vload4(0, (global float*)&A[i * n + c4]);
  float4 bb;
  bb.s01 = vload2(0, (global float*)&B[c4*k + j]);
  bb.s23 = vload2(0, (global float*)&B[(c4 + 1)*k + j]);
  accum_real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

Performance Portability

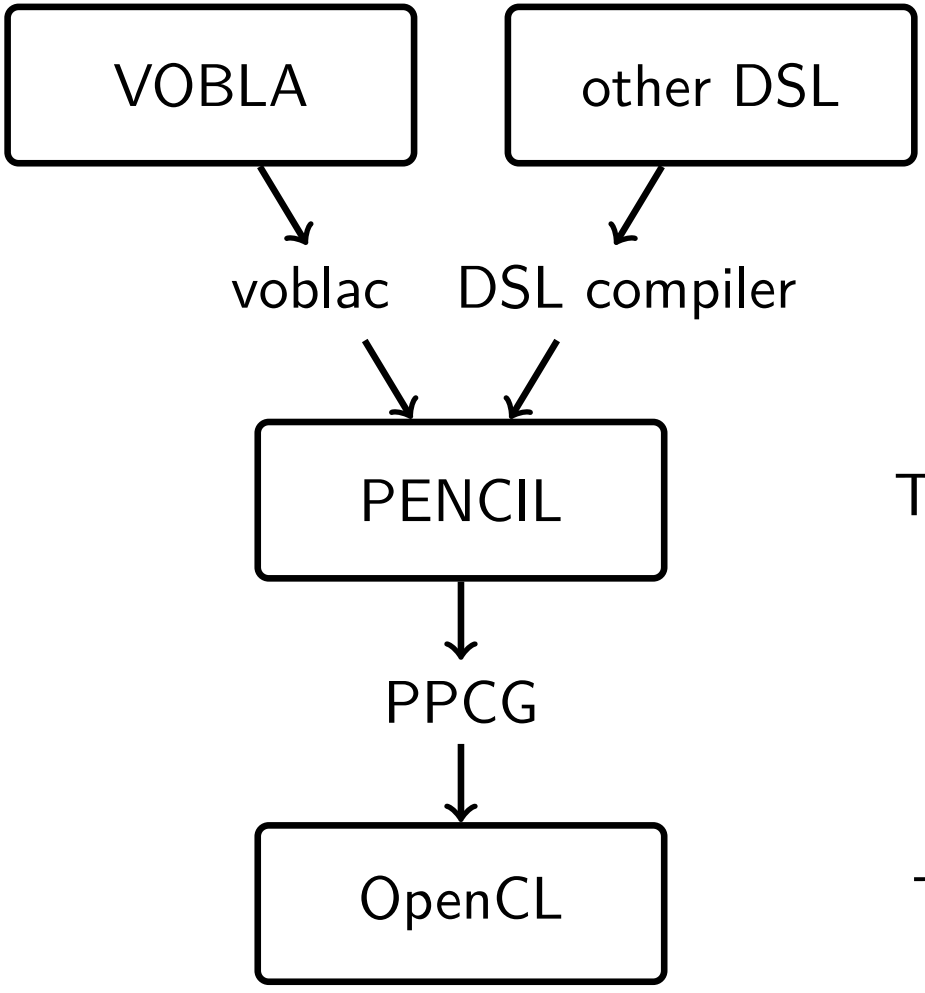
- OpenCL is functionally portable, but not *performance portable*.
- Code optimized for device A performs poorly on device B.
- In fact, some code may not run at all on a different device (e.g. out of resources).

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).



Solution



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

6	0	3	2
0	4	7	2
0	0	3	1
0	0	0	7

Access patterns

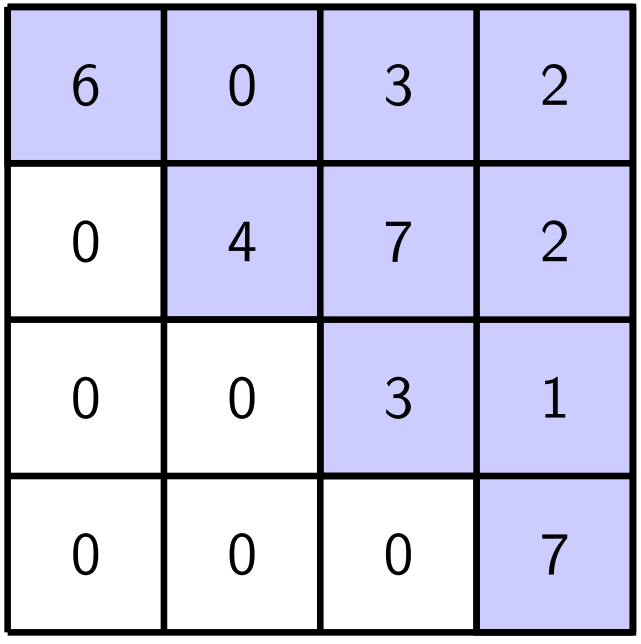
6	0	3	2
0	4	7	2
0	0	3	1
0	0	0	7

Iterate

- Enumerate all the elements

```
Aij = i * j forall i, j, Aij in A;
```

Access patterns



Iterate

- Enumerate all the elements

```
Aij = i * j forall i, j, Aij in A;
```

Sparse Iterate

- Enumerate the elements
- Skip zeros when possible

```
norm2 += Xi * Xi for _, Xi in X.sparse;
```

Access patterns

6	0	3	2
0	4	7	2
0	0	3	1
0	0	0	7

Iterate

- Enumerate all the elements

```
Aij = i * j forall i, j, Aij in A;
```

Sparse Iterate

- Enumerate the elements
- Skip zeros when possible

```
norm2 += Xi * Xi for _, Xi in X.sparse;
```

Random Access

- Access by element index

```
dot += xi * y[i] forall i, xi in X.sparse;
```

Array operators

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

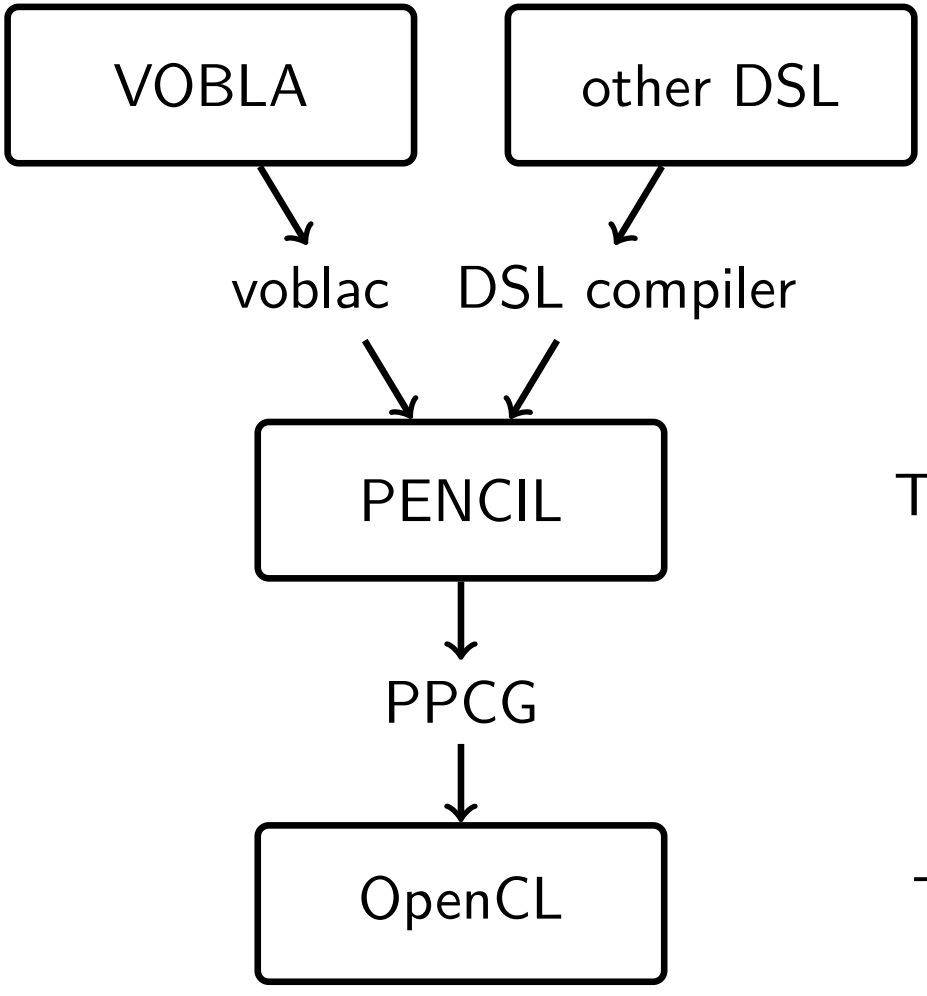
```
// Original code
x = Transpose(B)[i][j]
```

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

Features

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

Code generation



Domain specific

Easy to write

Target and domain independent

Harder to write

Target specific optimized code

Very hard to write

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

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 suitted for massive data parallelism
- Only works for regular access patterns

Pencil code generation from Vobla

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:

$$\boldsymbol{Y} \leftarrow \alpha \times \boldsymbol{A} \times \boldsymbol{X} + \beta \times \boldsymbol{Y}$$

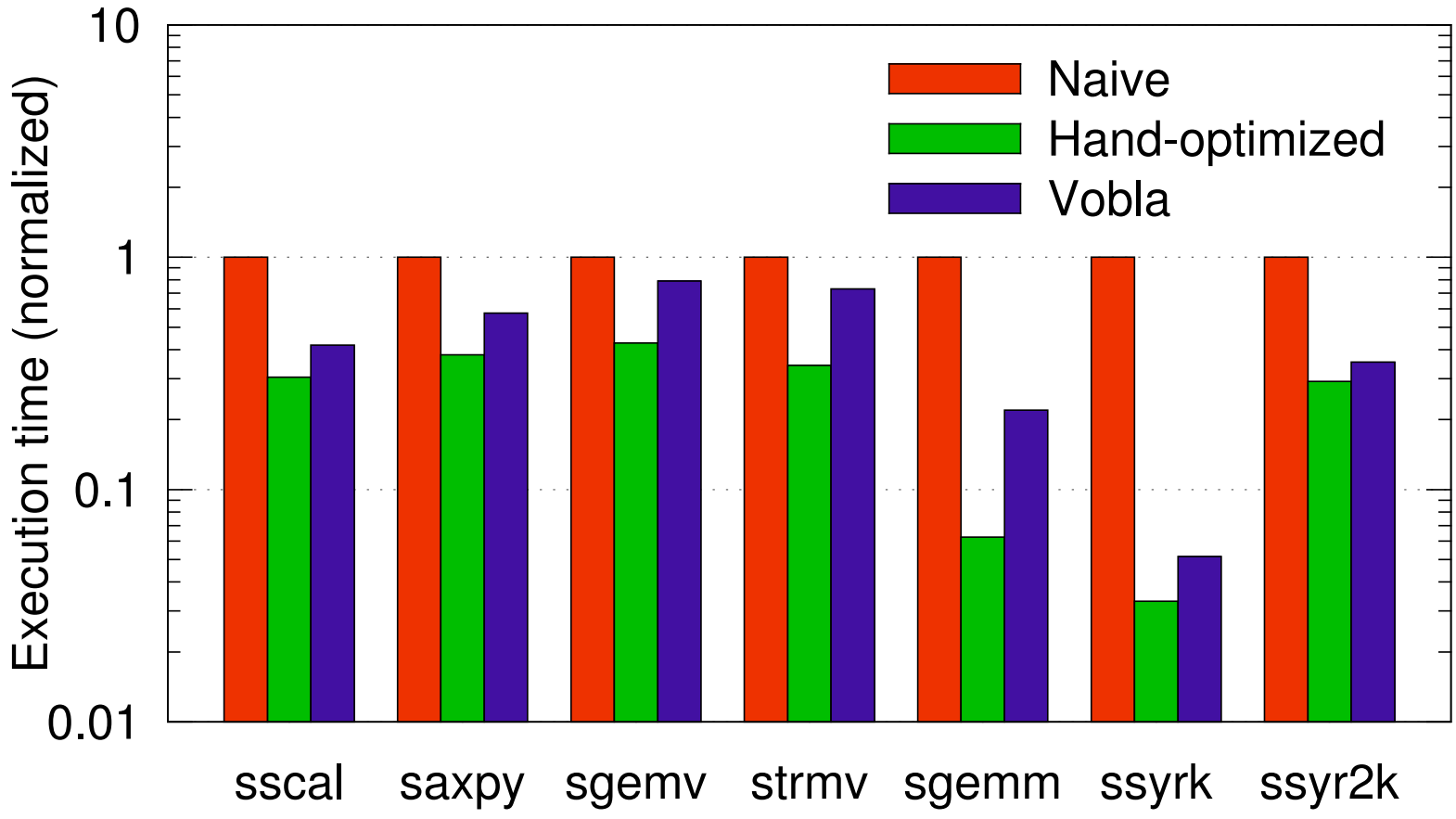
Example: Gemv

```
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

```
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.Im*A_data[j].Re + alpha.Re*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.Im*X[X_view.offset0+A_colIdx[j]].Re + alphaA.Re*X[X_view.offset0+A_colIdx[j]].Im;
      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



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× 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/carpproject

Learn more about the CARP project:

http://carpproject.eu