Domain Specific Languages to Tame Heterogeneous and Emerging Computing Systems

Jeronimo Castrillon
Chair for Compiler Construction (CCC)
TU Dresden, Germany

The Platform for Advanced Scientific Computing (PASC) Conference
Geneva (virtual), Switzerland
July 9, 2021
Evolution of computing

- single-core architectures
- Use multi-core architectures
- Dark Si: specialize
- Post CMOS?

- Massive parallel and heterogeneous systems
- Specialization: TPUs, AI engines, PCM, …
- Interconnected & distributed computing
- Emerging non-volatile memories

The *golden era* in computer architecture requires major changes in *programming methods* to *democratize* heterogeneous and emerging high-performance computing.
What's wrong with good old sequential languages?

What we want

\[ v_{ijk,e} = \sum_{i'=0}^{p} \sum_{j'=0}^{p} \sum_{k'=0}^{p} A_{kk'} A_{jj'} A_{ii'} u_{i'j'k'} \]

What we (naively) code

```c
void cfd_kernel(
    double A[restrict 7][7],
    double u[restrict 216][7][7][7],
    double v[restrict 216][7][7][7])
{
    /* element loop: */
    for(int e = 0; e < 216; e++) {
        for(int i0 = 0; i0 < 7; i0++) {
            for(int j0 = 0; j0 < 7; j0++) {
                for(int k0 = 0; k0 < 7; k0++) {
                    v[e][i0][j0][k0] = 0.0;
                    for(int i1 = 0; i1 < 7; i1++) {
                        for(int j1 = 0; j1 < 7; j1++) {
                            for(int k1 = 0; k1 < 7; k1++) {
                                v[e][i0][j0][k0] += A[i0][i1] * A[j0][j1] * A[k0][k1] * u[e][i1][j1][k1];
                            }
                        }
                    }
                }
            }
        }
    }
    /* end of element loop */
}
```

What compilers see

How many more times should we optimize this manually?
Polyhedral compilation: Hope for regular loops

- Recognize high-level patterns like matrix multiply-and-add operation (MMA)

  “Our method attained the performance of vendor optimized BLAS libraries”

Complex and sensitive pattern recognition to help close the performance gap

There is only su much we can do/reconstruct...

- Lots of progress: polyhedral compilers, trace-driven dynamic parallelization, patterns/idiom extraction

Bridge gap: Domain experts → C++/fortran

DSLs start here!

DSLs for performance: Halide, Spiral, TVM, TensorFlow, Firedrake...

```c
while (!queue.empty())
{
    // Dequeue a vertex from queue
    s = queue.front();
    queue.pop_front();

    // Apply function f to s, accumulate values
    result += f(s);

    // Get all adjacent vertices of s.
    if (adj[s].begin() != adj[s].end())
    {
        for (i = adj[s].begin(); i != adj[s].end(); ++i)
        {
            queue.push_back(*i);
        }
    }
}
```

Examples (1): Tensors expressions (CFD, ML)

- Expression-language for tensor operations and optimizations
  - Originally for spectral element methods in computational fluid dynamics

\[ v_e = (A \otimes A \otimes A) u_e \]

Interpolation kernel

```plaintext
source = ... 
var input A : matrix &
var input u : tensorIN &
var input output v : tensorOUT &
var input alpha : [] &
var input beta : [] &
v = alpha * (A # A # A # u .
[[5 8] [3 7] [1 6]]) + beta * v
```

<table>
<thead>
<tr>
<th>Fortran and C++ integration</th>
</tr>
</thead>
<tbody>
<tr>
<td>auto A = Matrix(m, n), B = Matrix(m, n), C = Matrix(m, n);</td>
</tr>
<tr>
<td>auto u = Tensor&lt;3&gt;(n, n, n);</td>
</tr>
<tr>
<td>auto v = (A<em>B</em>C)(u);</td>
</tr>
</tbody>
</table>

Semantic gap ➔ performance gap

What we (naively) code

$$v_e = (A \otimes A \otimes A) u_e$$

$$v_{ijk,e} = \sum_{i'=0}^{p} \sum_{j'=0}^{p} \sum_{k'=0}^{p} A_{kk'} A_{jj'} A_{ii'} u_{i'j'k'}$$

What performance experts code

```c
void cfd_kernel(
    double A[restrict 7][7],
    double u[restrict 216][7][7][7],
    double v[restrict 216][7][7][7][7])
{
    // element loop: */
    #pragma omp for
    for (int e = 0; e < 216; e++) {
        double t6[7][7][7];
        // 1st contraction: */
        #pragma simd
        for (int i0 = 0; i0 < 7; i0++) {
            for (int j0 = 0; j0 < 7; j0++) {
                for (int k0 = 0; k0 < 7; k0++) {
                    t6[i0][j0][k0] = A[i0][i1] * A[j0][j1] * A[k0][k1] + u[e][i1][j1][k1];
                }
            }
        }
    }
    // 2nd contraction: */
    #pragma simd
    for (int i1 = 0; i1 < 7; i1++) {
        for (int j1 = 0; j1 < 7; j1++) {
            for (int k1 = 0; k1 < 7; k1++) {
                t9 = t6[15][15][16];
                t7[14][15][16] = t9;
            }
        }
    }
    // 3rd contraction: */
    #pragma simd
    for (int i2 = 0; i2 < 7; i2++) {
        for (int j2 = 0; j2 < 7; j2++) {
            for (int k2 = 0; k2 < 7; k2++) {
                t10 = t9[15][15][16];
                t6[7][7][7] = t10;
            }
        }
    }
    // */ end of element loop */
}
```

Closing the performance gap

- Not really optimization magic
  - Leverage expert knowledge
  - Algebraic identities

\[ v_{ijk} = \sum_{l,m,n} (A_{kn} \cdot (A_{jm} \cdot (A_{il} \cdot u_{lmn}))) \]

\[ v_{ijk} = \sum_{l,m,n} (A_{kn} \cdot A_{jm}) \cdot (A_{il} \cdot u_{lmn}) \]

\[ v_{ijk} = \sum_{l,m,n} (A_{kn} \cdot ((A_{jm} \cdot A_{il}) \cdot u_{lmn})) \]


Easy to generate, hard to transform

Actual code variants

© Prof. J. Castri
Closing the performance gap

- Not really optimization magic
  - Leverage expert knowledge
  - Algebraic identities

\[
v_{ijk} = \sum_{l,m,n} (A_{kn} \cdot (A_{jm} \cdot (A_{il} \cdot u_{lmn})))
\]

\[
v_{ijk} = \sum_{l,m,n} (A_{kn} \cdot A_{jm}) \cdot (A_{il} \cdot u_{lmn})
\]

\[
v_{ijk} = \sum_{l,m,n} (A_{kn} \cdot ((A_{jm} \cdot A_{il}) \cdot u_{lmn}))
\]

TeML: Meta-programming for tensor optimizations

- Generalize for cross-domain tensor expressions
- Formal semantics and composition of transformations

\[ E \| \text{stripmine}(l, r, v) \| = \]
\[ \lambda \sigma. \text{let } \langle i_1, \ldots, \langle i_r, xs \rangle \ldots \rangle = \sigma(l) \]
\[ (b, e, 1) = i_r \]
\[ i' = (0, (e - b)/v - 1, 1) \]
\[ i'_{r+1} = (b + v \cdot i', b + v \cdot i' + (v - 1), 1) \]
\[ \text{in } \langle i_1, \ldots, \langle i'_{r+1}, xs \rangle \rangle \ldots \]

\[ E \| \text{interchange}(l, r_1, r_2) \| = \]
\[ \lambda \sigma. \text{let } \langle i_1, \ldots, \langle i_{r_1}, \ldots, \langle i_{r_2}, xs \rangle \ldots \rangle \ldots \rangle = \sigma(l) \]
\[ \text{in } \langle i_1, \ldots, \langle i_{r_2}, \ldots, \langle i_{r_1}, xs \rangle \ldots \rangle \ldots \rangle \]

Formally defined transformation primitives
Higher-level transformations via composition

Meta-programming for optimizations: Results

Performance of Pluto could be reproduced

Higher abstraction $\rightarrow$ more optimization potential

TeIL: Formal language – added value

- Core common to multiple tensor languages
- Index-free notation and strong type system
- **Provably** no out-of-bound accesses

\[
A = \text{placeholder}((m,h), \text{name}='A')
\]

\[
B = \text{placeholder}((n,h), \text{name}='B')
\]

\[
k = \text{reduce}(C_{ij} = \sum_{k=1}^{h} A_{ki} B_{kj})
\]

\[
C = \text{compute}((m,n), \text{lambda } i, j: \text{sum}(A[k, i] * B[k, j], \text{axis}=k))
\]

Examples (2): Particle-mesh simulations

- Particle-mesh simulations in computational biology
  - Discrete/continuous
  - Deterministic/stochastic

Syntax for interact, evolve, automatic insertion of interpolation, …

\[
\begin{align*}
t & = Du \ast \nabla^2 u - u \ast v^2 + F \ast (1 - u) \\
\frac{\partial v}{\partial t} & = Dv \ast \nabla^2 v + u \ast v^2 - v \ast (F + k)
\end{align*}
\]


Semantic gap ➔ Debugging gap

- OpenFPM library
  - Modern C++ template library (for CPUs and GPUs)
  - Support for dynamic load-balancing, checkpointing and communication abstractions

- Template meta-programming

\[
\frac{D\omega}{Dt} = (\omega \cdot \nabla)u + \nu \Delta \omega
\]

What we want

What we code
(already quite abstracted!)

P. Incardona, et al "OpenFPM: A scalable open framework for particle and particle-mesh codes on parallel computers", Computer Physics Communications, 2019
Model-to-model code generation

OpenPME DSL

Intermediate representation (IR)


Closing the performance gap

Lennard Jones (particles, discrete)

Gray-Scott (mesh, continuous)

Vortex in Cell (hybrid, continuous)

57 LOC vs 151 LOC

40 LOC vs 100 LOC

73 LOC vs 580 LOC

Missing loop fusion (to merge mesh processing)

Higher-level optimizations

- Insertion of ghost-gets, based on high-level dataflow
- Model-based auto-tunning for discretization
- Theoretical convergence to steer search

With comparable exploration time, oblivious auto-tuners orders of magnitude worse

1x, 8x, 16x more exploration time with various degrees of success
Formal language – added value

- Mathematical expressions: Possible to explore performance-accuracy trade-offs
- Type system: High-level semantics checks (e.g., units)

Examples (3): Big data (only briefly)

- Dataflow IR from a sequential syntax (Rust or Java-like)

```rust
let v1_1_0 = join(customer, sales, cc_sk, c_sk);
let v1_1 = join(v1_1_0, date_dim, sd_sk, d_d_sk);
let gs = group_by(v1_1, gb_key);
let mut v1 = Vec::new();
for group in gs {
    let result = compute(group, sale_type);
    v1.push(result);
}
```

Sequential code (implicit parallelism)

Collaboration with

Monolithic Program

- easy: develop test debug
- hard: update

Manual: high effort

Microservices

- hard: develop test debug update
- easy: update

Barkhausen Institut
Examples (3): Big data (only briefly)

- Dataflow IR from a sequential syntax (Rust or Java-like)
- IR to abstract from “cloud ISAs”

```rust
let v1_1_0 = join(customer, sales, cc_sk, c_sk);
let v1_1 = join(v1_1_0, date_dim, sd_sk, d_d_sk);
let gs = group_by(v1_1, gb_key);
let mut v1 = Vec::new();
for group in gs {
    let result = compute(group, sale_type);
    v1.push(result);
}
```

Sequential code (implicit parallelism)

---

Current work on large-scale EU H2020 project Everest
- Stencil and Tensor Operations in Weather Modelling (WRF)
- Interplay orchestration (dataflow) and kernels
- MLIR framework for reusable abstractions


CIMA Foundation
Challenges ahead: Emerging memories

- Example: Hybrid STT-/DRAM
  - Placement and layout optimization
  - Hints for memory controllers

- Racetrack memories
  - Extreme density
  - Sequential bit access per cell

- Memristive accelerators
  - In-memory computing
  - Compiler abstractions
Architecture and data layout optimization

- Underlying idea: Zig-zag through data to reduce number of shifts
  - Exploit explicit patterns in high-level DSLs
  - Recognize patterns with polyhedral compilers

Latency comparison vs SRAM

- Un-optimized and naïve mapping: Even worse latency than SRAM
- 24% average improvement (even with very conservative circuit simulation)

Higher savings due to less leakage power
74% average improvement (in addition to savings due to DRAM placement)

Generalization to stencils and other kernels

- Average improvements in performance (~20%) and energy consumption (~40%)

Summary

- **Tame ever-increasing system complexity**
  - Still highly-relevant optimizing compilers (polyhedral, ...)
  - DSL examples: expose higher semantics (efficiency, productivity)
  - Higher semantics key for emerging accelerators/systems!

- **Moving forward**
  - Semantic-preserving transformations
  - Larger use cases (e.g., WRF in the context of EVEREST)
  - Common abstraction across novel paradigms (e.g., as MLIR dialects across in-memory computing architectures)
Thanks! & Acknowledgements

Funded by

Deutsche Forschungsgemeinschaft
CO4RTM (Number 450944241)
OpenPME (Number 350008342)

This project has received funding from the European Union’s Horizon 2020 research and innovation programme under grant agreement No 957269

https://everest-h2020.eu

..., and previous members of the group (Norman Rink, Sven Karol, Sebastian Ertel), and collaborators (J. Fröhlich, I. Sbalzarini, A. Cohen, T. Grosser, T. Hoeffler, H. Härtig, H. Corporaal, C. Pilato, S. Parkin, P. Jääskeläinen)
Summary

Tame ever-increasing system complexity
- Still highly-relevant optimizing compilers (polyhedral, …)
- DSL examples: expose higher semantics (efficiency, productivity)
- Higher semantics key for emerging accelerators/systems!

Moving forward
- Semantic-preserving transformations
- Larger use cases (e.g., WRF in the context of EVEREST)
- Common abstraction across novel paradigms (e.g., as MLIR dialects across in-memory computing architectures)
References


