Compilers for Processors and Systems

Jeronimo Castrillon
Chair for Compiler Construction
TU Dresden
jeronimo.castrillon@tu-dresden.de

Forschungslinie – Einführung in die Forschung. Modul INF-D-910
April 2019
Inflection points in computing and programming

The programming interface continues to broaden as hardware evolves.

- Single-core architectures
- Use multi-core architectures (~2005)
- Dark Si: specialize
- Post CMOS?

Programming languages

Architectures & μ-arch

Single core (het.) Multi-processors

© J. Castrillon. Compilers for Processors/Systems

Research at the Chair for Compiler Construction

- Context: The Center for Advancing Electronics Dresden (cfaed)

- Research Overview
  - Programming parallel heterogeneous systems
  - Domain-specific languages and optimization
  - Tools and methodologies for Post-CMOS systems
  - Optimization: Performance, energy efficiency & resilience
In this lecture

- Classical compilers (refresh)
- Multi-cores: Parallel programming
- Emerging topics in compilers
Classical compilers
What is a compiler

- Compiler translates an input source code to a target code
  - Typical: target code closer to machine code (e.g., C → assembly)
  - Must recognize illegal code and generate correct code
  - Must agree with lower layers (e.g., storage, linker and runtime)
- Today: just-in-time compilers, auto-tuning, continuous compilation, …
### Structure of a compiler: Front-end + Middle-end

<table>
<thead>
<tr>
<th>Lexical analysis</th>
<th>Syntax analysis</th>
<th>Semantic analysis</th>
<th>IR-generation</th>
<th>IR-optimization</th>
</tr>
</thead>
<tbody>
<tr>
<td>Tokens</td>
<td>ST</td>
<td>ST</td>
<td>IR</td>
<td></td>
</tr>
</tbody>
</table>

- **Lexical (scanner):** Maps a character stream into words (tokens)
- **Syntax (parser):** Recognizes “sentences of tokens” according to a grammar
- **Semantic analysis:** Adds information and checks – types, declarations, ...
- **IR-generation:** Abstract representation for optimization and code generation
- **IR-optimization:** Simplification & improvements (e.g., remove redundancies)
Structure of a compiler: Backend

- Code selection: Decide which instructions should implement the IR
- Register allocation: Decide in which register to place variables
- Scheduling: Decide when to execute the instructions (e.g., ordering in assembly program) & ensure conformance with interfaces and constraints
while (y < z)
{
    int x = a + b;
    y += x;
}

Loop: x = a + b
     y = x + y
     _t1 = y < z
     if _t1 goto Loop

Abstract representation of the program, closer to the target

x = a + b
_y = x + y
_t1 = y < z
if _t1 goto Loop
Example: LLVM Framework

```c
int main()
{
    int a, b, c;
    a = 2; b = 3; c = 5;
    c += (a*b) >> a;
    c += foo(a);
    return 0;
}
```

```
define i32 @main() #0 {
    %1 = alloca i32, align 4
    %a = alloca i32, align 4
    %b = alloca i32, align 4
    %c = alloca i32, align 4
    store i32 0, i32* %1
    store i32 2, i32* %a, align 4
    store i32 3, i32* %b, align 4
    store i32 5, i32* %c, align 4
    %2 = load i32* %a, align 4
    %3 = load i32* %b, align 4
    %4 = mul nsw i32 %2, %3
    %5 = load i32* %a, align 4
    %6 = ashr i32 %4, %5
    %7 = load i32* %c, align 4
    %8 = add nsw i32 %7, %6
    store i32 %8, i32* %c, align 4
    %9 = load i32* %a, align 4
    %10 = call i32 @foo(i32 %9)
    %11 = load i32* %c, align 4
    %12 = add nsw i32 %11, %10
    store i32 %12, i32* %c, align 4
    ret i32 0}
```

Lots of room for optimization!
Control flow graph: Represent the branching structure of programs

1: a = 5;
2: c = 1;
3: L1: if (a > c) goto L2;
4: c = c + c;
5: goto L1;
6: L2: a = c – a;
7: c = 0;

Basic-blocks: sequence of statements w/o branching in between
Graph representations: data-flow information

- The compiler has to know where the data is coming from for optimizations

```
1: a = 5
2: c = 1
3: L1: if (a > c) goto L2;
4: c = c + c;
5: goto L1;
6: L2: a = c - a;
7: c = 0;
```

Data-flow information: Who could have defined c? (among other questions)

Limits to analysis: Pointers if (a > *pX)
Sample optimization: Function inlining

- Inlining: Replace function call by inserting the function body in the code
- Why
  - Increase potential for other optimizations (w/o complex inter-procedural analysis)
  - Reduce stack management overhead
  - Remove jumps (call and return)
- Common example of the tradeoff between code-size & performance

- Minimal example

```c
int f1(int x)
{ return x+1; }

int f2(int x)
{ return f1(x)+2; }
```

```c
int f1(int x)
{ return x+1; }

int f2(int x)
{ return x+1+2; }
```

```c
int f2(int x)
{ return x+3; }
```
Function inlining (2)

- Lots of support in compilers
- Intuition
  - Static calls impact code size
  - Dynamic calls impact performance (need profiling)
  - When to inline?
    - Small functions (comparable to calling overhead)
    - Single static call
    - Static calls within loops (high dyn. calls)
    - Functions with single switch-case and often called with constant parameters

- `fno-inline`
- `finline-small-functions`
- `findirect-inlining`
- `finline-functions`
- `finline-functions-called-once`
- `fearly-inlining`
- `finline-limit=n`
- `fno-keep-inline-dllexport`
- `fkeep-inline-functions`
- `fpartial-inlining`
- `flto[=n]` (for linked-time opt)

--param name=value
  - max-inline-insns-single
  - max-inline-insns-auto
  - inline-min-speedup
  - large-function-insns
  - large-function-growth
  - large-unit-insns
  - inline-unit-growth
  - max-inline-insns-recursive
  - max-inline-insns-recursive-auto

...
Consider a call graph \( CG = (V,E,B,C,D) \), with a node for every function, an edge \( e = (f_i, f_k) \) if \( f_i \) calls \( f_k \) and

- \( B(f_i) \) the code size of \( f_i \) w/o inlining
- \( C(e) \) the number of static calls of \( f_k \) in \( f_i \)
- \( D(f_i) \) the number of dynamic calls to \( f_i \)

**Code size:** Still quite alive (best-paper award CGO’19)

**Problem formulation:** Inline a set of functions so that performance is optimized while keeping the code size below a threshold \( L \)

- Not trivial due to **mutual dependence** between inlining of different functions
Function inlining: Formulation

- Given a CG = (V, E, B, C, D), let \( b_i = \begin{cases} 0 & \text{if } f_i \text{ not inlined} \\ 1 & \text{if } f_i \text{ inlined} \end{cases} \)

- Find inlining \( B = (b_1, \ldots, b_{|V|}) \in \{1, 0\}^{|V|} \) such that \( D(B) = \sum_{i: b_i = 0} D(f_i) \) is minimized, subject to a size constraint

\[
S(B) = \sum_{i=1}^{|V|} S(f_i) < T,
\]

\[
S(f_i) = B(f_i) + \sum_{j: b_j = 1} C(f_i, f_j) \cdot S_j
\]

Assumption: If inlined, then for all static call sites at once

Recursive computation (does not support cyclic call graphs)
Solution approaches

- Branch & bound solution: Expensive but amortizable for embedded applications
- Auto-tuning for dynamically compiled languages
  - J. Cavazos and M. F. P. O'Boyle, "Automatic Tuning of Inlining Heuristics," SC’05
- Using machine learning

Multi-cores: Parallel programming

- Introduction
- Auto-parallelization
- Dataflow programming
- Domain-specific languages
Multi-cores: Parallel programming

- Introduction
- Auto-parallelization
- Dataflow programming
- Domain-specific languages
Challenges in multi-core compilation

- Deal with mapping, scheduling, synchronization
- Different OSes
- Different APIs
- Compile vs. runtime
Multi-core compilers

- Deal with similar problems than classical compilers
  - Parse and understand high-level parallel language constructs
  - Search for parallelism, but at a higher level of abstraction (higher than ILP)
  - Requires a model of the target architecture, but at coarser level
  - Allocation and scheduling of data to memories and tasks to processors
  - Code generation via source-to-source compilation
On parallel programming models

- There are many, each with a different impact on compilers
- Sequential programming models: C/C++, Matlab, ...
- Parallel programming models
  - Shared memory: pthreads, OpenMP, Intel TBB, Cilk, ...
  - Distributed memory: MPI, Charm++, ...

- In this lecture: More automatism (not all general purpose)
  - Extracting coarse-grained parallelism from C code
  - Parallel dataflow models
  - Domain-specific languages
Multi-cores: Parallel programming

- Introduction
- Auto-parallelization
- Dataflow programming
- Domain-specific languages
Principle of operation

- Similar compiler flow, but more challenges
  - More aggressive data flow analysis
  - More aggressive program transformations
  - Different granularity (basic-blocks?, functions?)
  - Focus on coarse-grained parallelism patterns
  - Whole program analysis

Source code → Frontend → Middle-end → Backend → Target code

Architecture Model

Parallel implementation (e.g., OpenMP, pthreads, MPI,...)
Data-flow analysis

- **Dynamic** data flow analysis via execution traces
  - More exact: Find exact portions of memory being read/written
  - Not sound: Cannot completely rely on dynamic information
int main(void)
{
    float x = 0.0;
    float y = 0.0;
    float z = 0.0;
    float a = 9.2;

    for (int i = 0; i < MAIN_LOAD_PLP;
    {
        x = calculate_pi();
        y = doSomeWork(x / 2);
        y = y + a;
        z = z + doSomeMoreWork(y / 4);
    }

    printf("z=%f\n", z);
    return 0;
}
Granularity for analysis
- Depends on the target platform
- Whole-program information: Costs of called functions
- Need to take communication into account
- Is not given by traditional compiler boundaries: basic-blocks or functions

→ Use graph clustering algorithms
Coarse-grained parallelism patterns

- Search for known parallelism patterns
  - Task-level parallelism
  - Data-level parallelism
  - Pipeline-level parallelism
  - Others: Reduction, commutative operations, ...

Task Level Parallelism (TLP)

Data Level Parallelism (DLP)

Pipeline Level Parallelism (PLP)
Example: Generating OpenMP

- EP: Application in the NAS parallel benchmark
  - Automatic identification of DLP and private variables (20+ for some loops)
  - Automatic identification of Reduction pattern
- Very complex graphs
Example: Generating OpenMP (2)

- Manually parallelized code
  - Multiple nesting
  - Multiple annotations

```c
#pragma acc parallel loop reduction(+:sx,sy)
for (k = 1; k <= np; k++)
{
    kk = k_offset + k;
    t1 = S;
    t2 = an;

    /* Find starting seed t1 for this kk */
    #pragma acc loop seq
    for (i = 1; i <= 100; i++)
    {
        ik = kk / 2;
    }
```

OpenMP Annotated Code

- Reference pragma
- Generated pragma
- Reduction Correctly Identified
- Several Private Variables
Problems with auto-parallelization

- Difficult to find all dependencies (often impossible at compile-time)
- Coding style and the illusion of infinite shared memory
- Dependencies can sometimes be violated!

```c
for (i = 1; i <= 100; i++)
    for (j = 1; j <= 100; j++) {
        S1: X[i][j] = X[i][j] + Y[i-1][j];
        S2: Y[i][j] = Y[i][j] + X[i][j-1];
    }
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 an adjacent node hasn't been visited,
    // then mark it as visited and enqueue it
    for(i=adj[s].begin(); i!=adj[s].end(); ++i) {
        if(!visited[*i])
        {
            visited[*i] = true;
            queue.push_back(*i);
        }
    }
    return result;
}
```
Multi-cores: Parallel programming

- Introduction
- Auto-parallelization
- Dataflow programming
- Domain-specific languages
Sample compiler flow

KPN Application

Non-functional specification

Architecture model

Analysis

Synthesis

Code generation

PNargs_ifft_r.ID = 6U;
PNargs_ifft_r.PNchannel_freq_coef = filtered_coef;
PNargs_ifft_r.PNnum_freq_coef = 0U;
PNargs_ifft_r.PNchannel_time_coef = sink_right;
PNargs_ifft_r.channel = 1;
sink_left = IPC1l1mrf_open(3, 1, 1);
sink_right = IPC1l1mrf_open(7, 1, 1);
PNargs_sink.ID = 7U;
PNargs_sink.PNchannel_in_left = sink_left;
PNargs_sink.PNnum_in_left = 0U;
PNargs_sink.PNchannel_in_right = sink_right;
PNargs_sink.PNnum_in_right = 0U;
taskParams.arg0 = (xdc_UArg)&PNargs_src;
taskParams.priority = 1;
ti_sysbios_knl_Task_create((ti_sysbios_knl_Task_FuncPtr
&taskParams, &eb);
glob_proc_cnt++;
hasProcess = 1;
taskParams.arg0 = (xdc_UArg)&PNargs_fft_l;
taskParams.priority = 1;
ti_sysbios_knl_Task_create((ti_sysbios_knl_Task_FuncPtr
&taskParams, &eb);
glob_proc_cnt++;
hasProcess = 1;
taskParams.arg0 = (xdc_UArg)&PNargs_ifft_r;
taskParams.priority = 1;
ti_sysbios_knl_Task_create((ti_sysbios_knl_Task_FuncPtr
&taskParams, &eb);
glob_proc_cnt++;
hasProcess = 1;
taskParams.arg0 = (xdc_UArg)&PNargs_sink;
taskParams.priority = 1;
ti_sysbios_knl_Task_create((ti_sysbios_knl_Task_FuncPtr
&taskParams, &eb);
glob_proc_cnt++;
hasProcess = 1;
taskParams.arg0 = (xdc_UArg)&PNargs_sink;
taskParams.priority = 1;
Kahn Process Networks (KPNs)

- Graph representation of applications
  - Processes communicate only over FIFO buffers
  - Good model for streaming applications
  - Good match for signal processing & multi-media (cross-domain)

- Stereo digital audio filter

![Stereo digital audio filter diagram]
Language: C for process networks

- **FIFO Channels**
  ```c
  typedef struct { int i; double d; } my_struct_t;
  __PNchannel my_struct_t S;
  __PNchannel int A = {1, 2, 3}; /* Initialization */
  __PNchannel short C[2], D[2], F[2], G[2];
  ```

- **Processes & networks**
  ```c
  __PNkpn AudioAmp __PNin(short A[2]) __PNout(short B[2])
  __PNparam(short boost){
    while (1)
      __PNin(A) __PNout(B) {
        for (int i = 0; i < 2; i++)
          B[i] = A[i]*boost;
      }
  __PNprocess Amp1 = AudioAmp __PNin(C) __PNout(F) __PNparam(3);
  __PNprocess Amp2 = AudioAmp __PNin(D) __PNout(G) __PNparam(10);
  ```

Architecture model for heterogeneity

- System model including:
  - Topology, interconnect, memories
  - Computation: cost tables (as backup)
  - Communication: cost function (no contention)
- Example: Texas Instruments Keystone

M. Odendahl, et al., “Split-cost communication model for improved MPSoC application mapping”,
In International Symposium on System on Chip (SoC) pp. 1-8, 2013
Constraints

- Timing constraints
  - Process throughput
  - Latencies along paths
  - Time triggering
- Mapping constraints
  - Processes to processors
  - Channels to primitives
- Platform constraints
  - Subset of resources (processors or memories)
  - Utilization
Static models: Synchronous Dataflow (SDF)

- Fully specified rates, allow more compiler analysis
- Transformation: repetition vector serve to unroll the graph [1 3 2]
- Perform mapping and scheduling on the resulting directed acyclic graph (DAG)
Dynamic models: KPNs

- Channel accesses not visible at the graph
  - Need to look inside the processes

- Solutions: Use dynamic scheduling
- Methods: Employ simulations, genetic algorithms or devise heuristics
Analysis and synthesis: Overview

- CPN application
- Architecture model
- Non-functional specification

**Analysis: Instrumentation, profiling, tracing**
- Sequential performance estimation
  - Time-annotated traces
  - Mapping and scheduling
  - Parallel perf. estimation
  - Increase resources

**Mapping configuration**
Tracing: Dealing with dynamic behavior

- KPNs do not have firing semantics
- **White model of processes**: source code analysis and tracing
- Tracing: instrumentation, token logging and event recording

```plaintext
... for (;i < x;i++) {
    write(&c2);
    f1(...);
    read(&c1);
    f2(...);
    read(&c1);
    ...
```

Performance estimation for time between events
Parallel performance estimation

- Discrete event simulator to evaluate a solution
  - Replay traces according to mapping
  - Extract costs from architecture file (NoC modeling, context switches, communication)
Trace-based synthesis

- Synthesis based on code and trace analysis (using simple heuristics)
  - Mapping of processes and channels
  - Scheduling policies
  - Buffer sizing

Sample results from mapping exploration

Iterative Mapping

Real-time algorithm
Real-time constraint

Makespan (Gcycles)

MJPEG Trials

Config.: 3 PEs

Config.: 5 PEs

© J. Castrillon. Compilers for Processors/Systems
Flexible mappings: Run-time analysis

- Modified linux kernel to make it aware of symmetries
- Platform: Odroid XU4 (big.LITTLE)
- Multi-application scenarios: audio filter (AF) and MIMO
  - 1 x AF,
  - 4 x AF
  - 2 x AF + 2 x MIMO
- 3 mappings to two processors
  - T1: Best CPU time
  - T2: Best wall-clock time
  - T3: GBM heuristic

Goens, A. et al. “TETRIS: a Multi-Application Run-Time System for Predictable Execution of Static Mappings”, SCOPES’17
Flexible mappings: Multi-application results (1)

Way more predictable performance

Comparable performance to dynamic mapping
Multi-cores: Parallel programming

- Introduction
- Auto-parallelization
- Dataflow programming
- Domain-specific languages
What are DSLs?

- DSLs help bridge the gap between problem domain and general purpose languages

Adapted from lecture: “Concepts of Programming Languages”, Eelco Visser, TU Delft
What are DSLs?

- DSLs help bridge the gap between problem domain and general purpose languages

- Natural vocabulary for concepts are fundamental to problem domain

- Faster way to write common concepts (concrete syntax)

- Explanation is a domain-specific, task-specific, user-specific inference

- Optimization potential due to domain-specific information

Adapted from lecture: “Concepts of Programming Languages”, Eelco Visser, TU Delft

Why DSLs now?

- DSLs help bridge the gap between problem domain and general purpose languages

Adapted from lecture: “Concepts of Programming Languages”, Eelco Visser, TU Delft

- Programming permeating other disciplines with their own vocabulary
  - “High-level” language receives a different connotation
- Complexity of hardware, low-level APIs, low-level software
- Push from different domains abusing existing less powerful tools (e.g., XML)
- Maturity of tools (interpreters, compilers, IDEs, code generators)
**DSL: Definitions**

A **DSL** is a computer language **specialized** to a particular application domain

> A **DSL** is a programming language or executable specification language that offers, through **appropriate notations and abstractions**, expressive power focused on, and usually restricted to, a particular problem domain


A **DSL** is a computer programming language of **limited expressiveness** focused on a particular domain

Working on different fronts, touching different topics
- Performance: Avoid the “abstraction toll”
- Type systems, operational and denotational semantics
- Ohua
  - Functional abstraction on top of dataflow execution models
  - Applicable to system’s programming
- PPME: Parallel particle-mesh programming environment
  - For particle-based simulation
  - Applicable, e.g., for computational biology
- CFDlang: for computational fluid dynamics
  - Tensors and tensor operations in the syntax
Ohua: Implicit parallelism for systems

- Functional programming abstraction to implicitly create dataflow graphs
  - Functional program: High-level algorithm (clojure)
  - State-full functions: actual application logic (clojure, java, scala, ...)

```clojure
(ohua :import [web.translation]) ; import the namespace where the used
    ; functions are defined
(defn translate [server-port]
  (ohua (let [[cnn req] (read-socket (accept (open server-port)))]
    [_file-name _ lang] (parse-request req)
    ["List content length"] (if (exists? file-name)
      (load-file-from-disk file-name)
      (generate-reply "No such file.")
    )
    "String word (decompose content); poor man's translation"
    _ (log "translating word")
    updated-content (collect length (translate word lang))]
  (reply cnn (compose length updated-content)))
```

Ohua compiler

- Clean language semantics and state characterization allow automatic derivation of dataflow graph

---


© J. Castrillon. Compilers for Processors/Systems
Ohua applications: Map & Reduce

- Data processing pipeline within Hadoop
- Automatic exploitation of parallelism by compiler and runtime

Ohua applications: Micro-services

- Automatic batching of I/O in micro-services: Via graph rewrites
- Similar performance than e.g., Facebook, with better code style

PPME: Parallel particle-mesh environment

- Domain: Python-based DSL for Particle-Mesh simulations
  - Parallel particle-mesh library (PPM): abstractions for computational biology
  - Collaboration with the mosaic group (http://mosaic.mpi-cbg.de/)

- Provide more information about structure, data-usage and computation patterns to the compiler for parallel execution
PPME: Data model

**PPME: Type system**

- **High-level semantics checks (e.g., units)**

\[
\begin{align*}
\text{VAR} & \quad \Gamma(v) = \tau \\
\Gamma \vdash v : \tau \\
\text{VARDECL} & \quad \Gamma \vdash \tau x : \Gamma \cup \{x = \tau\} \\
\text{VARINIT} & \quad \Gamma \vdash e : \tau' \quad \tau' \leq \tau \\
\Gamma \vdash x : \tau \\
\Gamma \vdash e : \tau' \\
\Gamma \vdash x = e : \Gamma \cup \{x = \tau\} \\
\text{p} \to \text{a} & \quad \Gamma \vdash p \to F : [\mathbb{R}, m] \\
\Gamma \vdash mass : [\mathbb{R}, m] \\
\Gamma \vdash p \to F/\text{mass} : [\mathbb{R}, m] \\
\text{delta}_t^2 & \quad \Gamma \vdash \text{delta}_t^2 : [\mathbb{R}, t^2] \\
\text{E} & \quad \Gamma \vdash p \to v : [\mathbb{R}, v] \\
\Gamma \vdash 0.5 : [\mathbb{R}, 0] \\
\Gamma \vdash p \to a + p \to F/\text{mass} : [\mathbb{R}, a] \\
\Gamma \vdash 0.5 \ast (p \to a + p \to F/\text{mass}) : [\mathbb{R}, a] \\
\Gamma \vdash p \to a \ast p \to F/\text{mass} : [\mathbb{R}, a] \\
\Gamma \vdash v : [\mathbb{R}, v] \\
\Gamma \vdash 0.5 \ast (p \to a + p \to F/\text{mass}) \ast \text{delta}_t^2 : [\mathbb{R}, a \cdot t^2] \\
\text{ERRUNARY} & \quad \Gamma \vdash e : \tau \\
\Gamma \vdash \phi(e) : \text{E} \\
\text{ERRBIN} & \quad \Gamma \vdash e_1 : \tau_1 \\
\Gamma \vdash e_2 : \tau_2 \\
\Gamma \vdash \phi(e_1, e_2) : \text{E} \\
\phi & \in \{-, !, \sqrt{\cdot}, \times, \div, /, \text{\&\&}, \text{||}, \text{rel}\} \\
\text{Z} & = \text{Integer}, \text{R} = \text{Real}, \text{P} = \text{Particle}, \text{V} = \text{Vector}, \text{M} = \text{Matrix}, \text{E} = \text{Field/Property}, \text{E} = \text{Error}
\end{align*}
\]

PPME: Language

- DSL: Closer to domain expert
- Easier to execute in parallel
- Integration with other tools for expression optimization
- Example: Gray-Scott simulation
Tensor expressions typically occur in numerical codes

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

- Tensor product notation popular in the CFD domain

- On performance
  - Matrixes are small, so libraries like BLAS don’t always help
  - Expressions result in deeply nested for-loops
  - Performance highly depend on the shape of the loop nests

- No need to do complex polyhedral analysis if we know the tensors and the semantics of the operators!
CFDlang and tool flow

source =
type matrix : [mp np] &
type tensorIN : [np np np ne] &
type tensorOUT : [mp mp mp me] &

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

CFDlang and tool flow

- DSL: Embedded in Fortran, including APIs for calling and optimizing the kernel
- Tensor intermediate representation (IR)
  - Representation of iterators and implicit loops
  - Commands for loop transformations (parallelization, fusion, interchange, ...)
- Code generation
  - Just-in-time generation of C code (for autotuning)

Fortran embedding:

```fortran
source =

type matrix : [mp np] &
type tensorIN : [np np np ne] &
type tensorOUT : [mp mp mp me] &

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
```
Example: Interpolation operator

- **Interpolation:**
  \[ v_e = (A \otimes A \otimes A) u_e \]
  \[ v_{ijk} = \sum_{l,m,n} A_{kn} \cdot A_{jm} \cdot A_{il} \cdot u_{lmn} \]

- **Three alternative orders (besides naïve)**
  
  **E1:**
  \[ v_{ijk} = \sum_{l,m,n} (A_{kn} \cdot (A_{jm} \cdot (A_{il} \cdot u_{lmn}))) \]

  **E2:**
  \[ v_{ijk} = \sum_{l,m,n} (A_{kn} \cdot A_{jm}) \cdot (A_{il} \cdot u_{lmn}) \]

  **E3:**
  \[ v_{ijk} = \sum_{l,m,n} (A_{kn} \cdot ((A_{jm} \cdot A_{il}) \cdot u_{lmn})) \]

---


TeML: Meta-programming for Tensor Optimizations

- Extend grammar and semantics with semantic-preserving, composable transformations
  - For optimization expert or for automated search

\[
(L\text{expression}) \ ::= \ \text{build} (\langle id \rangle) \\
\quad | \ \text{stripmine} (\langle id \rangle, \langle int \rangle, \langle int \rangle) \\
\quad | \ \text{interchange} (\langle id \rangle, \langle int \rangle, \langle int \rangle) \\
\quad | \ \text{fuse\_outer} (\langle id \rangle, \langle id \rangle, \langle int \rangle) \\
\quad | \ \text{fuse\_inner} (\langle id \rangle, \langle int \rangle) \\
\quad | \ \text{unroll} (\langle id \rangle, \langle int \rangle)
\]

TeML: Results

- Extra control allow for new optimization (vs pluto): changing shapes
- General tensor semantics allow covering more benchmarks than TensorFlow

Emerging topics in compilers
Cfaed and Haec

- Excellence Cluster Cfaed
  - New devices/paradigms: Silicon nanowires, carbon nanotubes, organics, bio-inspired computing, ...
  - Software: Domain-specific languages, micro-kernels, formal analysis methods, ...


- HAEC: Highly adaptive energy efficient computing
  - On board optical interconnects
  - Wireless chip-to-chip communication

Emerging techs. example: Racetrack memories

- Racetrack memories: one of many future alternatives
- Predicted extreme density at low latency
  - 3D nano-wires with magnetic domains
  - One port shared for many bits
  - Domains move at high speeds (1000 ms\(^{-1}\))

- Sequential: Game changer for current HW/SW stack
  - Memory management
  - Integration with other memory architectures
  - Data layout and allocation

Sample problem: Variable allocation on the stack

- Order of allocation not a problem with random-access

Variables (V): a b c d e f

Memory trace (S): b c a e f d a c e d a c a d e f

Efficient data placement in RTM can reduce shifts by more than 200%!
"New" compiler optimization

- Model the accessing variables with a mapping given by $\beta: V \rightarrow N$

$$C = \left( \sum_{i=0}^{\mid S \mid - 2} \Delta(S_i, S_{i+1}) \right)$$

$$\Delta(u, v) = \mid \beta(u) - \beta(v) \mid \quad \forall u, v \in V$$

- Linearization: Integer Linear Programming (details omitted, 8 constraints)

$$C = \min \left( \sum_{i=0}^{n-1} \sum_{j=i+1}^{n-2} w_{vi}v_{i} \cdot (p_{ij} + q_{ij}) \right)$$

Results

- Comparing previous heuristic with our heuristic, and when possible against ILP
  - Comparison of offset assignment heuristics

![Bar chart showing comparison of heuristics](chart.png)

Results

- Comparing previous heuristic with our heuristic, and when possible against ILP

Data-layout optimization

- Basic architecture
Data-layout optimization: Tensor contraction

- Optimized layout

Results: Tensor contractions on racetracks

- Improved latency and energy consumption

---

Results: Tensor contractions on racetracks

- Improved latency and energy consumption

Summary
In this presentation

- Basics topics on compilers
- Programming heterogeneous multi and many-cores
  - Efforts to extract parallelism
  - Cross-domain dataflow models
  - DSLs: domain-specific higher level of abstraction
    - Cleaner syntax and semantics: Semantic preserving optimizations
- Insight into new problem formulations for emerging computer architectures

- Outlook
  - Right balance: Compile vs. run-time
  - New goals: Energy efficiency & resilience
Thanks for the attention!
Questions?