Embedded manycore programming: From auto-parallelization to domain specific languages

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

Keynote: International Symposium on Embedded Multicore/Many-core Systems-on-Chip (MCSoC’19)
Nanyang Technical University, Singapore. October 2 2019
Naturally powered by: Moore’s law, power density wall, …
Incredible evolution over the last decades

SoCs: Long history of specialization and interaction with environment

1999

2019

Sketch of TI Keystone II

Systems on Chip (SoC): Evolution (3)

- Incredible evolution over the last decades
- SoCs: Long history of specialization and interaction with environment

**1999**

**2019**

SoC programming: Evolution

- Auto-parallelization
- Formal model-based code/HW generation
- Higher-level programming abstractions

```
A = placeholder((m, h), name='A')
B = placeholder((n, h), name='B')
k = reduce_axis((0, h), name='k')
C = compute((m, n), lambda i, j:
    sum(A[k, i] * B[k, j], axis=k))
```
PnTransformSdfToKpn(D, S);
PnTransformToArrayAccess(D, S);
CollectChannelAccessRanges(D, S);
PropagateChannelAccessRanges(D, S);
PnStreamFactory streamFactory(BasePath);
switch (transTarget) {
  case TransMVP:
    PnTransformMVP(D, S);
    ErasePnProcessTemplates(D);
    PnPrintForMVP(D, S);
    break;
  case TransPthread:
    PnTransformPthreads(D, S, traces);
    ErasePnDefs(D);
    break;
  case TransSystemC:
    PrintForSystemC(D, S, traces, streamFactory);
    ErasePnDefs(D);
    break;
  case TransVPUtg:
    PrintForVPUtg(D, S, streamFactory);
    ErasePnDefs(D);
    break;
  case TransVPUmap:
    PrintForVPUmap(D, S, strMappingFileName, streamFactory);
    ErasePnDefs(D);
    break;
  case TransInvalid:
    assert(false);
    break;
  default:
    break;
}
Theorem (Allen/Kennedy): Any reordering transformation that preserves every dependence in a program preserves the meaning of that program.
Problems for auto-parallelizing compilers

1) Find all dependencies?

```
for (i = 1; i <= 100; i++)
    for (j = 1; j <= 100; j++) {
        X[i][j] = X[i][j] + Y[i-1][j];
        Y[i][j] = Y[i][j] + X[i][j-1];
    }
```

More often than not, impossible!

2) Coding style and the illusion of infinite shared memory

Example: Polyhedral compilation

```
struct a {
    struct s *a;
    int *pi;
    struct t *p;
    float *p;
    ...
}
```
1) Find all dependencies?

2) Coding style and the illusion of infinite shared memory

3) Dependencies can sometimes be violated! (they are artifacts of style)

```plaintext
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;
```

[Edler18]
Dynamic information for auto-parallelization

- Static analysis: Limited applicability (cf. Polyhedral compilation)
- Dynamic analysis: Extract dependencies based on application tracing
  - Summarize information in form weighted control-data flow graphs

```
int A[10];
int foo() { int A[10]; return A[5];
    int s = 0, i;
    A[8] = foo();
    for (i = 0; i < 2; i++) {
        s += A[i * 4];
    }
    return 0;
}
```

- Related approaches: Profile-driven and ML-based mapping, hierarchical task graphs, and Sambamba

[Cordes10] [Streit12] [Tournavitis09] [DAC08, Springer14]
Parallelism extraction

- Clustering: Right granularity?
- Identify patterns: data, task, pipeline, divide and conquer, ...

- Requires
  - Cost model of computation
  - Cost model of communication
  - Abstract notion of time for dependencies

[Leupers'19, Aguilar'18]
Results from a decade of work

<table>
<thead>
<tr>
<th>Step</th>
<th>Speedup</th>
<th>No. of PEs</th>
<th>Parallel Efficiency</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>3.61x</td>
<td>16</td>
<td>22.58%</td>
</tr>
<tr>
<td>2</td>
<td>5.48x</td>
<td>17</td>
<td>32.3%</td>
</tr>
<tr>
<td>3</td>
<td>5.48x</td>
<td>16</td>
<td>34.3%</td>
</tr>
<tr>
<td>manual</td>
<td>9.43x</td>
<td>19</td>
<td>49.6%</td>
</tr>
</tbody>
</table>

Table 1: Summary of JPEG Encoder Parallelization by MAPS

2008
Experimental multi-core from Tokyo Institute of Technology (Prof. Isshiki)

2018
Polybench on real Jetson TX1

[2008]
(DAC08)

[2018]
(Aguilar'18)
Dataflow and hybrid DSE
Dataflow programming

- Graph representation of applications
  - Implicit repetitive execution of tasks
  - Good model for streaming applications
  - Good match for signal processing & multi-media

- The why
  - Explicit parallelism
  - Often: Determinism
  - Better analyzability (scheduling, mapping, optimization)
Dataflow compilation

- Plenty of research on
  - Formal Models of Computation (MoCs)
  - Language, compiler and mapping algorithms
  - Hardware modeling, performance estimation
  - Runtime systems
  - Code generation for heterogeneous multicores

[IEEE TII’13, Springer14]
Static vs dynamic models

- Large body of successful work on static models like synchronous dataflow
  - Schedule and bounds computable at compile time!

- Dynamic models: Data-dependent behavior

... for (;i < x;i++) {
    write(&c2);
    f1(...);
    read(&c1);
    f2(...);
    read(&c1);
    ...
}
Automatic mapping to heterogeneous many-cores

- Lots of research on fixed design-time mapping algorithms (e.g., using genetic algorithms), e.g., Sesame, DOL
- We worked on trace-based heuristics

Multimedia apps on TI Keystone II

[Erbas06, Thiele07]

[DAC’12]

[MCSOC’16]
Support for HW acceleration in SW-defined radio

- **Application:** MIMO OFDM receiver
- **Hardware**
  - Platform 1: Baseline software
  - Platform 2: Optimized software
  - Platform 3: Optimized SW + HW

![Graph showing achieved rate @ 100 MHz]

Library 1: SW implementations

Library 2: SW optimized

Library 3: SW+HW accel.

[SDR’10, ALOG’11]
System dynamics: Hybrid mapping

- Applications not so static anymore!
- Hybrid DSE: a compile and run-time approach
  - Enable adaptivity: malleable, multi-variant
  - Run-time predictability, robustness & isolation
Data-level parallelism: Scalable and adaptive

- Change parallelism from the application specification
- Static code analysis to identify possible transformations (or via annotations)
- Implementation in FIFO library (semantics preserving)
- Similar to AdaPNet or parameterized SDFs

[PARMA-DITAM’18]

© Prof. J. Castrillon. MCSoc. Singapore, 2019
Exploiting symmetries

- Intuition
  - SW: Some tasks/processes/actors may do the same
  - HW: Symmetric latencies (CoreX ↔ CoreY)
  - Symmetry: Allows transformations w/o changing the outcome

  ➔ No need to analyze all possible mappings (prune search space)
  ➔ Work on formalization via inverse semi-groups and efficient algorithms
Flexible mappings: Generalized Tetris

- Given multiple canonical configs by compiler, select one at run-time
- Exploit mapping equivalences and similarities

[SCOPES'17b]
Flexible mappings: Run-time analysis

- Linux kernel: symmetry-aware
- Target: 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 [DAC’12]

Single AF

[SCOPES’17b]

© Prof. J. Castrillon. MCSoC. Singapore, 2019
Flexible mappings: Multi-application results (1)

More predictable performance

Comparable performance to dynamic mapping

Wall-clock time [s]

CPU time [s]

[COPES'17b]

instance 1 2 3 4

Mode CFS Dyn T1 T2 T3
Robustness

- Static mappings, transformed or not, provide good predictability
- However: Many things out of control
  - Application data, unexpected interrupts, unexpected OS decisions

→ Can we reason about robustness of mapping to external factors?
Design centering

- Design centering: Find a mapping that can better tolerate variations while staying feasible
- Studied field, in e.g., biology, circuit design or manufacturing systems.

- Currently
  - Using a bio-inspired algorithm
  - Robust against OS changes to the mapping

[SCOPES'17a]
Evaluation

- Analyze how robust the center really is
  - Perturbate mappings and check how often the constraints are missed
  - Signal processing applications on clustered ARM manycore and NoC manycore (16)

![Graph showing mappings passed in % for ARM SoC and NoC architectures.](image-url)
Evaluation

- Analyze how robust the center really is
  - Perturbate mappings and check how often the constraints are missed
  - Signal processing applications on clustered ARM manycore and NoC manycore (16)

[SCOPES'17α]
Ongoing work: Improve representations

- Work on embeddings: Architectures $\rightarrow$ Real numbers
- Novel mapping representations: faster heuristics & more efficient heuristics?
- Example: T-SNE Visualization for mappings space (8 tasks on Odroid XU4)

[MCSoC’18]
Higher-level programming abstractions

A = placeholder((m,h), name='A')
B = placeholder((n,h), name='B')
k = reduce_axis((0, h), name='k')
C = compute((m, n), lambda i, j: sum(A[k, i] * B[k, j], axis=k))
ML revolution: Frameworks and architectures

- Many existing frameworks, e.g., TVM, Tensor Comprehensions, TensorFlow, ...

- Lots of traction in hardware architectures: TPU, V100, ...

Example flow: TVM  [Chen, OSDI’18]
Domain-specific abstractions

- Commonality: Tensor expression languages
- Increase programmer’s productivity
- From compiler perspective: No abstraction toll
  - Easier access to information
  - Larger scope for optimization

\[
\begin{align*}
\text{var input } A & \quad : \text{matrix} \\
\text{var input } u & \quad : \text{tensorIN} \\
\mathbf{v} & = (A \# A \# A \# u) \\
& = ([5, 8] [3, 7] [1, 6])
\end{align*}
\]

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

[RWDSL'18, GPCE'17]
TeML: Results

- Extra control allows for new optimization (vs pluto): changing shapes
- General tensor semantics allows covering more benchmarks than TensorFlow
Emerging technologies: Racetrack memories

- Predicted extreme density at low latency
  - 3D nano-wires with magnetic domains
  - One port shared for many bits
  - Domains move at high speeds ($1000 \text{ ms}^{-1}$)

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

[Parkin-Nature’15]
Simulating RTMs

- RTSim: Configurable racetrack simulator
  - Allows running software benchmarks
  - Built on top of other simulator technology: NVMAIN 2, Gem5, SystemC, …

[IEEE CAL’19]

https://github.com/tud-ccc/RTSim
Architecture and data layout optimization

- Architecture – software co-optimization
  - Embedded system for inference: RTM as scratchpad with pre-shifting and other optimizations

![Diagram of architecture and data layout optimization](image)

[LCTES'19]
Data-layout: Reduce the number of shifts

- **Compulsory shifts**
- **Overhead shifts**
Latency comparison vs SRAM

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

![Graph showing latency comparison between SRAM, RTM-naïve, RTM-opt, and RTM-opt-ps for different tensor sizes. The graph includes a legend indicating the colors used for each category: SRAM (black), RTM-naïve (orange), RTM-opt (green), and RTM-opt-ps (red). The x-axis represents different tensor sizes (4, 8, 16, 32, 64, 128, 256, 512, 1024, 2048, and Avg). The y-axis represents normalized latency. The bars show the latency for each tensor size and category, with RTM-opt-ps generally having the lowest latency.]

[LCTES’19]
Energy comparison vs SRAM

- Higher savings due to less leakage power
- 74% average improvement

[Diagram showing energy comparison across different tensor sizes (4, 8, 16, 32, 64, 128, 256, 512, 1024, 2048) for SRAM, RTM-naïve, RTM-opt, and RTM-opt-ps. The diagram indicates that RTM-opt-ps consistently has the lowest normalized energy across all tensor sizes.]
Discussion
Summary

- Past ten years of efforts to handle the ever increasing complexity of SoCs
  - Advances in auto-parallelization (polyhedral, dynamic analysis)
  - Explicit parallel MoC-based programming models
  - Quest for more adaptivity but retaining time predictability
  - Higher-level abstractions: Example for tensor-based computations for accelerators

- Lots of challenges remain (thankfully)
  - Cost models and characterization of trade-offs (vs. blind searches)
  - Understand impact of emerging technologies
  - Syntax and semantics (for correctness): Lots of open questions
  - Time-semantics in programming and execution environments
References