Compiling a Gesture Recognition Application for an Ultra Low-Power Architecture

Phitchaya Mangpo Phothilimthana
Michael Schuldt

Rastislav Bodik

Berkeley

UNIVERSITY OF CALIFORNIA

UNIVERSITY of WASHINGTON
Classification Applications

- Intensive computation
- Large data storage for gesture models

- Not enough space for data and program
- Not enough energy
- Too slow
GreenArrays: Ultra Low Power Processor

- 144 cores
- No cache, no shared memory
- Stack-based 18-bit architecture
- 32 instructions

Programming challenges
- Manually partition a program across 144 cores
- Explicitly manage communication between cores
- Program in assembly-like stack-based language
How to program and compile for highly-constrained multicore processors with very small distributed memory?

**Key idea:** partition a program smartly to fit the code in tiny cores and achieve fast execution time by balancing *communication* and *code replication*.

In the context of partitioning both:
1. data & computation
2. control flow statements
**Existing Solution**

**Chlorophyll**

[Phothilimthana et al. PLDI’14]

**Input:** imperative sequential program

**Output:** per-core assembly programs

**Compilation strategy:** similar to SPMD
- Output per-core programs have the same structure (control flow).
- Data and computation are distributed across cores.

**Extra input:**
partial constraints on data and computation partitioning
int x[12];
for(i from 1 to 12)
  x[i] += x[i-1];
int[4]@[1,2,3] x[12];
for(i from 1 to 12)
  x[i] += x[i-1];

Partition Type
pins data and operators
to specific partitions
(logical cores)

Spatial programming model

Similar to [Chandra et al. PPoPP’08]
int[4]{1,2,3} x[12];
for(i from 1 to 12)
    x[i] += @loc(x[i]) x[i-1];

Spatial programming model

Partition Type
pins data and operators to specific partitions (logical cores)

Similar to [Chandra et al. PPoPP’08]
Int[4] x[12];
for(i from 1 to 12)
     x[i] += x[i-1];
Incomplete Annotations

```c
int[4] x[12];
for(i from 1 to 12)
    x[i] += x[i-1];
```

**Hard constraint:**
Code fits in each logical core (partition).

**Objective:**
Minimize number of messages sent between partitions.

Program + some partition annotations → Partition Type Inference → Complete partition annotations
## Problems

<table>
<thead>
<tr>
<th>Problem I</th>
<th>Generated code is too large.</th>
</tr>
</thead>
<tbody>
<tr>
<td>Cause</td>
<td><strong>Control flow statements partitioning strategy</strong> exploits <strong>code replication</strong> but not enough <strong>communication</strong>.</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Problem II</th>
<th>Slow execution time (no parallelism)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Cause</td>
<td><strong>Data &amp; computations partitioning strategy</strong> does <strong>not exploit code replication</strong>.</td>
</tr>
</tbody>
</table>
Problems

<table>
<thead>
<tr>
<th>Problem I</th>
<th>Generated code is too large.</th>
</tr>
</thead>
<tbody>
<tr>
<td>Cause</td>
<td>Control flow statements partitioning strategy exploits code replication but not enough communication.</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Problem II</th>
<th>Slow execution time (no parallelism)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Cause</td>
<td>Data &amp; computations partitioning strategy does not exploit code replication.</td>
</tr>
</tbody>
</table>
Control Flow Partitioning

SPMD strategy (original)

// source
int@2 f(int@2 i)
{ ... }

int@1 x;
for(i from 0 to 100)
    x += f(i);

Actor strategy
// annotate with
actor f;

More code
Less communication

Replicates relevant control flow constructs onto every partition.

“Requestor partition”  “Actor partition”

Sends a request to execute code to avoid control flow duplication.

Less code
More communication
Given a program with a complete partitioning assignment, how to generate code for each partition?

- Which control flow constructs need to be replicated in each partition?
- Which partition is a “requestor” of a function?
- Which partitions are “actors” of a function?
Compiling with Mixed Strategy

```c
fix1_t@21 f[8];     fix1_t@11 s[8];
fix1 t@23 b1[32];   fix1 t@13 b2[32];

actor get_b;
fix1 t@23 get_b(int@23 index) {
  if (index <@23 32)
    return b1[index] ;
  else
    return b2[index -@13 32];
}

actor step;
void step(int@22 g) {
  for (i from 0 to 8)
    f[i] = s[i] *@22 get_b((g <<@22 3) +@22 i);
}

void swap(int@21 n) {
  for (i from 0 to 8) s[i] = f[i] << @21 n;
}

void main() {
  while(1) {
    int@32 g = ...; int@32 shift = ...;
    step(g);
    swap(shift);
  }
}
```
Compiling with Mixed Strategy

```c
fix1_t@21 f[8];     fix1_t@11 s[8];
fix1 t@23 b1[32];   fix1 t@13 b2[32];

actor get_b;
fix1 t@23 get_b(int@23 index) {
    if (index <@23 32)
        return b1[index];
    else
        return b2[index -@13 32];
}

actor step;
void step(int@22 g) {
    for (i from 0 to 8)
        f[i] = s[i] *@22 get_b((g <<@22 3) +@22 i);
}

void swap(int@21 n) {
    for (i from 0 to 8) s[i] = f[i] << @21 n;
}

void main () {
    while(1) {
        int@32 g = ...; int@32 shift = ...
        step(g);
        swap(shift);
    }
}
```
Partition 23 is dominated by actor function get_b.

(Partition $p$ is dominated by function $f$, if all paths from main to $p$ pass $f$.)
Therefore,
• Partition 23 is an actor of `get_b`.
• Control flow of 23 starts at `get_b`.

Partition 23 is **dominated** by actor function `get_b`.

(Partition `p` is **dominated** by function `f`, if all paths from main to `p` pass `f`.)
Partition 21 is not dominated by actor function step.

Therefore,
- Partition 21 is not an actor of step.
- Control flow of 21 starts at main.
SPMD + Actor Strategy

actor call

actor partitions for \texttt{get\_b}

actor partitions for \texttt{step}
Problems

Problem I  Generated code is too large.
Cause     Control flow statements partitioning strategy exploits code replication but not enough communication.

Problem II Slow execution time (no parallelism)
Cause     Data & computations partitioning strategy does not exploit code replication.
fix1_t@1 classify(fix1_t@1 acc[3], fix1_t@2 model[N]) {...}

prob1 = classify(acc, model1);
prob2 = classify(acc, model2);
Work Around: Manual Replication

```latex
\text{fix1\_t@1 \texttt{classify1}(fix1\_t@1 \texttt{acc}[3], \texttt{fix1\_t@2 \texttt{model}[N]) \{\ldots\}}
\text{fix1\_t@5 \texttt{classify2}(fix1\_t@5 \texttt{acc}[3], \texttt{fix1\_t@6 \texttt{model}[N]) \{\ldots\}}

\text{prob1 = \texttt{classify1}(\texttt{acc}, \texttt{model1});}
\text{prob2 = \texttt{classify2}(\texttt{acc}, \texttt{model2});}
```
Solution: Automatic Replication

// Define module
module Classifier(model_init) {
    fix1_t@1 model[N] = model_init;
    fix1_t@2 classify(fit1_t@2 acc[3]) {
        ...
    }
}

// Create module instances
C1 = new Classifier(model1);
C2 = new Classifier(model2);

// Call two different functions
C1.classify(acc);
C2.classify(acc);

// Expanded from module instance 1
fix1_t@1 C1_model[N] = model1;
fix1_t@2 C1_classify(fit1_t@2 acc[3]) {
    ...
}

// Expanded from module instance 2.
fix1_t@3 C2_model[N] = model2;
fix1_t@4 C2_classify(fit1_t@4 acc[3]) {
    ...
}

C1_classify(acc);
C2_classify(acc);
Balance the use of **communication** and **code replication** to partition

- program control flow statements
- data and computation
Hand Gesture Recognition

1. I2C driver
2. Filter
3. Classifier for circle
4. Classifier for flip-roll

Process:
- I2C driver sends data to the Filter.
- Filter processes the data and sends it to the classifiers.
- Classifiers output probabilities.

Data Flow:
- 'acc'
- 'active acc'

Final Output:
- Probability for circle
- Probability for flip-roll
Hand Gesture Recognition

- I2C driver
  - acc
  - active acc

Filter

Classifier for circle

Quantizer
  - quantized group

HMM Classifier
  - Model data
  - probability

Classifier for flip-roll
  - probability
1. Use **mixed partitioning strategy** to make the application fit on GA144. *orange = actor cores*
2. Use **parallel module** to classify circle and flip-roll gestures in parallel.
Can we use Chlorophyll with our extensions to generate code for the gesture recognition application for GA144?

<table>
<thead>
<tr>
<th>Partitioning strategy</th>
<th>Number of cores</th>
<th>Overflowed cores</th>
<th>Size of largest core (words)</th>
</tr>
</thead>
<tbody>
<tr>
<td>SPMD</td>
<td>90</td>
<td>12</td>
<td>87</td>
</tr>
<tr>
<td>SPMD + Actor</td>
<td>82</td>
<td>0</td>
<td>64</td>
</tr>
</tbody>
</table>

Note: each core can contain up to 64 words.

Code occupies 82 out of 144 cores.
Prediction accuracy = 80-91% (similar to Wigee [Schlomer et al. 08])
GA144 vs. MSP430

How much energy consumption can we reduce by being able to compile for GA144?

*per one round of accelerometer reading

<table>
<thead>
<tr>
<th>Processor</th>
<th>Execution time (ms)</th>
<th>Energy consumption (μJ)</th>
<th>Accelerometer</th>
<th>Computation</th>
<th>Total</th>
</tr>
</thead>
<tbody>
<tr>
<td>GA144</td>
<td>2.6</td>
<td></td>
<td>1.7</td>
<td>0.6</td>
<td>2.2</td>
</tr>
<tr>
<td>MSP430</td>
<td>61.3</td>
<td></td>
<td>0.8</td>
<td>41.2</td>
<td>41.9</td>
</tr>
</tbody>
</table>

23x faster

19x more energy-efficient
Demo
Summary

Partition a program smartly to fit the code in tiny cores and achieve fast execution time by balancing communication and code replication.
0.5 mW!