Lecture 14: Mapping to parallel hardware

Embedded Software
Michael O’Boyle
Overview

• Mapping tasks to parallel hardware
  – crux issue but no agreed approach
• Developing (Discovering) parallel tasks
  – very difficult but infrequent
• Mapping
  – can be (semi-) automated
  – repeated for every platform
• Look at an explicit parallel task language StreamIT
  – consider different mapping approaches
Mapping problem

- Sequential program
- Parallel program
  - machine unaware
- Partition into parallel tasks
- Allocate parallel tasks to processors
  - machine aware
- Schedule tasks
- If assume one task per processor
  - then partitioning is mapping
The Streaming Domain

- Widely applicable and increasingly prevalent
  - Embedded systems
    - Cell phones, handheld computers, DSP’s
  - Desktop applications
    - Streaming media
    - Software radio
    - Real-time encryption
    - Graphics packages
  - High-performance servers
    - Software routers (Example: Click)
    - Cell phone base stations
    - HDTV editing consoles
- Based on audio, video, or data stream
- StreamIT from MIT
  - a language that supports this domain
StreamIT Programs

- Operate on a large sequence of data items (data stream) in pipeline fashion.

- Each data item is processed for a limited time before being discarded.

- Each program has a stream graph

- Explicit parallel tasks

- Processing unit (or functions)

- Data flow
Structured Streams

• Hierarchical structures:
  – Pipeline
  – SplitJoin
  – Feedback Loop

• Basic programmable unit: Filter
float->float filter LowPassFilter(int N) {
    float[N] weights;

    init {
        for (int i=0; i<N; i++)
            weights[i] = calcWeights(i);
    }

    work push 1 pop 1 peek N {
        float result = 0;
        for (int i=0; i<N; i++)
            result += weights[i] * peek(i);
        push(result);
        pop();
    }
}

StreamIT Filter Example:

Thursday, 6 March 2014
Filter Example: LowPassFilter

float->float filter LowPassFilter(int N) {
    float[N] weights;

    init {
        for (int i=0; i<N; i++)
            weights[i] = calcWeights(i);
    }

    work push 1 pop 1 peek N {
        float result = 0;
        for (int i=0; i<N; i++)
            result += weights[i] * peek(i);
        push(result);
        pop();
    }
}
float->float filter LowPassFilter(int N) {
    float[N] weights;

    init {
        for (int i=0; i<N; i++)
            weights[i] = calcWeights(i);
    }

    work push 1 pop 1 peek N {
        float result = 0;
        for (int i=0; i<N; i++)
            result += weights[i] * peek(i);
        push(result);
        pop();
    }
}
Filter Example: LowPassFilter

```c
float->float filter LowPassFilter(int N) {
    float[N] weights;

    init {
        for (int i=0; i<N; i++)
            weights[i] = calcWeights(i);
    }

    work push 1 pop 1 peek N {
        float result = 0;
        for (int i=0; i<N; i++)
            result += weights[i] * peek(i);
        push(result);
        pop();
    }
}
```
float->float filter LowPassFilter(int N) {
    float[N] weights;

    init {
        for (int i=0; i<N; i++)
            weights[i] = calcWeights(i);
    }

    work push 1 pop 1 peek N {
        float result = 0;
        for (int i=0; i<N; i++)
            result += weights[i] * peek(i);
        push(result);
        pop();
    }
}
complex->void pipeline BeamFormer(int numChannels, int numBeams)
{
    add splitjoin {
        split duplicate;
        for (int i=0; i<numChannels; i++) {
            add pipeline {
                add FIR1(N1);
                add FIR2(N2);
            };
        }
        join roundrobin;
    };
    add splitjoin {
        split duplicate;
        for (int i=0; i<numBeams; i++) {
            add pipeline {
                add VectorMult();
                add FIR3(N3);
                add Magnitude();
                add Detect();
            }
        }
        join roundrobin;
    };
}
Partitioning the Program

- Map the input program graph to threads
- Need to find a good one from many possible partitions

3 possible partitions on a 2-core machine
Partitioning

• Mapping filters
  – Each filter should have similar amount of work
    • Throughput determined by the filter with most work

• Original StreamIT Compiler Algorithm
  – Two primary transformations
    • Filter fission
    • Filter fusion
  – Uses a greedy heuristic
Partitioning - Fission

- Fission - splitting streams
  - Duplicate a filter, placing the duplicates in a SplitJoin to expose parallelism.

-Split a filter into a pipeline for load balancing
Partitioning - Fusion

• Fusion - merging streams
  – Merge filters into one filter for load balancing and synchronization removal

![Diagram showing fusion of filters]

Thursday, 6 March 2014
Example: Radar Array Front End
Example: Radar Array Front End
Example: Radar Array Front End
Example: Radar Array Front End
Example: Radar Array Front End
Example: Radar Array Front End
Example: Radar Array Front End (Balanced)
Performance?

- The best partitioning strategy varies across programs and platforms

(a) Intel Xeon 4-core (2x dual-cores)  (b) Intel Xeon 8-core (2x quad-cores)
Using Machine Learning

• Problem
  – Tuning heuristics is hard
  – Architectures and system software keep changing

• Goal
  – Replace an heuristic with a machine learning one

• Machine learning (ML) performs very well (Zheng Wang 2010)
How to model partitioning for ML

- Use a sequence of merging and splitting operations to generate a partition

Compact graph representation.
A Two Step Approach

1. Predict characteristics of the ideal partition
2. Search for a partition with those characteristics

Do NOT run any of the generated partition for searching
Characterise the Ideal Partition

• Use static compiler information to characterise the ideal partitioning structure

<table>
<thead>
<tr>
<th>Information from compiler</th>
<th>Information from compiler</th>
</tr>
</thead>
<tbody>
<tr>
<td>#Processing units</td>
<td>#communication channels</td>
</tr>
<tr>
<td>Pipeline depth</td>
<td>Data parallel section width</td>
</tr>
<tr>
<td>Load balance</td>
<td>Computation-communication ratio</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
</tr>
</tbody>
</table>

Thursday, 6 March 2014
Predictive Modelling

• Use a machine learning model to predict characteristics of the ideal structure
Build a Machine Learning

1. Determining the feature values of many training programs
2. Try different partitions and find the best for each
3. Give the training examples to a machine learner to learn a model
Use the Model: Step 1

- Nearest neighbour algorithm chooses a training program that is the most similar to the input program.
Use the Model: Step 2

- Select a randomly generated partition whose structure is the most close to the predicted one.

* We do not run the program!

Distance is measured by calculating the Euclidean distance of 2 characteristic vectors.
Experimental Setup

• Platforms
  – 2 x Dual Intel Xeon 5160 processors
    (4 cores in total)
  – 2 x Quad Intel Xeon 5450 processors
    (8 cores in total)

• Compilers:
  – StreamIt version 2.1.1

• Benchmarks:
  ▪ 17 StreamIt applications

• Comparison:
  ▪ 2 StreamIt compiler built-in partitioners
  ▪ An analytical-based model (Navarro et al., PACT 2009)
  ▪ Baseline: StreamIt dynamic programming-based partitioner
1.9x Improvement over State-of-Art (4-core)
Compare with the Best-found Performance (4-Core)

![Graph showing speedup comparison between Machine Learning and Best-Found for various applications like Radixsort, Filterbank, TDE, SAR, DES, SERPENT, CH-Vocoder, Matmul, MP3Decoder, Insert Sort, FFT, DCT, Beamformer, FM, MPEG2, Vocoder, Lattice, and Average. The graph indicates speedup values ranging from 0.5 to 4.4 for Machine Learning and from 0.5 to 6.981 for Best-Found.]
1.8x Improvement over State-of-Art (8-Core)

*Baseline: StreamIt compiler default strategy

**Chart Details:**
- **Y-Axis:** Speedup
- **X-Axis:** Applications
- **Legend:**
  - Yellow: Analytic
  - Pink: Greedy
  - Black: Machine Learning

**Applications:**
- RADIXSORT
- FILTERBANK
- TDE
- SAR
- DES
- SERPENT
- CH:VOCoder
- MATMUL
- MP3DECODER
- INSERT:SORT
- FFT
- DCT
- BEAMFORMER
- FM
- MPEG2
- VOCODER
- LATTICE
- AVERAGE

**Comparative Performance:**
- Analytic: 4.5, 4.3, 3.5, 4.9, 4.3
- Greedy: 0.5, 0.5, 0.5, 0.5, 0.5
- Machine Learning: 0.5, 0.5, 0.5, 0.5, 0.5

**Note:**
- The chart illustrates the performance improvement of various applications over the baseline strategy.
Comparison to ‘Oracle’ (Best) Performance

- Similar performance on the two platforms
- ML significantly outperforms others
Conclusion

• Mapping is a crux issue
• Depends on how parallelism and tasks represented
• Looked at StreamIT
• Smart heuristics vulnerable to change
  – ml can help
• Wide open research area
• Large research interest at ICSA Edinburgh