Automated Space/Time Scaling of Streaming Task Graphs

Hossein Omidian
Supervisor: Guy Lemieux
Contents

- Introduction
- KPN-based HLS Tool for MPPA overlay
- Experimental Results
- Future Work
- Conclusion
Introduction

- Clock frequency scaling has essentially stopped due to power issue
  - Coarse-grained parallelism is achieved in multi-core processor
    - High-level Languages
  - Fine-grained parallelism is achieved in FPGA and ASIC
    - Hardware-level solutions
- Both can be challenging
- In this work we wish to explore the design space in between traditional multi-core CPU and low-level FPGA/ASIC solutions
Motivation/Objectives

- We will investigate whether an array of ALUs or very lightweight processors (MPPA overlay), can achieve sufficient levels of performance, and make design entry sufficiently easy.

- To explore the MPPA as an alternative target, we need three things:
  - A tool flow that can compile algorithms into the target
  - A detailed architecture description or implementation, and
  - A set of benchmarks to compile into the framework.
Programming Model/KPN model

- Ambric Programming model
- Java-based
- Java object is created for each thread, becoming nodes that communicate together through FIFO communication channels
- Similar to Kahn Processing Network
KPN-based HLS for MPPA overlay
KPN-based HLS for MPPA overlay

```java
class C_PE1 {
    public static void main(InputStream in1, InputStream in2, OutputStream out) {
        out = writeTo(in1.readFrom() & in2.readFrom());
    }
}

PE PE1 (outbound out, inbound in1, inbound in2) {
     implementation "PE1_imp.java";
}

PE PE2 (outbound out, inbound in) {
     implementation "PE2_imp.java";
}

binding b1 {
    channel C0 = {PE1.out, PE2.in};
}

class C_PE2 {
    public static void main(InputStream in, OutputStream out) {
        out = writeTo(in.readFrom() * 2);
    }
}
```

A = {f_1, f_2, ..., f_N}
KPN-based HLS for MPPA overlay

1. First find the best throughput
   - Expanding
   - Combining
   - Pipelining
KPN-based HLS for MPPA overlay

- Finding different implementation
  - Clustering
  - Send operations back and forth between clusters
KPN-based HLS for MPPA overlay

\[ A = \{ f_1, f_2, \ldots, f_N \} \]

Program

Java

aStruct

FPGA

MPPA overlay

DB of Implementations for each node

\[ p_1^1, p_1^2, \ldots, p_1^{s_{m_1}} \]

\[ p_2^1, p_2^2, \ldots, p_2^{s_{m_2}} \]

\[ \ldots \]

\[ p_N^1, p_N^2, \ldots, p_N^{s_{m_N}} \]

Stream Task Graph

Compiler

Intra-Node Optimizer

Inter-Node Optimizer

Trade-off Finder

MPPA Overlay Generator

Back-end

Architecture library
Inter-Node Optimizer (INO)

- Implementations $P_m$
  
  $$P_m = \{P_m^1, P_m^2, ..., P_m^{S_m}\}$$

- Each implementation $P_m^s$
  
  - Can perform functionality of $f_m$
  - Area cost $A(P_m^s)$ and Initiation Interval $II(P_m^s)$
  - Different inverse throughput for each channel
    - Input inverse throughput $v_{in}(P_m^s) = \frac{II(P_m^s)}{In(f_m)}$
    - Output inverse throughput $v_{out}(P_m^s) = \frac{II(P_m^s)}{Out(f_m)}$
N-Body Problem

- Gravity Force Calculation

\[
\vec{F}_{i,j} = G \frac{M_i M_j}{r^2} = 0.0625 \frac{M_i M_j}{|\vec{P}_i - \vec{P}_j|^3} (\vec{P}_i - \vec{P}_j),
\]
N-Body Problem

- Expanding
N-Body Problem

- Different implementations

[Chart showing the relationship between reverse throughput and area, with points marked at specific data points (e.g., (0, 33), (5, 1)) and area labeled on the x-axis ranging from 0 to 35.]
Trade-off Finder

- **Two modes**
  - Given available area and different implementations for each node, maximizing throughput
  - Given throughput target and different implementations for each node, minimizing area

- **Two approaches**
  - Integer Linear Programming (GNU linear programming kit)
  - Our heuristic approach
Integer Linear Programming approach

- For each composite node
  - Which Implementation should be selected
  - How many replicas are needed
- Disadvantages:
  - Lack of flexibility
  - Time inefficient
Our heuristic Trade-off Finder

- Throughput Analysis
- Application Throughput Propagation and Balancing
- Bottleneck Optimizer
Throughput Analysis

- Finding throughput slack
- To achieve minimum inverse throughput $v_{mo}$, the input data have to be ready with expected inverse throughput $v_{ei}$. We define slack $v_s$ for each channel as

$$v_s = v_{mo} - v_{ei}$$
Throughput Analysis

- Calculating weight for each node

\[ W_m = \frac{\sum_{j=1}^{N_{out}} t_{sj} - \sum_{i=1}^{N_{in}} t_{si}}{N_{in} + N_{out}} \]
Throughput Propagation and Balancing

\[ \nu_{\text{out}}^k (f_m) = \frac{\min_j \{v_{\text{in}}^j (f_m) \cdot \text{In}^j (f_m)\}}{\text{Out}^k (f_m)} \]  (7)

J. Cong et al., "Combining module selection and replication for throughput-driven streaming programs"
Bottleneck Optimizer

- Similar to ILP, it selects an implementation and makes replicas
- Unlike ILP it considers neighbouring nodes
- Saving area
Bottleneck Optimizer

$$H = \lceil \log_{nf} nr \rceil$$

For $nf=4$ we will save 75% of area overhead
Heuristic Approach

Starting with implementation with the highest throughput for each node → Throughput Analysis in order to find bottlenecks → Calculate Area cost → Budgeting → Bottleneck Optimizer → Passed the area limit

Budget the bottleneck → Application Throughput Propagation and Balancing → Calculate estimated Area cost for nodes → Passed the area limit
Results

• JPEG

```
<table>
<thead>
<tr>
<th>Module</th>
<th>Color Conversion</th>
<th>DCT</th>
<th>Quantization</th>
<th>Encoding</th>
</tr>
</thead>
<tbody>
<tr>
<td>Version</td>
<td>v1 v2 v3 v4</td>
<td>v1 v2 v3 v4 v5</td>
<td>v1 v2 v3 v4 v5</td>
<td>v1 v2 v3 v4 v5 v1</td>
</tr>
<tr>
<td>Inverse T</td>
<td>1 2 4 8</td>
<td>1 2 4 6 32</td>
<td>1 2 4 8 128 512</td>
<td></td>
</tr>
<tr>
<td>Area</td>
<td>512 256 128 64</td>
<td>800 400 224 160 50</td>
<td>512 256 128 64 4</td>
<td>22</td>
</tr>
</tbody>
</table>
```

```
<table>
<thead>
<tr>
<th>Method</th>
<th>T</th>
<th>Color Conversion</th>
<th>DCT</th>
<th>Quantization</th>
<th>Encoding</th>
<th>Overhead</th>
<th>Total Area</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>impl rep</td>
<td>impl rep</td>
<td>impl rep</td>
<td>impl rep</td>
<td></td>
<td></td>
</tr>
<tr>
<td>ILP</td>
<td>8</td>
<td>v4 1</td>
<td>v4 1</td>
<td>v4 1</td>
<td>v1 32</td>
<td>1280</td>
<td>2976</td>
</tr>
<tr>
<td>Heuristic</td>
<td></td>
<td>v4 1</td>
<td>v5 4</td>
<td>v5 16</td>
<td>v1 32</td>
<td>0</td>
<td>1736</td>
</tr>
</tbody>
</table>
```
Future Work

- MPPA Overlay Generator
- Time and Resource sharing
**MPPA Overlay Generator**

- 1 TOPS -> 32bit integer overlay and
- 800GFLOPS -> 32bit single-precision floating point overlay on the largest Stratix 10 device
Time and Resource Sharing

- Using underutilized nodes
- Stalling most of the times \[ \frac{nf^i - 1}{nf^i} \]
Conclusion

- Studied the problem of automatically finding area/throughput trade-off of streaming applications being mapped onto MPPA overlays
- We introduce a high-level synthesis tool that compiles an stream application written in Java as a streaming task graph
- Partitions it into composite nodes
- Finds all degrees of parallelism for each
- Uses different approaches in order to find different implementations for each node
- And finally finds a good trade off between area and throughput.
- Our approach differs from existing approaches
  1. it automatically investigates partitioning and finding different implementations, and
  2. it combines module selection and replication methods with node combining and splitting in order to automatically find a better area/throughput trade-off
- This approach has been verified with small designs in StreamIt and a few larger designs like the JPEG encoder
Questions?
Results

- JPEG

```
<table>
<thead>
<tr>
<th>Module</th>
<th>Color Conversion</th>
<th>DCT</th>
<th>Quantization</th>
<th>Encoding</th>
</tr>
</thead>
<tbody>
<tr>
<td>Version</td>
<td>v1 v2 v3 v4</td>
<td>v1 v2 v3 v4</td>
<td>v1 v2 v3 v4 v5</td>
<td>v1 v2 v3 v4 v5 v1</td>
</tr>
<tr>
<td>T</td>
<td>1 2 4 8</td>
<td>1 2 4 6 32</td>
<td>1 2 4 8 128 512</td>
<td></td>
</tr>
<tr>
<td>Area</td>
<td>512 256 128 64</td>
<td>800 400 224 160 50</td>
<td>512 256 128 64 4 22</td>
<td></td>
</tr>
</tbody>
</table>
```

Table 2.1. Implementation library for JPEG modules

```
<table>
<thead>
<tr>
<th>Method</th>
<th>T</th>
<th>Color Conversion</th>
<th>DCT</th>
<th>Quantization</th>
<th>Encoding</th>
<th>Overhead</th>
<th>Total Area</th>
</tr>
</thead>
<tbody>
<tr>
<td>ILP</td>
<td>1</td>
<td>v1 1</td>
<td>v1 1</td>
<td>v1 1</td>
<td>v1 512</td>
<td>10880</td>
<td>23968</td>
</tr>
<tr>
<td>Heuristic</td>
<td></td>
<td>v1 v5 32</td>
<td>v5 64</td>
<td>v1 128 512</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>ILP</td>
<td>2</td>
<td>v2 1</td>
<td>v2 1</td>
<td>v2 1</td>
<td>v1 128</td>
<td>5376</td>
<td>11920</td>
</tr>
<tr>
<td>Heuristic</td>
<td></td>
<td>v2 v5 16</td>
<td>v5 64</td>
<td>v1 128 256</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>ILP</td>
<td>4</td>
<td>v3 1</td>
<td>v3 1</td>
<td>v3 1</td>
<td>v1 64</td>
<td>2688</td>
<td>5984</td>
</tr>
<tr>
<td>Heuristic</td>
<td></td>
<td>v3 v5 8</td>
<td>v5 32</td>
<td>v1 128 64</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>ILP</td>
<td>8</td>
<td>v4 1</td>
<td>v4 1</td>
<td>v4 1</td>
<td>v1 32</td>
<td>1280</td>
<td>2976</td>
</tr>
<tr>
<td>Heuristic</td>
<td></td>
<td>v4 v5 4</td>
<td>v5 16</td>
<td>v1 32 0</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
```

Table 2.2. ILP vs. Heuristic method results