Array Scalarization In High Level Synthesis

Preeti Ranjan Panda¹, Namita Sharma¹, Arun Kumar Pilania², G. Krishnaiah², Sreenivas Subramoney² and Ashok Jagannathan²

¹Indian Institute of Technology Delhi
²Intel Technology India Pvt. Ltd.
What is Array Scalarization?

- Arrays are usually mapped to memories (SRAM) in High Level Synthesis
  - Performance bottlenecks due to memory port constraints.
- Scalarization
  - Transforming an array into a group of scalar variables.
  - Entire array is stored in discrete registers.

**Question:** How to automatically decide when to Scalarize?
Previous Work

- Commercial tools (Cadence, Synopsys) allow user pragmas for controlling scalarization
- In this paper: *Automatic strategy for Scalarization*
for (i = 0; i < 4; i++)

Latency = 4 x 3 = 12 cycles
Loop Unrolling

Loop Unrolling

for (i = 0; i < 4; i++)

Unroll
(Resource: 2-port Mem)

Latency = 4 cycles

P. R. Panda et al., ASPDAC 2014
Scalarization

Unroll + Scalarization

Latency = 12 cycles

for (i = 0; i < 4; i++)

LD

ST

A

x

A

x

A

x

A

x

RA_0

RA_1

RA_2

RA_3

Cycle 1

RA_0 = RA_0 + x;
RA_1 = RA_1 + x;
RA_2 = RA_2 + x;
RA_3 = RA_3 + x;

92% reduction over rolled loop

75% reduction over unrolled loop

Latency = 1cycle

P. R. Panda et al., ASPDAC 2014
What makes ARRAY SCALARIZATION Difficult?

P. R. Panda et al., ASPDAC 2014
Challenges

- **Area Overheads**
  - Discrete Registers => Extra area
  - $a, b$ have same area
  - Scalarizing $b$ results in more latency reduction

- **Sparse Array Accesses**
  - large area penalty
  - not much performance gain

```c
int MetricMap [128];
... 
curr = MetricMap [3];
for (i = 0; i < 128; i++)
a[i] = a[i] * p;
for (j =0; j < 128; j++)
b[j] = b[j] << s;
for (k = 0; k < 128; k++)
b[k] = b[k] + v;
```
Challenges

- Loop Carried Dependencies and Resource Constraints
  - Dependence => Sequentialization
  - Not much benefit from Scalarization

- Data dependent array indexing
  - Scalarizing \textbf{a} useful only if \textbf{b} is also scalarized
  - Scalarizing \textbf{b} is expensive
    - data dependent access => large 100-to-1 MUX connecting to all registers

\begin{verbatim}
for (i = 0; i < 50; i++)
\end{verbatim}

\begin{verbatim}
for (i = 0; i < 150; i++)
    index = calc (i) % 100;
    val = b[index];
    a [i] = a[i] << val;
\end{verbatim}

P. R. Panda et al., ASPDAC 2014
Problem Definition

Given a behavioral description with loops $L_1, L_2, ..., L_n$ accessing arrays $A_1, ..., A_m$, with
- each loop $L_j$ accessing a subset $S_j$ of the arrays
- array size for $A_i$ is $\text{Size}(i)$
- an allowed area overhead constraint $OV$

compute the **Scalarization Vector** $SV [1...m]$

$SV [i] = \text{TRUE}$ means $A_i$ is scalarized and $= \text{FALSE}$ means $A_i$ is not scalarized

such that the overall latency (cycle count) reduction is maximized while the total area overhead $< OV$. 

P. R. Panda et al., ASPDAC 2014
APPROACH
L1: for(i = 0; i<50; i++)
    A[i] = B[i] + 3;

L2: for(i = 0; i<50; i++)
    C[i] = D[i] + B[i];

L3: for(i = 0; i<50; i++)
    D[i] = A[i] * 4;

Hyperedge = Loop
Node = Array

P. R. Panda et al., ASPDAC 2014
Cost Estimation

- Cost of scalarizing an array comprises of area overheads due to
  - Storing the array elements in discrete registers instead of SRAM cells

  \[ Storage\_Area(i) = Size(i) \times \text{ElementSize}(i) \times (RArea - SArea) \]

- MUX costs due to scalarization
  - Due to resource sharing by the scalarized registers = \( Size(i)/R \)
  - Data dependent accesses = \( Size(i) \)

  \[ \text{Cost}(A_i) = Storage\_Area(i) + MUX\_Area(i) \]

  \[ \text{Cost}(L_i) = \sum \text{Cost}(A_j), \ A_j \in S_i \]
Latency Estimation

- Compute the **Initiation Interval (II)** for the loop which is a function of
  - Resource constraints
  - Loop carried dependencies

\[
II = \max \left( \max \left[ \frac{ops(i) \cdot lat(i)}{FU(i)} \right] \right) \forall i, \max \left[ \frac{dep(t)}{it(t)} \right] \forall t
\]

\[
Lat(i) = Body(i) + II \cdot (iter(i) - 1)
\]


P. R. Panda et al., ASPDAC 2014
Priority Function

\[
\text{Priority}(L_i) = \frac{\text{Red}(L_i)}{\text{Cost}(L_i)}
\]

\[
\text{Red}(L_i) = \text{Body}(i) \times \text{iter}(i) - \text{Lat}(i)
\]

\[
\text{Cost}(L_i) = \sum \text{Cost}(A_j), \quad A_j \in S_i
\]
Build the **array access graph**

∀ Edges

Compute **Cost** \( (L_i) \), **Red** \( (L_i) \), **Priority** \( (L_i) \)

**Update** Candidate List **CL**

\[ E_i = \text{Candidate List } CL \rightarrow \text{head} \]

\[ \text{OV} > \text{Cost} (L_i) \]

**SV\[j\] = TRUE**

**Update** Cost \( (L_j) \), **Priority** \( (L_j) \) ∀ \( j \)

Candidate List **CL & OV**

\[ CL! = \phi \land \text{OV} > 0 \]

\[ CL! = \phi \land \text{OV} > 0 \]

**End**

P. R. Panda et al., ASPDAC 2014
Experimental Results

Latency reduction with simple Unrolling

P. R. Panda et al., ASPDAC 2014
Experimental Results

Latency reduction with Unrolling & Scalarization

P. R. Panda et al., ASPDAC 2014
Experimental Results

P. R. Panda et al., ASPDAC 2014
Conclusions

- Investigated *Array Scalarization*
  - a behavioral synthesis optimization
  - selectively transforms arrays accessed in loops to scalars
    - performance improvement under area overhead constraint.

- Automated strategy
  - prioritize the loops based on:
    - expected performance benefits
    - resulting area overhead
Thank You!