COMBA: A Comprehensive Model-Based Analysis Framework for High Level Synthesis of Real Applications

Jieru Zhao  Hong Kong Univ. of Science and Technology
Liang Feng  Hong Kong Univ. of Science and Technology
Sharad Sinha  Nanyang Technological Univ., Singapore
Wei Zhang  Hong Kong Univ. of Science and Technology
Yun Liang  Peking Univ., China
Bingsheng He  National Univ. of Singapore
Why FPGA?

- **High** performance
- **Low** power
- Post-fabrication **reconfigurability**
- **Short** time-to-market
FPGA Applications

Data Center

Medical

Wireless Communications

Broadcast

Automotive

Wired Communications

Aerospace and Defense
Programmability of FPGA is further improved!
Pragmas in HLS tool

Selection of directives/pragmas Determines The quality of RTL designs

How to choose the most suitable pragmas quickly?
Motivational Example

More than **millions** of combinations.

The performance difference can be very **large**!

**Important:**
To choose the **best** configuration.

**Difficult:**
To **quickly** find the best one.

For this application, COMBA spends 10 min to find the high-performance configuration, in a design space with $7.61 \times 10^{12}$ points.
Contributions

State-of-the-art[1]

- **Three** pragmas
  1) Loop unrolling
  2) Loop pipelining
  3) Array partitioning

COMBA

- **Seven** pragmas
  1) Loop unrolling
  2) Loop pipelining
  3) Array partitioning
  4) Function pipelining
  5) Dataflow
  6) Loop flattening
  7) Function inlining

Contributions

State-of-the-art[1]

- **Performance** models for **simple** code structures
- **No resource** models
- Analysis on **execution trace**
- **Small** design space: $\sim 10^2$ points
- Design space exploration: **Brute force** method

COMBA

- **Performance** models for more **complex** code structures
- **Resource** models: DSP and BRAM
- Analysis on **source code**
- **Enlarged** design space: $\sim 10^8$ points
- Design space exploration: **Efficient** Metric-Guided DSE algorithm

---

Framework

1. RDC
2. Models
3. MGDSE

RDC: Recursive data collector. Computes the parameters required by our model.

Model:
- Performance Model
- Resource Model
- Guided Search

MGDSE: an algorithm for design space exploration. Evaluates the estimated results. Sets the next configuration for RDC.

COMBA iterates until it finds the high-performance configuration.
RDC: Recursive data collector. Computes the parameters required by our model.
Overview

RDC:
- Recursive data collector.
  - Computes the parameters required by our model.

Model:
- Estimates the performance and resource usage.
  - Performance Model
  - Resource Model

MGDSE:
- Guided Search
- Redundancy Elimination
- Optimal Configuration

C/C++ Source Code

Clang Front End

DFG Construction

Operation Chaining

Library

Pragmas

Frequency
Overview

RDC: Recursive data collector. Computes the parameters required by our model.

Model:
- Performance Model
- Resource Model

MGDSE: an algorithm for design space exploration
- Evaluates the estimated results
- Sets the next configuration for RDC.

MGDSE
- Redundancy Elimination
- Guided Search

Optimal Configuration
Overview

RDC:
- Recursive data collector. Computes the parameters required by our model.

Model:
- Estimates the performance and resource usage.
- Evaluation of estimated results.
- Sets the next configuration for RDC.

MGDSE:
- Algorithm for design space exploration.
- Evaluates the estimated results.
- Sets the next configuration for COMBA.

COMBA iterates until it finds the high-performance configuration.

Guided Search

Optimal Configuration

Redundancy Elimination

Resource Model

Performance Model

Operation Chaining

DFG Construction

Pragmas

Library

Frequency

Clang Front End

C/C++ Source Code
Data Collection

Recursive Data Collection

1. Library
2. Pragmas
3. Parameters
4. Frequency
5. LLVM IR
6. DFG Construction
7. Operation Chaining
8. Framework
9. RDC
10. Models
11. MGDSE

More than one operation can be scheduled in one cycle if possible.

A dynamic programming approach is employed to trace each path and compute the latency in the critical path.
Recursive Data Collection

- RDC: an optimization pass based on `llvm::Module` class.
- LLVM IR: LLVM intermediate representation
Recursive Data Collection

Data flow graph (DFG) construction

```c
void function(...) {
    int a = t * m;
    int c = a + b;
    sub_func1(a, *p);
    sub_func2(c, *q);
    loop1{
        p[i] ...
    }
    loop2{
        q[i] ...
    }
}
```

Diagram:
- **Logic operations**
- **Invoked functions**
- **Loops**
Recursive Data Collection

Operation chaining

More than one operation can be scheduled in one cycle if possible.

A dynamic programming approach is employed to trace each path and compute the latency in the critical path.
Models: Performance Model

Framework

1. RDC

2. Models

3. MGDSE

Equations:

Iteration latency:
\[ C_{U_k}^{L_k} = C_{U_{k+1}}^{L_{k+1}} \cdot \frac{B_{k+1}}{U_{k+1}} \cdot U_k + C_{U_k}^{L_k \setminus L_{k+1}} \]

Loop latency:
\[ Cycle_{L_k} = C_{L_k}^{U_k} \cdot \frac{B_k}{U_k} \]
Analytical Models: Performance Model

1. Loop Unrolling

Iteration latency:

\[ C_{L_k}^{U_k} = C_{L_{k+1}}^{U_{k+1}} \cdot \frac{B_{k+1}}{U_{k+1}} \cdot U_k + C_{L_k \setminus L_{k+1}}^{U_k} \]

Loop latency:

\[ \text{Cycle}_{L_k} = C_{L_k}^{U_k} \cdot \frac{B_k}{U_k} \]
Analytical Models: Performance Model

1. Loop Unrolling

\[ L_k \]
\[
\text{for(int } i = 0; i < 8; i++)
\{
    //pragma HLS UNROLL factor=2
    for(int } j = 0; j < 8; j++)
    {
        //pragma HLS UNROLL factor=2
        a[i] += b[j] * j;
    }
\]
\[ L_{k+1} \]
\[ c[i] *= a[i]; \]

Loop latency of $L_{k+1}$

Iteration latency:

\[ C_{L_k}^{U_k} = C_{L_{k+1}}^{U_{k+1}} \cdot \frac{B_{k+1}}{U_{k+1}} \cdot U_k + C_{L_k \setminus L_{k+1}}^{U_k} \]

Loop latency:

\[ \text{Cycle}_{L_k} = C_{L_k}^{U_k} \cdot \frac{B_k}{U_k} \]
Analytical Models: Performance Model

1. Loop Unrolling

**Iteration latency:**

\[
C_{L_k}^{U_k} = C_{L_{k+1}}^{U_{k+1}} \cdot \frac{B_{k+1}}{U_{k+1}} \cdot U_k + C_{L_k \setminus L_{k+1}}^{U_k}
\]

**Loop latency:**

\[
Cycle_{L_k} = C_{L_k}^{U_k} \cdot \frac{B_k}{U_k}
\]
1. Loop Unrolling

Iteration latency:

\[
C_{U_k}^{L_k} = C_{U_{k+1}}^{L_{k+1}} \cdot \frac{B_{k+1}}{U_{k+1}} \cdot U_k + C_{L_k}^{U_k} \cdot U_{k+1} \]

Loop latency:

\[
Cycle_{L_k} = C_{L_k}^{U_k} \cdot \frac{B_k}{U_k}
\]
2. Loop Pipelining

Latency of one iteration

Pipeline depth

$D$

Initiation interval

Latency between the beginning of two consecutive iterations

Latency of a pipelined loop:

$Cycle_L = D + II \cdot (Tc - 1)$
Analytical Models: Performance Model

2. Loop Pipelining: pipeline depth, \( Cycle_{Li} = D_i + II_i \cdot (Tc - 1) \)

Load & Store operations: access different arrays, \( D_i = \frac{C_{Li}^{Ui}}{II_i} \)

access the same array, \( D_i = \left[ \frac{C_{Li}^{Ui}}{II_i} \right] \cdot II_i \)
Analytical Models: Performance Model

2. Loop Pipelining: pipeline depth, \( C_{Li} = D_i + II_i \cdot (Tc - 1) \)

Load & Store operations:
- access different arrays, \( D_i = C_{Li}^{Ui} \)
- access the same array, \( D_i = \left[ \frac{C_{Li}^{Ui}}{II_i} \right] \cdot II_i \)

Load & Store operations: 
- Access different arrays
- Access the same array
2. Loop Pipelining: pipeline depth, \( Cycle_{Li} = D_i + II_i \cdot (T_c - 1) \)

Load & Store operations: access different arrays, \( D_i = C_{Li}^{U_i} \)

access the same array, \( D_i = \left[ \frac{C_{Li}^{U_i}}{II_i} \right] \cdot II_i \)

\( C_{Li}^{U_i} = 5 \)

\( II_i = 2 \)
Analytical Models: Performance Model

2. Loop Pipelining: pipeline depth, $Cycle_{L_i} = D_i + II_i \cdot (Tc - 1)$

Load & Store operations:
- access different arrays, $D_i = C_{L_i}^{U_i}$
- access the same array, $D_i = \left[ \frac{C_{L_i}^{U_i}}{II_i} \right] \cdot II_i$

Diagram:
- $C_{L_i}^{U_i} = 5$
- $D_i = 6$
- $II_i = 2$
Analytical Models: Performance Model

2. Loop Pipelining: initiation interval, \( Cycle_{Li} = D_i + II_i \cdot (Tc - 1) \)

\[
II_{i,\text{min}} = \max(II_{i,\text{min}}^{\text{res}}, II_{i,\text{min}}^{\text{rec}})
\]

Resource-constrained II

\[
II_{i,\text{min}}^{\text{res}} = \max_m \left( \left\lfloor \frac{Access_m}{Ports_m} \right\rfloor \right)
\]

1. Load Load Add … Store
2. Load Load Add … Store
Analytical Models: Performance Model

2. Loop Pipelining: initiation interval,  \( Cycle_{Li} = D_i + II_i \cdot (Tc - 1) \)

\[
II_{i,\text{min}} = \max (II_{i,\text{res}, \min}, II_{i,\text{rec}, \min})
\]

Resource-constrained II

\[
II_{i,\text{res}, \min} = \max_m \left( \left\lceil \frac{Access_m}{Ports_m} \right\rceil \right)
\]

1. Load Load Add \( \cdots \) Store

2. \( II \)

\[
II_{i,\text{rec}, \min} = \frac{2}{1} = 2
\]
2. Loop Pipelining: initiation interval, \( Cycle_{Li} = D_i + II_i \times (Tc - 1) \)

\[
II_{i,\text{min}} = \max \left( II_{i,\text{min}}^{\text{res}}, II_{i,\text{min}}^{\text{rec}} \right)
\]

Resource-constrained II

\[
II_{i,\text{min}}^{\text{res}} = \max_m \left( \left\lceil \frac{Access_m}{Ports_m} \right\rceil \right)
\]

Recurrence-constrained II

\[
II_{i,\text{min}}^{\text{rec}} = \max_p \left( \left\lceil \frac{Delay_p}{Distance_p} \right\rceil \right)
\]
2. Loop Pipelining: \( \text{initiation interval, } \) 
\[
Cycle_{Li} = D_i + II_i \cdot (Tc - 1)
\]

\[
II_{i,\text{min}} = \max \left( II_{i,\text{res},\text{min}}, II_{i,\text{rec},\text{min}} \right)
\]

Resource-constrained II

\[
II_{i,\text{res},\text{min}} = \max_m \left( \left\lfloor \frac{\text{Access}_m}{\text{Ports}_m} \right\rfloor \right)
\]

Recurrence-constrained II

\[
II_{i,\text{rec},\text{min}} = \max_p \left( \left\lfloor \frac{\text{Delay}_p}{\text{Distance}_p} \right\rfloor \right)
\]

\( II_{i,\text{min}} = \frac{2}{1} = 2 \)
2. Loop Pipelining: initiation interval, \( Cycle_{Li} = D_i + II_i \cdot (Tc - 1) \)

\[
II_{i,\min} = \max (II^{res}_{i,\min}, II^{rec}_{i,\min})
\]

**Resource-constrained II**

\[
II^{res}_{i,\min} = \max_m \left( \left\lceil \frac{Access_m}{Ports_m} \right\rceil \right)
\]

**Recurrence-constrained II**

\[
II^{rec}_{i,\min} = \max_p \left( \left\lceil \frac{Delay_p}{Distance_p} \right\rceil \right)
\]
Analytical Models: Performance Model

2. Loop Pipelining: initiation interval, \( Cycle_{Li} = D_i + II_i \cdot (Tc - 1) \)

\[
II_{i,\text{min}} = \max (II_{i,\text{res}}^{\text{min}}, II_{i,\text{rec}}^{\text{min}})
\]

Resource-constrained II

\[
II_{i,\text{res}}^{\text{min}} = \max_m \left( \left\lceil \frac{\text{Access}_m}{\text{Ports}_m} \right\rceil \right)
\]

Recurrence-constrained II

\[
II_{i,\text{rec}}^{\text{min}} = \max_p \left( \left\lceil \frac{\text{Delay}_p}{\text{Distance}_p} \right\rceil \right)
\]

Diagram showing the resource-constrained and recurrence-constrained II calculations with examples for each.
2. Loop Pipelining: initiation interval, \( Cycle_{Li} = D_i + II_i \cdot (Tc - 1) \)

\[
II_{i,\text{min}} = \max \left( II_{i,\text{res},\text{min}}, II_{i,\text{rec},\text{min}} \right)
\]

Resource-constrained II

\[
II_{i,\text{res},\text{min}} = \max_m \left( \left\lceil \frac{\text{Access}_m}{\text{Ports}_m} \right\rceil \right)
\]

Recurrence-constrained II

\[
II_{i,\text{rec},\text{min}} = \max_p \left( \left\lceil \frac{\text{Delay}_p}{\text{Distance}_p} \right\rceil \right)
\]

If the loop contains sub-functions, \( II_i = \max (II_{i,\text{min}}, II_{\text{sub},\text{max}}) \)
Analytical Models: Performance Model

2. Loop Pipelining: trip count, \( Cycle_{Li} = D_i + II_i \cdot (Tc - 1) \)

**Perfect** nested loop:

```c
for (int i = 0; i < 8; i++)
{
    for (int j = 0; j < 8; j++)
    {
        #pragma HLS PIPELINE II=1
        #pragma HLS UNROLL factor=2
        a[j] += b[j] * j;
    }
}
```

Outer loops are flattened to feed the inner loop with more data.

\[
Tc = \frac{B_i}{U_i} \cdot B_{i-1}B_{i-2} \cdots B_k
\]

\[
= \frac{8}{2} \cdot 8 = 32
\]

**Non-perfect** nested loop:

```c
for (int i = 0; i < 8; i++)
{
    for (int j = 0; j < 8; j++)
    {
        #pragma HLS PIPELINE II=1
        #pragma HLS UNROLL factor=2
        a[i] += b[j] * j;
    }
}
```

Codes between loop statements stop outer loops from flattening with the inner loop.

\[
Tc = \frac{B_i}{U_i} = \frac{8}{2} = 4
\]
3. Array Partitioning

**Partition number in dimension i**

- **Block:** \( P_i = \left\lfloor \frac{\text{index}_i}{\text{size}_i/f_i} \right\rfloor \)
- **Cyclic:** \( P_i = (\text{index}_i) \mod (f_i) \)
- **Complete:** \( P_i = \text{index}_i \)

**Partition number considering n-dimension array partitioning**

\[ P = P_1 + \sum_{i=2}^{n} (P_i \cdot \prod_{k=1}^{i-1} f_k) \]

Partition number shows which partition this array element is located in.
Analytical Models: Performance Model

4. Function Pipelining  “Fine-grain” pipelining: operators

\[ II = \max \left( II_{\text{res}}^{\text{min}}, II_{\text{max}}^{\text{sub}} \right) \]

- \( II_{\text{res}}^{\text{min}} \): resource-constrained minimum II,
- \( II_{\text{max}}^{\text{sub}} \): the maximum function II among all sub-functions.

5. Dataflow  “Coarse-grain” pipelining: functions and loops

\[ II = II_{\text{max}}^{\text{sub}} = \max_i (II_i^{\text{sub}}) \]

- \( II_{\text{max}}^{\text{sub}} \) is the maximum II among all sub-functions and sub-loops.
Analytical Models: Resource Model

1. DSP Estimation

   - **Sharable**: the maximum number of parallel operators
   - If a loop is **pipelined**, \[ N_{\text{op}}^{\text{min}} = \left\lfloor \frac{N_{\text{op}}}{II} \right\rfloor \]

2. BRAM Estimation

   \[ R_{\text{bram}} = \left\lfloor \frac{\text{#bits}}{\text{width}} \right\rfloor \cdot \left\lfloor \frac{\text{#element}}{\text{depth}} \right\rfloor \cdot \text{#partition} \cdot d \]

   - \( R_{\text{bram}} \): the number of **blocks** on BRAM.
Analytical Models: Resource Model

1. DSP Estimation

- Sharable: the **maximum** number of parallel operators
- If a loop is **pipelined**, \( N_{\text{min}}^{\text{op}} = \left\lfloor \frac{N_{\text{op}}}{II} \right\rfloor \)

2. BRAM Estimation

\[
R_{\text{bram}} = \left[ \frac{\text{#bit}}{\text{width}} \cdot \left\lfloor \frac{\text{#element}}{\text{depth}} \right\rfloor \cdot \#\text{partition} \cdot d \right]
\]

- **Width ratio**
- **Depth ratio**

- **#bit**: the width of each array element. e.g., int: 32 bits
- **width** is the **bandwidth** of the selected block configuration.
- **#element**: the number of **elements** in one memory partition.
- **depth** is the **depth** of the selected block configuration.
Analytical Models: Resource Model

1. DSP Estimation

- Sharable: the maximum number of parallel operators
- If a loop is pipelined, \( N_{\text{op}}^{\text{min}} = \left\lfloor \frac{N_{\text{op}}}{II} \right\rfloor \)

2. BRAM Estimation

\[
R_{\text{bram}} = \left[ \frac{\#\text{bits}}{\text{width}} \right] \cdot \left[ \frac{\#\text{element}}{\text{depth}} \right] \cdot \#\text{partition} \cdot d
\]

- \#partition: the number of memory partitions.
- \( d \): equals 2 if dataflow is applied; equals 1 if not.
Guided Design Space Exploration

Framework

1. RDC
2. Models
3. MGDSE

Stage: Redundancy Elimination

Delete redundant design points based on the rules of HLS tools.

- Top function: pipelined,
- Sub-loops: unrolled completely, cannot be pipelined.

Points related to LU, LP will be removed.

Stage: Guided Search

- To evaluate the performance of the current design point
- To determine the next design point

Evaluation metrics are proposed.
3 Metric-Guided Design Space Exploration

1. The **first** stage: Redundancy Elimination
   - To **remove** the **redundant** design points based on the rules of HLS tools.
     
     *E.g.,* top-function: pipelined,
     
     sub-loops: unrolled completely, cannot be pipelined.
     
     Points related to LU, LP will be removed.

2. The **second** stage: Guided Search
   - To **evaluate** the performance of the **current** design point
   - To **determine** the **next** design point
   
   ➤ **Three** evaluation **metrics** are proposed
Three evaluation metrics:

1. **Guided Design Space Exploration**

   \[
   M_{\text{diff}} = C_{\text{max}}^{\text{sub}} - C_{s, \text{max}}^{\text{sub}}
   \]

   - Find the **bottleneck**: the longest sub-function/sub-loop is assumed to have the greatest influence.

   \[
   M_{\text{res}} = \max \left( \frac{\text{DSP}_{\text{used}}}{\text{DSP}_{\text{total}}}, \frac{\text{BRAM}_{\text{used}}}{\text{BRAM}_{\text{total}}} \right)
   \]

   - Check the **resource** constraints: whether the resource usage exceeds the available resources on FPGA.

   \[
   M_{\text{aprt}, t} = \frac{\# \text{loads}}{\max \left( \sum_{i,k} \frac{\text{index}_i^t - \text{index}_i^k + 1}{\text{index}_i^t - \text{index}_i^k + 1} \right)}
   \]

   - Decide how to **partition**: which option is better, block or cyclic.

   \[
   M_{\text{aprt}, s} = \frac{\# \text{stores}}{\max \left( \sum_{j,k} \frac{\text{index}_j^t - \text{index}_i^k + 1}{\text{index}_j^t - \text{index}_i^k + 1} \right)}
   \]
MGDSE Flow

1. Yes
   - Sub-functions: pipelined
   - Loops: unrolled

2. No
   - Function pipelining
     - \( M_{\text{diff}} \)
     - Select the current longest sub-function or loop

3. True
   - Producer-consumer
     - \( M_{\text{diff}} \)
     - end

4. No
   - No pragma is set

Array optimization

- Set array partitioning
- Evaluate performance

- \( M_{\text{apt}}, M_{\text{res}} \)
- All arrays considered

- end

- sub-function/loop optimization

- Function or loop?
  - Function
    - Set LU, LP
    - Set array partitioning
  - Loop
    - All cases considered

- end
Framework

1. RDC
2. Models
3. MGDSE

Results

### Estimation Accuracy

#### From Polybench

<table>
<thead>
<tr>
<th>Benchmark</th>
<th>Lin_analyzer [1] (%)</th>
<th>COMBA (%)</th>
</tr>
</thead>
<tbody>
<tr>
<td>ATAX</td>
<td>2.68</td>
<td>0.71</td>
</tr>
<tr>
<td>BICG</td>
<td>1.24</td>
<td>0.33</td>
</tr>
<tr>
<td>GEMM</td>
<td>3.25</td>
<td>0.46</td>
</tr>
<tr>
<td>GESUMMV</td>
<td>5.15</td>
<td>1.25</td>
</tr>
<tr>
<td>MM</td>
<td>2.69</td>
<td>0.83</td>
</tr>
<tr>
<td>MVT</td>
<td>2.05</td>
<td>0.48</td>
</tr>
<tr>
<td>SYR2K</td>
<td>1.32</td>
<td>0.47</td>
</tr>
<tr>
<td>SYRK</td>
<td>2.78</td>
<td>0.44</td>
</tr>
</tbody>
</table>

The estimation error is reduced!

#### Results

<table>
<thead>
<tr>
<th>Application</th>
<th>COMBA (%)</th>
</tr>
</thead>
<tbody>
<tr>
<td>JPEG</td>
<td>1.54</td>
</tr>
<tr>
<td>Seidel</td>
<td>0.91</td>
</tr>
<tr>
<td>Rician</td>
<td>1.12</td>
</tr>
</tbody>
</table>

COMBA also works for more complicated applications!
## DSE Results: Comparison

<table>
<thead>
<tr>
<th>Benchmark</th>
<th>Design Space</th>
<th>Performance Speed-up</th>
<th>MGDSE Time (s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>ATAX</td>
<td>85</td>
<td>$1.31 \times 10^8$</td>
<td>8.35</td>
</tr>
<tr>
<td>BICG</td>
<td>95</td>
<td>$5.76 \times 10^8$</td>
<td>15.88</td>
</tr>
<tr>
<td>GEMM</td>
<td>85</td>
<td>$1.05 \times 10^{10}$</td>
<td>8.15</td>
</tr>
<tr>
<td>GESUMMV</td>
<td>85</td>
<td>$8.39 \times 10^8$</td>
<td>15.42</td>
</tr>
<tr>
<td>MM</td>
<td>85</td>
<td>$6.34 \times 10^{13}$</td>
<td>15.22</td>
</tr>
<tr>
<td>MVT</td>
<td>95</td>
<td>$1.05 \times 10^{10}$</td>
<td>15.30</td>
</tr>
<tr>
<td>SYR2K</td>
<td>85</td>
<td>$1.05 \times 10^{10}$</td>
<td>7.27</td>
</tr>
<tr>
<td>SYRK</td>
<td>85</td>
<td>$1.64 \times 10^8$</td>
<td>8.12</td>
</tr>
</tbody>
</table>

Larger design space, Better speed-up, Efficient design space exploration
DSE Results: Optimizations of Polybench

<table>
<thead>
<tr>
<th>Benchmark</th>
<th>Array size</th>
<th>Optimizations</th>
</tr>
</thead>
<tbody>
<tr>
<td>ATAX</td>
<td>16</td>
<td>Pipeline: top-loop; Unroll: 2; Partition: A:(block,8,2); x:(complete,16,1); y:(block,8,1); tmp:(not partitioned)</td>
</tr>
<tr>
<td>BICG</td>
<td>32</td>
<td>Pipeline: top-loop; Unroll: 2; Partition: A:(block,16,2) (block,2,1); s:(block,16,1); p:(block,8,1); r.q:(not partitioned)</td>
</tr>
<tr>
<td>GEMM</td>
<td>16</td>
<td>Pipeline: top-loop; Unroll: 2; Partition: B:(block,8,2) (block,2,1); C:(cyclic,2,2); A:(not partitioned)</td>
</tr>
<tr>
<td>GESUMMV</td>
<td>16</td>
<td>Pipeline: top-loop; Unroll: 2; Partition: A,B:(block,8,2) (block,2,1); x:(complete,16,1); y.tmp:(not partitioned)</td>
</tr>
<tr>
<td>MM</td>
<td>8</td>
<td>Pipeline: top-loops(at the same level); Unroll: 2,2; Partition: A,D,tmp:(complete,8,2); B,C:(complete,8,2) (complete,8,1)</td>
</tr>
<tr>
<td>MVT</td>
<td>16</td>
<td>Pipeline: top-function; Unroll: 16,16; Partition: y1,y2:(complete,16,1); A:(block,8,2) (block,2,1); x1,x2:(not partitioned)</td>
</tr>
<tr>
<td>SYR2K</td>
<td>16</td>
<td>Pipeline: top-loop; Unroll: 1; Partition: A,B:(complete,16,2); C:(not partitioned)</td>
</tr>
<tr>
<td>SYRK</td>
<td>16</td>
<td>Pipeline: top-loop; Unroll: 4; Partition: A,C:(complete,16,2)</td>
</tr>
</tbody>
</table>

For different code structures, the optimization schemes are different!
Case Study: JPEG

The optimal configuration found by COMBA: “FP_AllPartition”, which is to

- **pipeline** the top-function,
- **partition** the three arrays completely.

Design space: $7.61 \times 10^{12}$

MGDSE time: 613s

Speed-up: 28.8x
Conclusion

The proposed model-based framework, COMBA:

- Optimizes different applications with various code structures.
- Performs better than the state-of-the-art on simple benchmarks.
- Optimizes more complicated applications efficiently, within minutes.

Our tool is available at: https://github.com/zjru/COMBA
Thank you for listening!

Q & A