Synthesis-Aided Compiler
Classical vs. Synthesis Compiler

Classical Compiler
Classical vs. Synthesis Compiler

Classical Compiler

Input specification

Optimal program

Search space

Synthesis-Aided Compiler
Compiler Workflow

Spatial Program

Partitioner

Program + partition annotation
(logical cores)

Layout

Program + location annotation & routing info
(physical cores)

Code Separator

Per-core code with communication code

Code Generator

Constraint solving minimizing # of msgs

Quadratic Assignment Problem

a traditional transformation

Superoptimization

Per-core optimized machine code
Spatial programming model

Spatial Program

Partitioner

Program + partition annotation (logical cores)

Layout

Program + location annotation & routing info (physical cores)

Code Separator

Per-core code with communication code

Code Generator

Per-core optimized machine code
int a, b;
int ans = a * b;
Spatial programming model

```c
int@1 a, b;
int@3 ans = a * b;
```

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

Similar to [Chandra et al. PPoPP’08]
Spatial programming model

```
int@1 a, b;
int@3 ans = a *@2 b;
```

Partition Type

*pins data and operators to specific partitions (logical cores)*

Similar to [Chandra et al. PPoPP’08]
int @1 a, b;
int @3 ans = a * @2 b;

Do not need to handle data routing and communication code
Unspecified Partitions

How to compile a partially annotated program?

```java
int a, b;
int@3 ans = a * b;
```
How to compile a partially annotated program?

int@?? a, b;
int@3 ans = a *@?? b;
Partitioning Synthesizer

Program

Partitioner

Program + partition annotation (logical cores)

Layout

Program + location annotation & routing info (physical cores)

Code Separator

Per-core code with communication code

Code Generator

Per-core optimized machine code

constraint solving to minimize # of msgs
Program

Partitioner

Program + partition annotation (logical cores)

Layout

Program + location annotation & routing info (physical cores)

Code Separator

Per-core code with communication code

Code Generator

Per-core optimized machine code

superoptimization
Code Generator

Classical compiler backend

IR\(^1\) $\rightarrow$ IR\(^2\) $\rightarrow$ IR\(^3\) $\rightarrow$ … $\rightarrow$ Optimized machine code

Optimizing Code Gen

Our compiler backend

Naïve Code Gen $\rightarrow$ Machine code $\rightarrow$ Super-optimizer $\rightarrow$ Super-optimized machine code

Optimal program = minimum cost

[Massalin et al. ASPLOS’87, Bansal et al. ASPLOS’06, Gulwani et al. PLDI’11, …]
Empirical Evaluation

Our compiler produces code comparable to the expert’s code.

On a largest benchmark (MD5), Chlorophyll was
• 19% slower
• 31% less energy-efficient

On 3 critical functions, Chlorophyll found

<table>
<thead>
<tr>
<th>Func</th>
<th>Charley’s</th>
<th>Chlorophyll</th>
</tr>
</thead>
<tbody>
<tr>
<td>f</td>
<td>push over - push and pop pop and over $fffff</td>
<td>dup push or and pop or</td>
</tr>
<tr>
<td></td>
<td>or and or</td>
<td>[7 slots]</td>
</tr>
<tr>
<td></td>
<td>[19 slots]</td>
<td></td>
</tr>
<tr>
<td>g</td>
<td>a! push a and pop a - and over $fffff or and or</td>
<td>a! over or dup a and or nop or</td>
</tr>
<tr>
<td></td>
<td>[19 slots]</td>
<td>[9 slots]</td>
</tr>
<tr>
<td>i</td>
<td>a! push a - over $fffff or and or pop or</td>
<td>a! over - $fffff a or and or or</td>
</tr>
<tr>
<td></td>
<td>[18 slots]</td>
<td>[14 slots]</td>
</tr>
</tbody>
</table>