SEJITS: Productive Performance with Pattern-Specific Compilation
Shoaib Kamil, Armando Fox, Bryan Catanzaro, Katherine Yelick

### Four Main Ideas

1. Specializer == pattern-specific compiler
   - exploit pattern-specific strategies that may not generalize
   - target specific hardware per pattern
2. Can happen at runtime
3. Productivity language program always valid even without SEJITS support
   - i.e. vs. incompatibly extending syntax; inspired by DSEL vs. DSL argument
4. Specializers can be written in Productivity Language

### Producing an Answer vs. Producing Software

- SEJITS delivers adaptive parallel software
- SEJITS is a highly productive way to produce exactly the code variants you need
- SEJITS makes research code productive
- Exploit full libraries, tools, etc. of Productivity Lang
- Performance competitive with Efficiency Lang code
- Develop specializers to target new HW features

### A SEJITS Taxonomy

<table>
<thead>
<tr>
<th>Variant Selection</th>
<th>Code Generation</th>
</tr>
</thead>
<tbody>
<tr>
<td>SEJITS-0</td>
<td>None</td>
</tr>
<tr>
<td>SEJITS-1</td>
<td>In Efficiency Language Library</td>
</tr>
<tr>
<td>SEJITS-2</td>
<td>In Productivity Language Library</td>
</tr>
<tr>
<td>SEJITS-3</td>
<td>Single variant generated</td>
</tr>
<tr>
<td>SEJITS-4</td>
<td>Variant in Efficiency Language code generated from Productivity Language code</td>
</tr>
</tbody>
</table>

SEJITS-0: Efficiency Lang library exposed to Python
SEJITS-1: Efficiency Lang code statically precompiled w/ variant selection in Efficiency Lang code
SEJITS-2: Efficiency Lang code statically precompiled w/ variant selection in Productivity Lang code
SEJITS-3: Efficiency Lang source generated by translating Productivity Lang source and JIT-compiled
SEJITS-4: Multiple Efficiency Lang variants generated from Productivity Lang source, “runtime” empirical planning of which variant to use

### Example: Stencils for Ruby

- Ruby class encapsulates SG pattern
- body of anonymous lambda specifies filter function
- Introspection used to read AST of function body
- Code generator produces OpenMP for multicore x86
- ~1000-2000x faster than Ruby
- Minimal per-call runtime overhead
- 90% of pure C performance

### Two Approaches for Explicit Annotation

1. Annotate functions that fit into wide, shallow pattern
2. Encapsulate each pattern into its own OO class

### Wide Patterns: Data Parallel SEJITS with Copperhead

- Copperhead is a SEJITS Framework for Data Parallelism, embedded in Python
- Built on data parallel prelude: map, reduce, scan, sort, scatter, gather,…
- Currently has a specialist for CUDA

- See Bryan Catanzaro’s poster for details & demo

```ruby
@cu
def saxpy(a, x, y):
    return [a*xi + yi for xi, yi in zip(x, y)]
```

### Narrow Patterns: SEJITS with Python Classes Per-Pattern

- A collection of narrow specialists, each implemented as a Python class
- Shared infrastructure through inheritance
  - AST manipulation, Code Generation, Caching, etc.
  - Work in Progress
    - SDK for Specializer Writers
    - Basic Specializers to infrastructure
    - Parallel Map
- Goal: Democratization of Specializer Creation