SEJITS: Raising the Abstraction Level of Productivity Programming
Armando Fox, Bryan Catanzaro, Shoaib Kamal, Yunus Lee, Krste Asanovic, Kurt Keutzer, Dave Patterson

SEJITS in a nutshell: Selective, Embedded Just-in-Time Specialization
- Productivity programmers write in general purpose, modern, high level PLL
- SEJITS infrastructure specializes computation patterns selectively at runtime
- Specialization uses runtime info to generate and JIT-compile ELL code targeted to hardware
- Embedded because PLL's own machinery enables (vs. extending PLL interpreter)

Layering vs. "Stovepiping"
- Layering: one or a few common intermediate languages
  - Must be flexible enough to support many DSLs
  - And map to wide variety of HW

Virt. worlds | Data vts | Robotics | Music
---|---|---|---
Rendering | Probabilistic Physics Line Alg.

Traditional Layers
- App domains
  - Computation domains
  - Language

Thick Runtime Hardware
- OOG | GPU | SIMD | FPGA | ???

App domains | Computation domains | Language
---|---|---
Virt. | Data vts | Robotics | Music

SEJITS "Stovepipes"
- Dense Matrix | Sparse Matrix | Stencil

Thin Runtime Hardware
- OOG | GPU | SIMD | FPGA | ???

SEJITS Exploits Productivity Level Language (PLL) Mechanisms
1. Some functions in productivity app annotated as potentially specializable
2. SEJITS intercepts calls using dynamic language features, uses introspection to examine function's Abstract Syntax Tree
   - If AST contains function call or pattern known in local catalog, specializer is invoked and handed AST
3. Specializer generates source code in an efficiency language (C, OpenMP, CUDA, ...), compiles & links
4. Specialized function binary is called, results returned to productivity language
5. (Optional) performance recorded, code cached for future calls

Selective specialization: If any step fails, fall back to PLL (no need to JIT or specialize the whole app)

Early case study: Python + CUDA
- Ruby => OpenMP on multicore x86 (S. Kamil)
  - ~1000-2000x faster than pure Ruby
  - Minimal per-call overhead at runtime
- Python => x86a GPU (B. Catanzaro, Y. Lee)
  - Stencils & Category-reduce (image processing)
  - Python decorators denote specializable functions
  - ~1000x Faster than pure Python
  - 3x-12x slower than handcrafted CUDA (including specialization overhead)
- Overheads: Naive code generation & caching, Type propagation, CUDA compilation, data marshaling
- Productivity programmer only writes Python/Ruby, not CUDA or OpenMP

Status, Ongoing Work, Challenges
- Prototypes working for NVidia, x86 multicore, RAMP (SPARC v8)
- Generalize infrastructure for catalog, pattern matching, call site annotation, history
- Integrate with PySKI/autotuning
- Cloud computing: Integrate with Nexus
- Cloud/multicore synergy: specialize intra-node as well as generate cloud code
- Capture additional motifs as specializers

Subverting PLL Mechanisms
- Observation: mechanisms intended to promote reuse also enable SEJITS
  - Hypothetical: generate & JIT-compile efficiency code to replace PLL code for this function
    - Make decisions at runtime based on available HW, argument values, etc., vs. "static" autotuning
  - Observation: mechanisms intended to promote reuse also enable SEJITS
- Metaprogramming: generate & JIT-compile efficiency code to replace PLL code for this function
  - "Stovepiping" vs. "Layering"
- Embedded: SEJITS machinery uses PLL features, no need to modify or extend PLL interpreter

Other Opportunities
- Autotuning
  - SEJITS can intercept calls and substitute autotuned code (see PySKI)
  - Locus of control for making co-tuning decisions
- Cloud Computing
  - Generate Hadoop (Java) code expressing PLL computation as MapReduce
  - Generate code for multiple cloud frameworks

Conclusions
- Enables code-generation strategy per-function, not per-app
- Uniform approach to productive programming
  - Same app on cloud, multicore, autotuned libraries
- Research enabler
  - Incrementally develop specializers for different motifs, prototype HW
  - Don’t need full compiler & toolchain just to get started