Cerebral Blood Flow Simulation – Moving Towards a Parallel Implementation
Meriem Ben-Salah, Chris Chaplin, Razvan Carbunescu, Farzana Ansari

**Motivation**
- Productivity layer programming language (e.g., Python) vs. efficiency layer programming language (e.g., C++)
- 5x faster development and 3-10x fewer lines of code
- BLF: 10x-100x less performance without exploring hardware model
- Scientists → Productivity Layer Programmers
- Computational expensive problems and lack of computer science knowledge → Disaster!
- In need of a technique that permits:
  - high level knowledge of computer architecture abstraction
  - and simultaneously good performance
- SEJITS
- Scientists/Productivity Layer Programmer input is needed

**Current Implementation**
- **Blood flow simulation**
  - Prototype in Matlab
  - Code tested on simple geometries, additional testing necessary for validation and verification
  - Incomplete Python version
- **Image processing**
  - Algorithm code development underway for object identification and skeletonization; geometric measurement to come

**Numerics**
- Numerical Scheme: finite Volume Method in space and Forward Difference in time
- Complexity:
  - O(10^4) time steps, each time step is 10 microseconds long
  - Maximal time: ~5 to 10 seconds
  - O(10^4)-grid points, spatial grid size is 0.1 millimeter
  - Total length of vasculature: 1-1.5 meter
  - Time allotted: 30 seconds
  - Gigaflops: 240 Gigaflops/second

**Spatio-temporal Solution Space**

**Design Space**

**SEJITS – Selective Embedded Just In Time Specialization**
- Libraries are written in Low Level Programming Languages and are too specific
- Stick with a High Level Programming Language e.g., Python
- Hidden to the productivity layer programmer but good to know:
  - Computational Patterns (good for reuse) are selected if it is worth it and specialized at runtime
  - A specialized function is written in a low level language targeted to the "parallel" hardware architecture (so called stovepipe technique)
  - Use of Autotuning techniques (e.g., in Pyski, autotuned Linear Algebra methods embedded in Python)
  - The low level code is compiled at run time
  - The low level code is dynamically linked using the high level language interpreter

**SEJITS Implementation (CS 294 Class Project)**
- Majority of code has been implemented in Python using data types and linear algebra operations that are intrinsic to NumPy and SciPy
- Specializers have been developed for stencil operators and iterative solvers
- These specializers have been integrated with the Python code using AST
- Introduced a structural “meta-specializer” in the form of a parallel executor that compiles individual specializers into a single compilation unit
- Initial performance tests show an order of magnitude speed-up for the 3T compiled version when compared to the pure-Python implementation
- Thanks to Ben Carpenter, James Ide, and Steven Liu for their work on this project

**Future Work**

**Blood Flow Simulation**
- Verify and validate the blood flow simulation using the prototype
- Finish the Python implementation of the model
- Compare run-time performance of code as applied to typical CT scans

**Image Processing**
- Refine the algorithm and prototype
- Convert code to Python and then exploit SEJITS
- Combine both codes to achieve an overall analysis time of ~60 seconds

**Collaborators**

**Researchers**
- Meriem Ben-Salah, Chris Chaplin, Razvan Carbunescu
- Farzana Ansari

**Undergraduates**
- Sarah Tanaka
- Walter Caliboso
- Ben Carpenter
- James Ide
- Steven Liu

**Research supported by Microsoft (Award #024262) and Intel (Award #024894) funding and by matching funding by U.C. Discovery (Award #DIGST-10227).**

Additional support comes from Par Lab affiliates National Instruments, NEC, Nokia, NVIDIA, Samsung, and Sun Microsystems.

Contacts: Meriem Ben-Salah: ben-salah@cs.berkeley.edu, Chris Chaplin: cchaplin@cs.berkeley.edu, Razvan Carbunescu: caresan@eecs.berkeley.edu