Streaming Beyond Arrays

- **Goal**: Optimize apps with pointer-intensive data structures
  - Parallelize access to trees, hash tables, sets, linked list, Markov chains, ... 
  - Optimize memory behavior for irregular data structures
- **Approach**: Exploit use of high-level data structure accesses
  - Accesses to data structures have well-defined algorithmic behavior
  - Want the compiler to be able to reason about dependencies, ordering, and program complexity at this algorithmic level
  - Allows us to ignore the “sequential nature” of low-level data structure access code
  - Don’t want programmer to specify the implementation of data structures
  - No need for excessive dependence / alias analysis
- **Solution**: Make data access library calls “compiler-aware”
  - Take advantage of STL in C++, standard container in Java
  - Compiler can now reason about dependencies, ordering, and access patterns
  - “Smart library compilation” is invisible to programmers

Speculative / Transactional Threads

- **Goal**: Simplify development of parallel applications
  - No need for detailed understanding of dependencies in any code
  - Eliminate race and reduce the cost of synchronization
  - Just requires programming APIs and schemes to buffer speculative state . . .
- **API #1**: Code explicitly parallel tasks as transactions
  - Two ways to express transactions in code:
    - Programmer specifies speculative transaction code regions
    - Programmer specifies variables that can only be accessed specularly
    - Transactions must execute completely without hazards from other processors
    - Provides excellent parallel performance when transactions may modify the same data, but this actually occurs only very rarely
  - Can be used for automatic parallelization
    - Java works well this way, as in our IRMP system:
      - Can be used to aid manual parallelization
        - Allows programmer to focus on speedup, not correctness
        - Threads are typically much larger and more parallel this way

- **Hardware**: Adding transactions and TLS to Smart Memories
  - Introducing additional TenSilica instructions to control memory system
  - Designing low-overhead speculative buffering within a single tile
  - Two processors in a tile alternate in taking threads
  - Use control bits in the memory mate to indicate specularly read and written words
  - Not scalable as it is optimized to minimize all speculative threads
  - Designing full-chip speculative buffering mechanism
    - Threads can access any number of processors across the chip
  - Scalability critical, optimized for fast forward progress
  - Dependence violations will impose high overhead when they occur (hopefully rarely)

The Stanford Smart Memories Project: Software

- **TVM Code**
  - The Stanford Smart Memories Project: Software
  - Make data access library calls “compiler-aware”

The Smart Memories Compilation Process

- We have designed our compilation environment to be PCA compatible
- Will focus on C/C++ for threaded and Brook for streaming apps
- Threaded STL with transactions to facilitate parallel programming
- PCA high-level compilers to break down source code to SVM/TVM
- SVM: Reference Implementation Simulator

Brook: A High-Level Streaming Language

- **Our existing “upper-level” compilation framework**
- **Brook source code is just an extension to C**
  - Threads are actually in C itself
  - Streams are array-like views of memory
  - Data-parallel stream kernels look like C function calls, but work with stream I/O and are completely parallelizable
  - “Calls” to these kernels are expanded into threaded control code
  - An example, with Brook additions in bold:
    ```
    typedef stream float * float; 
    kernel void Add(floats A, floats B, float out(floats C)) {
      // C = A + B; 
    } 
    ```
  - Associative reductions can be specified with explicit operators

- **Support for multidimensional streams**
  - Allows programmer to easily take advantage of true “shape” of data
  - Lets programmer defines stencils of nearby neighbors in all dimensions which may be accessed by a kernel processing a stream element
  - Additional features support boundary conditions:
    - Periodic: Wraparound the edges
    - Halo: Elements in the boundary not updated
    - Clamp: Specified value for boundary cells
  - Multi-mode partitioning based on UPC

- **Many applications have been written in Brook**
  - PCA radar applications (FFT, FIR, beamforming, SVD)
  - StreamFlo (TLFO): A Navier-Stokes solver for external aerodynamics with a turbulence model
  - StreamMD: Molecular dynamics, Gromacs
  - StreamFEM: Discontinuous Galerkin (DG) Finite Element Method (FEM) on 2D triangulated domains

The SVM Code Model: Control and Kernel

- **SVM Control Code**: Management and thread interface
  - Normal TVM-based C code
  - Includes SVM API calls to launch and manage kernels
- **SVM Kernels**: Performing stream work
  - Work with stream and block data types
  - Supports a (often circular) queue of records located in memory
  - Blocks describe random-access regions of memory
  - Object constructor for stream blocks and define characteristics
  - Basic stream access
    - Pop/push records from/to a one-dimensional stream
    - `element = inputStream.pop()`
    - `outputStream.push(element)`
    - “Peeking” allows look-ahead without a pop
    - `element = inputStream.peek()`
  - Most code is a limited subset of C
    - No pointers, global accesses, etc. are allowed
  - Constants and result data must be part of kernel object

SVM: Reference Implementation Simulator

- Need a way to write and verify SVM code today
  - Want to allow application developers to start working with SVM
- Reference simulator emulates streaming hardware
  - Packages the SVM API into a library of C calls
  - Makes it possible to compile SVM code on a desktop computer
  - Runs control processor and “streaming processor(s)” as separate threads
- Simulator allows us to verify application development
  - Verify that SVM code produces correct results
- Estimate performance using:
  - Machine description parameters (BW, IntOPS/FloatOPS) from architecture metadata
  - Simulated memory bandwidth demands
  - Operation counts from kernel loop(s) and startup/shutdown overhead

SVM: Implementation on Smart Memories

- Smart Memories does streaming with full quadruples
  - 8 processors in a quad are all allocated as a single streaming coprocessor
  - All work together in MIMD mode, running the same kernel code
- Quad memory tiles have two uses:
  - Stream buffer for data (i.e. Imagine’s SRF)
  - Kernel instruction memory
    - Contains compiled stream code
    - Contains compiled kernel code
    - Requires caching
  - Synchronization using standard SM memory primitives
    - Uses the existing full/empty bit functionality in memory mats
    - This mechanism can perform arbitrary fine-grained synchronization (at a word level)
    - Sync between stream memory operations (stream/read_Store) and kernel start
    - Sync between kernel completion and controlling TVM thread
- Data movement is highly optimized
  - Different operands are used for access to local memory mats or other tiles’ mats
  - Allows efficient load/store between processing elements
  - DMA controller in cache/network interface handles stream loads and stores
- May add efficient instruction steering mechanism
  - Would allow us to share kernel instructions across a whole quad
  - Still evaluating costs and benefits of this feature