Probabilistic Reasoning Algorithms

**Goal:** Evaluate Smart Memories architectural features with a new generation of leading-edge algorithms

- Demonstrate utilization of reconfigurable components
- Understand the computation and communication requirements of these algorithms

**Algorithm #1:** Belief propagation for probabilistic graphs
- Iterate until convergence: 1. Computing messages to neighboring nodes based on current beliefs
- 2. Updating beliefs based on messages received
- Within each phase, each node’s computation is independent

**Algorithm #2:** Recursive conditioning of Bayesian nets
- Recursively traverse a binary tree representing the network
- At internal nodes, iterate over possible value assignments to a set of variables
- Task & loop parallelism available

Current status:
- Baseline parallel version implemented on Linux
  - Speedup of 1.6-1.7 on two processor SMP using p threads
- Programs being ported to the Stanford Smart Memories simulation environment
  - Algorithms will adapt to use SM memory and nets effectively
  - Will also adapt to utilize SM light-weight thread support

Transnational Coherence and Consistency

**Goal:** Ease parallel programming with a new shared memory coherence and consistency protocol

- Want to let parallel programmer focus on performance, not correctness

**Approach:** Have all processors continually execute transactionally
- Processors speculatively execute atomic regions of code called transactions
- Exposed reads and writes to other processors’ caches
- Must restart if others write to these locations
- Writes buffered until end of each transaction
- Writes not visible to other CPUs until end of T
- Forms a continuous transaction cycle
  1. Speculatively execute
  2. Allocate for committing transaction
  3. Broadcast write globally in a block transfer
  4. Other CPUs snoop and restart if necessary
  5. Repeat (double-buffering can allow early repeat)

- Allows parallelization without regard to transaction dependencies in the original code, much like thread-level speculation (TLS)
- Snooping and invalidating when necessary ensures that correct execution occurs
- Synchronization handled by commit arbitration control
- Processors only synchronize at transaction commit points
- Unordered transactions arbitrate among each other equally at commit time
- Ordered transactions (TLS-like node or at barriers) are prioritized by phase number

**Results:** Have analyzed execution traces from parallel programs
- Parallel performance is typically similar to or better than conventional
- Much easier to program, with only minor changes to sequential applications

The Smart Memories Compilation Process

**Goal:** 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
  - Java can also be used in a more conventional manner
- PCA high-level compilers to break down source code to SVM/TVM
  - xecc compiler is automatically generated by Tensilica tools
  - We will add compiler passes to allow additional parallelization functions
  - Will leverage existing IRPM Java dynamic compilation framework

SAPIENT Application Analysis Framework

**Goal:** Efficient transformation of sequential applications for parallel execution on polymorphic architectures

**Approach:** Profile and analyze individual regions of application execution to find available parallelism automatically
- Identifies data-level parallel (DLP) and thread-level parallel (TLP) loops
- Current focus is on finding opportunities for manual parallelization of loops
- Provides feedback to the programmer and provides feedback
- Enables informed decisions regarding how to apply parallel programmer and what kinds of performance tradeoffs must be considered

Using Streams with GPUs

**Goal:** Allow the streaming execution model to work on programmable 3D graphics acceleration processors (GPUs)
- Graphics processors are fast, and improving faster than general purpose CPUs:
  - Pentium 4 SSE theoretical: 5GHz / 4 wide x 5 in/cycle = 4 GFLOPS
  - GeForce FX 9500 NV35 fragment shader observed: 20 GFLOPS, equivalent to a 10 GHz P4
  - and getting faster: 3x improvement over NV30 (6 months)

**Method #1:** Compile Brook directly to GPUs
- Our tool, BRCC, converts Brook kernels to a combination of nVIDIA’s Cg kernel language and normal C/C++
- Some challenges for scatter operations & reductions
- Moving towards a virtualization of hardware
  - Automatic blocking for better texture cache usage
  - Splitting kernels when extra registers necessary
  - Mapping multidimensional streams to 2D textures
- Several applications already ported:
  - Kernels like SAXPY and FPT
  - Compared against Intel Math Library
  -� 36% version uses SSE and cache blocking
  - 2-3x faster than CPU implementation
  - Bitonic sort
  - LINPACK & sparse matrix multiply
  - Gromacs (molecular dynamics solver)
  - Ray tracer
- Source is publicly available: www.sourceforge.net/projects/brook
- Nearly 5,000 downloads in less than 3 months

**Method #2:** Compile SVM code to GPUs
- Convert tools to use SVM code in order to use R-Stream compiler
- Allows us to verify the SVM model on real “streaming” hardware
- SVM Machine model for a GPU is somewhat unusual
- Machine models are composed of Processors, Memories and Network links all characterized by performance (Gops, BW, SIMD)
- GPUs are good candidates as Stream Processors
- We’ve examined the best offerings from nVIDIA and ATI
- These match performance ratios of dedicated stream processors like Imagine
- Big difference is that “local” memory is off-chip texture DRAM, ~256 MB sizes today
- nVIDIA’s NV30 architecture local RAM allows whole datasets to be in “local” memory

**SVM simulator validation**
- SVM simulator estimates performance of run-time of application based on machine model performance characteristics
- Used micro-kernels written in SVM code to evaluate performance characteristics
- Will use larger applications when the R-Stream compiler becomes more robust
- Ran applications on several platforms and compared results:
  1. 3 kinds of hardware: Imagine streaming processor, ATi and nVIDIA GPUs
  2. Also ran all applications on the SVM simulator with all three machine models
- We found that the SVM simulator predicts hardware performance very well
- Shows that the SVM simulator will be a good tool for analyzing streaming code
- Indicates that the SVM is a good intermediate representation and compiler target

**SAPIENT Application Analysis Framework**