The Stanford Smart Memories Project: Software

M. Horowitz, W. Dally, D. Engler, P. Hanrahan, C. Kozyrakis, M. Lam, K. Olukotun, M. Rosenblum

PCA Compilation Process

A two-level compiler should be used with PCA architectures
- Upper level compiles to an architecturally neutral VM
- Lower level compiles from VM to architecture

Architecture “Virtualization”

But PCA architectures are complex . . .
- VM is parameterized with metadata
- Architecture model for a wide range of architectures can be built with small, standardized VM building blocks

Use four different standard building “nodes” to represent HW
- Threaded VM blocks represent “conventional” processors
- Streaming VM blocks represent slave streaming processors
- Memory blocks
- Generic network segments to connect other nodes

Put together, these “virtualize” an entire architecture . . .

The Compiler Framework

The upper-level compiler must now . . .
- Break up application into:
  - Threads which can run on DVVM-modeled nodes
  - Stream kernels which can be run remotely on streaming nodes
- Map threads, kernels, and data to virtualized architecture
- Insert explicit communication between nodes
- Generate final TVM/SVM code and memory map

The lower-level compiler must now . . .
- Expand communication between nodes
- Do intra-node compilation and optimization

The “New” SVM: Thread Interface

Threads create Graph objects consisting of a set of kernels

The “New” SVM: Kernel API

Stream kernels are C++ routines that process stream elements
- Work with Stream and Block data types
- Streams represent a (often circular) queue of records located in memory
- Blocks describe random-access regions of memory
- Object constructors locate streams/block and define characteristics

Basic stream access
- Pop/push elements from/to a one-dimensional stream
- “Peeking” allows lookahead without a pop
- element = outStream.peek(lastelement_recode)

Stream testing
- Check to see if there’s space for pushing or elements for popping
- Check if stream is full before pushing

Block access
- Blocks use direct read/write access
- element = InBlock.read(element_index)
- outBlock.write(element_index, element)

Rest of code is just a limited subset of C
- No pointers, global accesses, etc. are allowed
- Constants and reduction results must be part of kernel object
- Kernel must define its own setup routines and stack
- Annotations describe kernel characteristics
- Notations marking SIMD-parallelizable kernels

Brook: A High-Level Streaming Language

- Our existing “upper-level” compilation framework
- Brook source code is just an extension to C
- Streams 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:

```c
typedef stream float (*stream_func);

void Add(float A, float B, float 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 define 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
  - Male: Elements in the boundary not updated
  - Clamp: Specific value for boundary cells
- Multi-node partitioning based on UPC

- Many applications have been written in Brook
  - Sections of PCA radar applications (FFT, FIR, beamforming, SVD)
  - Brook’s (TFO): 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

Streams on Graphics Processors

The latest graphics processors can execute simple but computationally intensive fragment programs
- Typical 2003 performance figures:
  - 16 GFLOPS of arithmetic power
  - 20 GB/s of texture memory bandwidth
  - About 1k assembly instructions per fragment

We want to use this hardware for other purposes, also
- Goal: Port Brook programs to graphics hardware
  - We convert Brook kernels to nVIDIA’s Cg kernel language
  - Working towards a relatively automatic solution
  - Some challenges for scatter operations and efficient reductions
  - Moving towards a virtualization of hardware resources
  - Automatic blocking for better texture cache performance
  - Splitting kernels when extra registers are necessary
  - Mapping multidimensional streams to 2D textures

Several applications have been ported already:
- Bitonic sort
- Sparse matrix multiply
- Gromacs (molecular dynamics solver)