MIMD Interpretation on a GPU

Hank Dietz and Dalton Young

LCPC, Oct. 8, 2009

University of Kentucky
Electrical & Computer Engineering

GPUs?

- Graphics Processing Units
- Lots of PEs, each with FP hardware
- Cheap & scalable hardware...
  - SIMD-ish multi-threaded execution using multiple, simplified, narrow SIMD engines
  - The host does all the messy stuff
- Programming model is quirky and dominated by vendor-dependent languages

MIMD on a GPU?

- Hide the quirks & improve portability
- Use MIMD programs & programming tools
- Hardware isn't converging on a simple design: Intel Larabee & AMD Fusion
- MIMD execution on SIMD was done before; why not MIMD on a GPU?

Basic MIMD Interpreter

1. `IR=mem[PC++]`
2. Decode instruction from `IR`
3. Repeat for each instruction type:
    1. Disable PEs where `IR`!=instruction
    2. Simulate instruction
    3. Enable all PEs
4. Goto 1

Performance Issues

- Interpretation overhead
  - Coding of `switch` statement
  - Sum of instruction simulation times
- Indirection – each PE from it's own address
  - Banking, caching, & “owner writes”
  - `mem[N/W][M][W]` memory layout
- Masking overhead
  - Divergent flow (within a warp)
  - Predication
  - Skipping (warps)

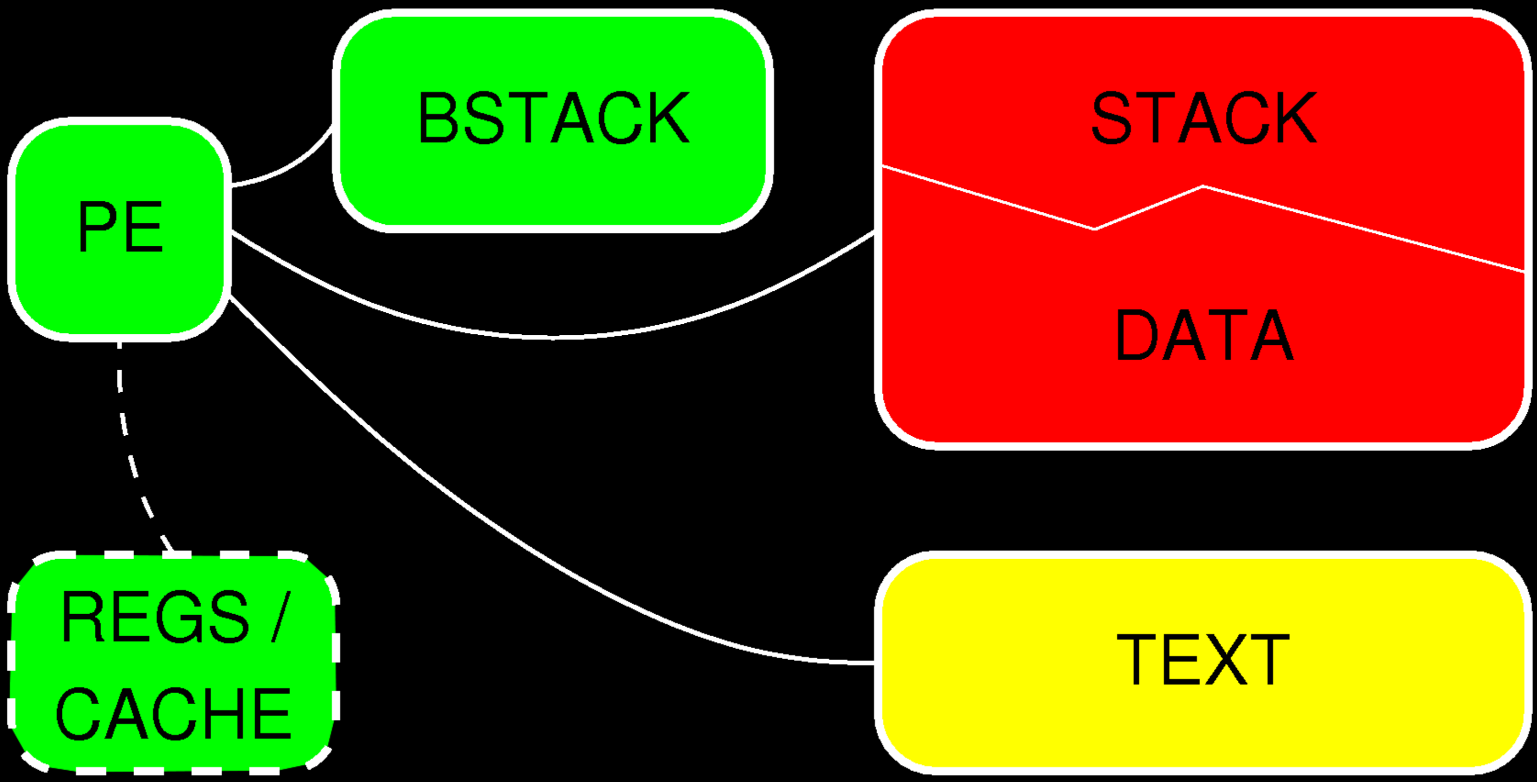
Assembler (`mogasm`)

- Multi-pass assemble to binary image coded as initialized data structures for `mogsim`
- Can combine multiple related/independent programs/libraries with conditional assembly; supports multi-lingual MIMD, not just SPMD
- Instruction bit patterns & field layout (8, 16, or 32-bit instruction words) can be automatically customized per application

Simulator (`mogsim`)

- About 2,500 lines of C/CUDA source code (compiler, assembler, etc. ~70,000 lines)
- C code repeatedly calls CUDA `emulate()`, which runs until timeout or `SYS`
- Can be a generic interpreter or automatically recoded to optimize a specific application
- Currently runs on any NVIDIA CUDA GPU

MOG PE Structure



Sequence of Single-Instruction Subinterpreters (SIS)

- Subinterpreter handles just 1 instruction type
- Order subinterpreters to minimize cycles
- Frequency bias subinterpreter execution
- Consider the code: `PUSH LD ADD ADD`
  - `ADD LD PUSH` takes 4 cycles
  - `PUSH ADD LD` takes 3 cycles
  - `PUSH LD ADD` takes 2 cycles
  - `PUSH LD ADD ADD` takes just 1 cycle

Determining the Subinterpreter Sequence for SIS & Opt-SIS

- Analysis based on instruction and instruction digram frequencies from application runs
- Instruction frequencies determine mix
- Genetic algorithm evolves best order by minimizing sum of frequency-weighted digram spans
- Order using a generic application is SIS, using the selected application is Opt-SIS

Selection of a Present Instruction to Interpret (SIR)

- Method ensures fairness & progress
- Each PE fetches an instruction into his IR
- The designated PE within each warp copies his IR into the warp-shared IR (SIR)
- All PEs decode SIR, but only those where IR==SIR perform the instruction
- Opt-SIR uses decoder tree optimized using application statistics

Divergent Factored Decoding (DFD)

- Decoding is slow; why not let each PE decode the instruction in its IR, diverging, but partially factoring decode?
- Decode is accomplished via an optimized decode tree with the opcodes remapped for the application in Opt-DFD

Factoring using Common Subexpression Induction (CSI)

- The most effective method for MasPar MP1
- Break each instruction into microinstructions
- Maximally factor the microinstructions, inducing common subexpressions
- Minimizes cost of PE memory references & other expensive micro-ops, but increases conditionals & per-PE state

The SW Variants (Opt-SIR-SW, Opt-DFD-SW, Opt-CSI-SW)

- Opt variants rebuild `mogasm` and `mogsim` for the particular application (profiling)
- SW variants use `switch` instead of a decode tree:
  - Opt-SIR-SW
  - Opt-DFD-SW
  - Opt-CSI-SW

Experiments

- GPU MOG vs. GPU Native (not vs. CPU)
- Two simple per-PE benchmark codes:
  - `perf`: 1M SIMD multiply-accumulates
  - `fact`: 10K recursive, divergent, MIMD!
- Executed on various NVIDIA CUDA GPUs with various host processors
- All 11 approaches tested everywhere...

UNIVERSITY OF KENTUCKY

Benchmark System Configurations

Feature	"Laptop"	"Desktop1"	"Desktop2"	"Desktop3"
Host Processor	Intel T8300	AMD 4200+	Intel 920	AMD 4200+
NVIDIA GPU (CC)	8600M GT (1.1)	8800 GTS (1.0)	9800 GT (1.1)	GTX 280 (1.3)
GFLOPS: Host/GPU	9.2 / 91.2	10.5 / 345.6	21.36 / 544.3	10.5 / 933
Power: Host/GPU	35 / 22	89 / 146	130 / 125	89 / 236
GPU Cores/PEs	32 / 1,024	96 / 2,304	112 / 3,584	240 / 10,560
Best Time: `perf`/`fact`	9.63 / 10.55	7.77 / 7.7	6.66 / 7.2	8.33 / 9.76

Kentucky UNBRIDLED SPIRIT™

Experimental Results

- Difference between GPUs was small and trends were very similar on every target (remember work scales with # of PEs)
- For `perf` (best native 1.46s):
  - Worst-case MOG slowdown ~6.6X
  - SYS calling native slowdown ~1.7%
- For `fact` (not natively possible):
  - No recursion support in CUDA
  - Making CUDA interuptable ~4X

Performance for `perf` & `fact`