Arch: Data-Level Parallelism

Alex Chi

Update: May 21, 2020

Contents

1 Introduction

1.1 Classification

- SISD: Single Instruction, Single Data
- SIMD: Single Instruction, Multiple Data
- MIMD: Multiple Instruction, Multiple Data
- MISD: Multiple Instruction, Single Data

1.2 Approaches to Exploiting Parallelism

- Data-Level Parallelism: execute multiple operations of the same type (vector / SIMD execution)
- Instruction-Level Parallelism: execute independent instructions from one stream in parallel (pipelining, superscalar, VLIW)

- Thread-level Parallelism: execute independent instruction stream in parallel

1.3 Introduction

- SIMD for matrix computing and media processing
- SIMD is more energy-efficient than MIMD
- SIMD allow programmer to think sequentially

1.4 SIMD

- single instruction, multiple data
- SIMD parallelism
  - vector architectures
  - SIMD extensions
  - GPU

2 Vector Architecture

- basic idea: read sets of data elements into vector registers
- operate on those registers
- write result back to memory
- compiler controls these registers (used to hide memory latency, leverage memory bandwidth)

2.1 VMIPS Architecture

- vector registers (64 elements * 64 bits per reg, 16 R + 8 W ports)
- vector functional units (fully pipelined + detect data and control hazard)
- vector load-store unit (fully pipelined + one word per clock cycle after latency)
- scalar registers (32 GP, 32 FP)

2.2 VMIPS ISA

- refer to slides
- arithmetic: V+S, V+V, V-S, V-V, S-V, V*S, V*V, V/S, S/V, V/V

- memory: L/S R, L/S B+kR, L/S V[i]
- compact, expressive, scalable
  - 1 inst for N operations
  - machine-independent inst encoding

2.3 Example

- `Y = aX + Y`
- MIPS
- VMIPS

```
L.D F0, a
LV V1, Rx
MULVS.D V2, V1, F0
LV V3, Ry
ADDVV.D V4, V2, V3
SV V4, Ry
```

2.4 Vector Arithmetic Execution

- deep pipeline
- no hazard detection, as elements are independent

2.5 Vector Execution Time

- depend on 3 factors
  - length of operand vectors
  - structural hazards
  - data dependencies
- FU consumes 1 element per cycle

2.6 Vector Chaining

- like forwarding in pipeline / register renaming in Tomasulo
- forward data from load unit to FU, and between FU
- as soon as data arrive

- instead of write to register, then read from register

2.7 Optimization

- use lane, multiple elements per clock cycle
- automatic vectorization with compiler
- handle unaligned vector: use vector length register
- handle conditional statement: masked vector instructions
- requirement for memory system: bandwidth
  - vector memory-to-memory system
  - vector register machines (cannot directly operate on memory)
- multi-dimensional matrices: use non-unit stride
- sparse matrices: use index vectors

3 SIMD Instruction Set Extensions

3.1 SIMD Multimedia Extensions

- application operate on narrower data types
- usually 64-bit register
- newly designs have 256-bit registers

3.2 MMX Instructions

- 8 8b, 4 16b, 2 32b in parallel
- disconnect carry chain

3.3 Why Multimedia SIMD popular?

- cost little to add instruction
- vector architecture requires a lot of memory bandwidth
- do not have to deal with virtual memory page fault

3.4 Cons

- limited instruction set
- limited vector register length
- limited mask registers

4 GPU

- 3 stage of GPU
  - fixed pipeline
  - programmable pipeline
  - GPGPU

4.1 Life of a Triangle (Mesh)

- vertex processing: map mesh to triangle in screen space
- rasterisation: triangle to pixels on screen
- raster operations: fill with color
- fragment processing: output to screen

4.2 GPU Architecture

- a GPU has hundreds of cores
- inside a core there's many different processors
- GPU core = CPU core without CU
- GPU Processor = CPU Function Unit

4.3 CUDA: Heterogeneous Computing

- CPU: serial code
- GPU: highly parallel
- other GPGPU libraries: OpenCL
- graphics library which has same functionality: OpenGL / DirectX / Metal, shader is similar to CUDA code

4.3.1 Overview

- kernel: grid of thread blocks

4.4 Hardware Execution Model

- CPU Memory - CPU - GPU - GPU Memory
- GPU is built from multiple parallel cores (streaming multi-processors)
- each core a SIMD processor
- CPU sends grid to GPU, it distributes thread blocks among cores

4.5 Memory Hierarchy

- registers (R/W thread)
- local memory (R/W thread)
- shared memory (R/W block)
- global memory (R/W grid)
- constant memory (R-o grid)
- texture memory (R-o grid)

4.6 CUDA code example

- refer to slides
- grid works on 8192 elements
- each block works on 512 elements (16 b per g)
- one SIMD inst works on 32 elements (16 SIMD per b)
- CUDA wrap = SIMD thread
- CUDA thread = operation on one element

4.7 Multi-threaded SIMD processor

- wrap scheduler + scoreboard
- a wrap executes one instruction
- SIMD has a scheduler
- issue instructions to all SIMD lanes
- wraps are independent

- global block scheduler: distribute blocks of threads to SM

4.8 Life Cycle of a Thread

- grid started on GPU
- block of threads allocated to SM
- SM organizes threads to wraps
- Wraps are scheduled on SM

5 Review

- Fundamentals
  - dependability
  - performance
  - quantitative principles
- Instructions: MIPS
- Single-cycle and Multi-cycle Processor
  - datapath + control
- Pipelining and ILP
  - pipeline
  - forwarding
  - hazards
  - dynamic branch prediction
- Memory Hierarchy Design
  - cache organization
  - basic / advanced optimization
- Exploiting DLP
  - vector architecture optimizations
  - SIMD extensions
  - GPU