Computer Architecture: VLIW, DAE, Systolic Arrays

Prof. Onur Mutlu
Carnegie Mellon University
A Note on This Lecture

- These slides are partly from 18-447 Spring 2013, Computer Architecture, Lecture 20: GPUs, VLIW, DAE, Systolic Arrays

- Video of the part related to only SIMD and GPUs:
  - [http://www.youtube.com/watch?v=vr5hbSkb1Eg&list=PL5PHm2jkkXmidJOD59REog9jDnPDTG6IJ&index=20](http://www.youtube.com/watch?v=vr5hbSkb1Eg&list=PL5PHm2jkkXmidJOD59REog9jDnPDTG6IJ&index=20)
VLIW and DAE
Remember: SIMD/MIMD Classification of Computers


- **SISD**: Single instruction operates on single data element
- **SIMD**: Single instruction operates on multiple data elements
  - Array processor
  - Vector processor
- **MISD?** Multiple instructions operate on single data element
  - Closest form: systolic array processor?
- **MIMD**: Multiple instructions operate on multiple data elements (multiple instruction streams)
  - Multiprocessor
  - Multithreaded processor
SISD Parallelism Extraction Techniques

- We have already seen
  - Superscalar execution
  - Out-of-order execution

- Are there simpler ways of extracting SISD parallelism?
  - VLIW (Very Long Instruction Word)
  - Decoupled Access/Execute
VLIW
VLIW (Very Long Instruction Word)

- A very long instruction word consists of multiple independent instructions packed together by the compiler
  - Packed instructions can be logically unrelated (contrast with SIMD)

- Idea: Compiler finds independent instructions and statically schedules (i.e. packs/bundles) them into a single VLIW instruction

- Traditional Characteristics
  - Multiple functional units
  - Each instruction in a bundle executed in lock step
  - Instructions in a bundle statically aligned to be directly fed into the functional units

ELI: Enormously longword instructions (512 bits)
SIMD Array Processing vs. VLIW

- Array processor
VLIW Philosophy

- Philosophy similar to RISC (simple instructions and hardware)
  - Except multiple instructions in parallel

- RISC (John Cocke, 1970s, IBM 801 minicomputer)
  - Compiler does the hard work to translate high-level language code to simple instructions (John Cocke: control signals)
    - And, to reorder simple instructions for high performance
  - Hardware does little translation/decoding → very simple

- VLIW (Fisher, ISCA 1983)
  - Compiler does the hard work to find instruction level parallelism
  - Hardware stays as simple and streamlined as possible
    - Executes each instruction in a bundle in lock step
    - Simple → higher frequency, easier to design
VLIW Philosophy (II)

More formally, VLIW architectures have the following properties:

There is one central control unit issuing a single long instruction per cycle.

Each long instruction consists of many tightly coupled independent operations.

Each operation requires a small, statically predictable number of cycles to execute.

Operations can be pipelined. These properties distinguish VLIWs from multiprocessors (with large asynchronous tasks) and dataflow machines (without a single flow of control, and without the tight coupling). VLIWs have none of the required regularity of a vector processor, or true array processor.
Commercial VLIW Machines

- Multiflow TRACE, Josh Fisher (7-wide, 28-wide)
- Cydrome Cydra 5, Bob Rau
- Transmeta Crusoe: x86 binary-translated into internal VLIW
- TI C6000, Trimedia, STMicro (DSP & embedded processors)
  - Most successful commercially

- Intel IA-64
  - Not fully VLIW, but based on VLIW principles
  - EPIC (Explicitly Parallel Instruction Computing)
  - Instruction bundles can have dependent instructions
  - A few bits in the instruction format specify explicitly which instructions in the bundle are dependent on which other ones
VLIW Tradeoffs

Advantages
- No need for dynamic scheduling hardware → simple hardware
- No need for dependency checking within a VLIW instruction → simple hardware for multiple instruction issue + no renaming
- No need for instruction alignment/distribution after fetch to different functional units → simple hardware

Disadvantages
- Compiler needs to find N independent operations
  - If it cannot, inserts NOPs in a VLIW instruction
  - Parallelism loss AND code size increase
- Recompilation required when execution width (N), instruction latencies, functional units change (Unlike superscalar processing)
- Lockstep execution causes independent operations to stall
  - No instruction can progress until the longest-latency instruction completes
VLIW Summary

- VLIW simplifies hardware, but requires complex compiler techniques
- Solely-compiler approach of VLIW has several downsides that reduce performance
  - Too many NOPs (not enough parallelism discovered)
  - Static schedule intimately tied to microarchitecture
    - Code optimized for one generation performs poorly for next
  - No tolerance for variable or long-latency operations (lock step)
- Most compiler optimizations developed for VLIW employed in optimizing compilers (for superscalar compilation)
  - Enable code optimizations
- VLIW successful in embedded markets, e.g. DSP
DAE
Motivation: Tomasulo’s algorithm too complex to implement
- 1980s before HPS, Pentium Pro

Idea: Decouple operand access and execution via two separate instruction streams that communicate via ISA-visible queues.

Decoupled Access/Execute (II)

- Compiler generates two instruction streams (A and E)
  - Synchronizes the two upon control flow instructions (using branch queues)

q = 0.0
Do 1 k = 1, 400
1 x(k) = q + y(k) * (r * z(k+10) + t * z(k+11))

Fig. 2a. Lawrence Livermore Loop 1 (HYDRO EXCERPT)

<table>
<thead>
<tr>
<th>Access</th>
<th>Execute</th>
</tr>
</thead>
<tbody>
<tr>
<td>A7 = -400</td>
<td></td>
</tr>
<tr>
<td>A2 = 0</td>
<td></td>
</tr>
<tr>
<td>A3 = 1</td>
<td></td>
</tr>
<tr>
<td>X2 = r</td>
<td></td>
</tr>
<tr>
<td>X5 = t</td>
<td></td>
</tr>
<tr>
<td></td>
<td><strong>load loop invariants</strong></td>
</tr>
<tr>
<td>loop: X3 + z + 10, A2</td>
<td></td>
</tr>
<tr>
<td>X7 + z + 11, A2</td>
<td><strong>load z(k+10)</strong></td>
</tr>
<tr>
<td>X4 + X2 *f X3</td>
<td><strong>load z(k+11)</strong></td>
</tr>
<tr>
<td>X3 + X5 *f X7</td>
<td><strong>r*z(k+10)-flt. mult.</strong></td>
</tr>
<tr>
<td>X7 + y, A2</td>
<td>t * z(k+11)</td>
</tr>
<tr>
<td>X6 + X3 +f X4</td>
<td><strong>load y(k)</strong></td>
</tr>
<tr>
<td>X4 + X7 *f X6</td>
<td><strong>r<em>z(x+10)+t</em>z(k+11))</strong></td>
</tr>
<tr>
<td>A7 + A7 + 1</td>
<td><strong>increment loop counter</strong></td>
</tr>
<tr>
<td>x, A2 + X4</td>
<td><strong>store into x(k)</strong></td>
</tr>
<tr>
<td>A2 + A2 + A3</td>
<td><strong>increment index</strong></td>
</tr>
<tr>
<td>JAM loop</td>
<td><strong>Branch if A7 &lt; 0</strong></td>
</tr>
</tbody>
</table>

Fig. 2b. Compilation onto CRAY-1-like architecture

Fig. 2c. Access and execute programs for straight-line section of loop
Decoupled Access/Execute (III)

Advantages:
+ Execute stream can run ahead of the access stream and vice versa
  + If A takes a cache miss, E can perform useful work
  + If A hits in cache, it supplies data to lagging E
  + Queues reduce the number of required registers
  + Limited out-of-order execution without wakeup/select complexity

Disadvantages:
-- Compiler support to partition the program and manage queues
  -- Determines the amount of decoupling
-- Branch instructions require synchronization between A and E
-- Multiple instruction streams (can be done with a single one, though)
Astronautics ZS-1

- Single stream steered into A and X pipelines
- Each pipeline in-order

Astronautics ZS-1 Instruction Scheduling

- Dynamic scheduling
  - A and X streams are issued/executed independently
  - Loads can bypass stores in the memory unit (if no conflict)
  - Branches executed early in the pipeline
    - To reduce synchronization penalty of A/X streams
    - Works only if the register a branch sources is available

- Static scheduling
  - Move compare instructions as early as possible before a branch
    - So that branch source register is available when branch is decoded
  - Reorder code to expose parallelism in each stream
  - Loop unrolling:
    - Reduces branch count + exposes code reordering opportunities
Loop Unrolling

**Idea:** Replicate loop body multiple times within an iteration
+ Reduces loop maintenance overhead
  - Induction variable increment or loop condition test
+ Enlarges basic block (and analysis scope)
  - Enables code optimization and scheduling opportunities
-- What if iteration count not a multiple of unroll factor? (need extra code to detect this)
-- Increases code size
Systolic Arrays
Why Systolic Architectures?

- **Idea:** Data flows from the computer memory in a rhythmic fashion, passing through many processing elements before it returns to memory.

- Similar to an assembly line
  - Different people work on the same car
  - Many cars are assembled simultaneously
  - Can be two-dimensional

- **Why?** Special purpose accelerators/architectures need
  - Simple, regular designs (keep # unique parts small and regular)
  - High concurrency $\rightarrow$ high performance
  - Balanced computation and I/O (memory access)
Systolic Architectures


![Diagram of systolic array](image)

**Figure 1. Basic principle of a systolic system.**

- Memory: heart
- PEs: cells
- Memory pulses data through cells
Systolic Architectures

- Basic principle: Replace a single PE with a regular array of PEs and carefully orchestrate flow of data between the PEs → achieve high throughput w/o increasing memory bandwidth requirements

- Differences from pipelining:
  - Array structure can be non-linear and multi-dimensional
  - PE connections can be multidirectional (and different speed)
  - PEs can have local memory and execute kernels (rather than a piece of the instruction)
Systolic Computation Example

- Convolution
  - Used in filtering, pattern matching, correlation, polynomial evaluation, etc ...
  - Many image processing tasks

Given the sequence of weights \( \{w_1, w_2, \ldots, w_k\} \)
and the input sequence \( \{x_1, x_2, \ldots, x_n\} \),
compute the result sequence \( \{y_1, y_2, \ldots, y_{n+1-k}\} \)
defined by

\[
y_i = w_1x_i + w_2x_{i+1} + \ldots + w_kx_{i+k-1}
\]
Systolic Computation Example: Convolution

- $y_1 = w_1x_1 + w_2x_2 + w_3x_3$
- $y_2 = w_1x_2 + w_2x_3 + w_3x_4$
- $y_3 = w_1x_3 + w_2x_4 + w_3x_5$

Figure 8. Design W1: systolic convolution array (a) and cell (b) where $w_i$’s stay and $x_i$’s and $y_i$’s move systolically in opposite directions.
Worthwhile to implement adder and multiplier separately to allow overlapping of add/mul executions
More Programmability

- Each PE in a systolic array
  - Can store multiple “weights”
  - Weights can be selected on the fly
  - Eases implementation of, e.g., adaptive filtering

- Taken further
  - Each PE can have its own data and instruction memory
  - Data memory \(\rightarrow\) to store partial/temporary results, constants
  - Leads to *stream processing, pipeline parallelism*
    - More generally, *staged execution*
Pipeline Parallelism

Figure 1. (a) The code of a loop, (b) Each iteration is split into 3 pipeline stages: A, B, and C. Iteration i comprises Ai, Bi, Ci. (c) Sequential execution of 4 iterations. (d) Parallel execution of 6 iterations using pipeline parallelism on a three-core machine. Each stage executes on one core.
File Compression Example

Figure 3. File compression algorithm executed using pipeline parallelism
Systolic Array

Advantages
- Makes multiple uses of each data item $\rightarrow$ reduced need for fetching/refetching
- High concurrency
- Regular design (both data and control flow)

Disadvantages
- Not good at exploiting irregular parallelism
- Relatively special purpose $\rightarrow$ need software, programmer support to be a general purpose model
The WARP Computer

- HT Kung, CMU, 1984-1988

- Linear array of 10 cells, each cell a 10 Mflop programmable processor

- Attached to a general purpose host machine

- HLL and optimizing compiler to program the systolic array

- Used extensively to accelerate vision and robotics tasks


The WARP Computer

Figure 1: Warp system overview
The WARP Computer

Figure 2: Warp cell data path
Systolic Arrays vs. SIMD

- Food for thought...
Some More Recommended Readings

- Recommended: