# Modern Processor Architectures (A compiler writer's perspective)

L25: Modern Compiler Design

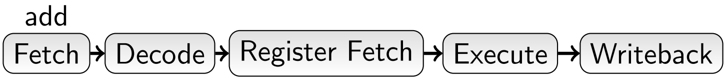
## The 1960s - 1970s

- Instructions took multiple cycles
- Only one instruction in flight at once
- Optimisation meant minimising the number of instructions executed
- Sometimes replacing expensive general-purpose instructions with specialised sequences of cheaper ones

## The 1980s

- CPUs became pipelined
- Optimisation meant minimising pipeline stalls
- Dependency ordering such that results were not needed in the next instruction
- Computed branches became very expensive when not correctly predicted

## Stall Example

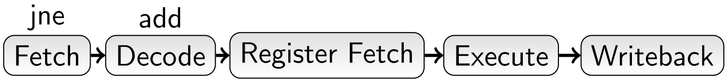


```
(int i=100 ; i!=0 ; i--)

...
```

```
start:
    ...
    add r1, r1, -1
    jne r1, 0, start
```

## Stall Example

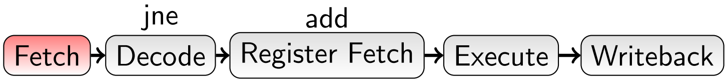


```
(int i=100 ; i!=0 ; i--)


...
```

```
start:
    ...
    add r1, r1, -1
    jne r1, 0, start
```

## Stall Example

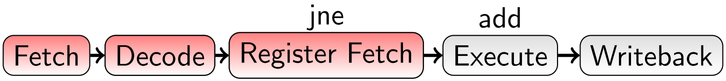


```
(int i=100 ; i!=0 ; i--)


...
```

```
start:
    ...
    add r1, r1, -1
    jne r1, 0, start
```

## Stall Example

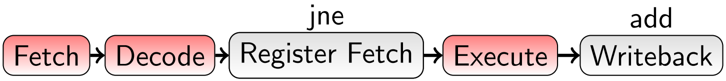


```
(int i=100 ; i!=0 ; i--)


...
```

```
start:
    ...
    add r1, r1, -1
    jne r1, 0, start
```

## Stall Example

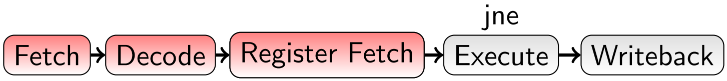


```
(int i=100 ; i!=0 ; i--)


...
```

```
start:
    ...
    add r1, r1, -1
    jne r1, 0, start
```

## Stall Example

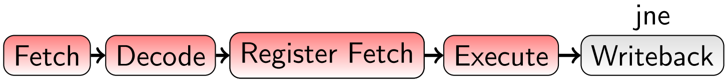


```
(int i=100 ; i!=0 ; i--)


...
```

```
start:
    ...
    add r1, r1, -1
    jne r1, 0, start
```

## Stall Example



```
(int i=100 ; i!=0 ; i--)


...
```

```
start:
    ...
    add r1, r1, -1
    jne r1, 0, start
```

## Fixing the Stall

```
(int i=100 ; i!=0 ; i--)

...
```

```
start:
    add r1, r1, -1
    ...
    jne r1, 0, start
```

## Fixing the Stall

```
(int i=100 ; i!=0 ; i--)

...
```

```
start:
    add r1, r1, -1
    ...
    jne r1, 0, start
```

Is this a good solution?

## Note about efficiency

- In-order pipelines give very good performance per Watt at low power
- Probably not going away any time soon (see ARM Cortex A7, A53)
- Compiler optimisations can make a big difference!

## The Early 1990s

- CPUs became much faster than memory
- Caches hid some latency
- Optimisation meant maximising locality of reference, prefetching
- Sometimes, recalculating results is faster than fetching from memory
- Note: Large caches consume a lot of power, but fetching a value from memory can cost the same as several hundred ALU ops

## The Mid 1990s

- CPUs became superscalar
    - Independent instructions executed in parallel
- CPUs became out-of-order
    - Reordered instructions to reduce dependencies
- Optimisation meant structuring code for highest-possible ILP
- Loop unrolling no longer such a big win

## Superscalar CPU Pipeline Example: Sandy Bridge

Can dispatch up to six instructions at once, via 6 pipelines:

1. ALU, VecMul, Shuffle, FpDiv, FpMul, Blend
2. ALU, VecAdd, Shuffle, FpAdd
3. Load / Store address
4. Load / Store address
5. Load / Store data
6. ALU, Branch, Shuffle, VecLogic, Blend

## Branch Predictors

- Achieve 95+% accuracy on modern CPUs
- No cost when branch is correctly predicted
- Long and wide pipelines mean very expensive for the remaining 5%!

With 140 instructions in-flight on the Pentium 4 and branches roughly every 7 cycles, what's the probability of filling the pipeline?

## Branch Predictors

- Achieve 95+% accuracy on modern CPUs
- No cost when branch is correctly predicted
- Long and wide pipelines mean very expensive for the remaining 5%!

With 140 instructions in-flight on the Pentium 4 and branches roughly every 7 cycles, what's the probability of filling the pipeline?
Only 35%!

## Branch Predictors

- Achieve 95+% accuracy on modern CPUs
- No cost when branch is correctly predicted
- Long and wide pipelines mean very expensive for the remaining 5%!

With 140 instructions in-flight on the Pentium 4 and branches roughly every 7 cycles, what's the probability of filling the pipeline?
Only 35%!
Only 12% with a 90% hit rate!

## The Late 1990s

- SIMD became mainstream
- Factor of 2-4$\times$ speedup when used correctly
- Optimisation meant ensuring data parallelism
- Loop unrolling starts winning again, as it exposes later optimisation opportunities

## The Early 2000s

- (Homogeneous) Multicore became mainstream
- Power efficiency became important
- Parallelism provides both better throughput and lower power
- Optimisation meant exploiting fine-grained parallelism

## The Late 2000s

- Programmable GPUs became mainstream
- Hardware optimised for stream processing in parallel
- Very fast for massively-parallel floating point operations
- Cost of moving data between CPU and CPU is high
- Optimisation meant offloading operations to the GPU

## The 2010s

- Modern processors come with multiple CPU and GPU cores
- All cores behind the same memory interface, cost of moving data between them is low
- Increasingly contain specialised accelerators
- Often contain general-purpose (programmable) cores for specialised workload types (e.g. DSPs)
- Optimisation is hard.
- Lots of jobs for compiler writers!

## Common Programming Models

- Sequential (can we automatically detect parallelism)?
- Explicit message passing (e.g. MPI, Erlang)
- Annotation-driven parallelism (e.g. OpenMP)
- Explicit task-based parallelism (e.g. libdispatch)
- Explicit threading (e.g. pthreads, shared-everything concurrency)

## Parallelising Loop Iterations

- Same techniques as for SIMD
- Looser constraints: data can be unaligned, flow control can be independent
- Tighter constraints: loop iterations must be completely independent
- (Usually) more overhead for creating threads than using SIMD lanes

## Communication and Synchronisation Costs

- Consider OpenMP's parallel-for

## Communication and Synchronisation Costs

- Consider OpenMP's parallel-for
- Spawn a new thread for each iteration?
- Spawn one thread per core, split loop iterations between them?
- Spawn one thread per core, have each one start a loop iteration and check the current loop induction variable before doing the next one?
- Spawn one thread per core, pass batches of loop iterations to each one?
- Something else?

## HELIX: Parallelising Sequential Segments in Loops

- Loop iterations each run a sequence of (potentially expensive) steps
- Run each step on a separate core
- Each core runs the same number of iterations as the original loop
- Use explicit synchronisation to detect barriers

## Execution Models for GPUs

- GPUs have no standardised public instruction set
- Code shipped as source or some portable IR
- Compiled at install or load time
- Loaded to the device to run

## SPIR

- Standard Portable Intermediate Representation
- Khronos Group standard, related to OpenCL
- Subsets of LLVM IR (one for 32-bit, one for 64-bit)
- Backed by ARM, AMD, Intel (everyone except nVidia)
- OpenCL programming model extensions as intrinsics
- Design by committee nightmare, no performance portability

## SPIR-V

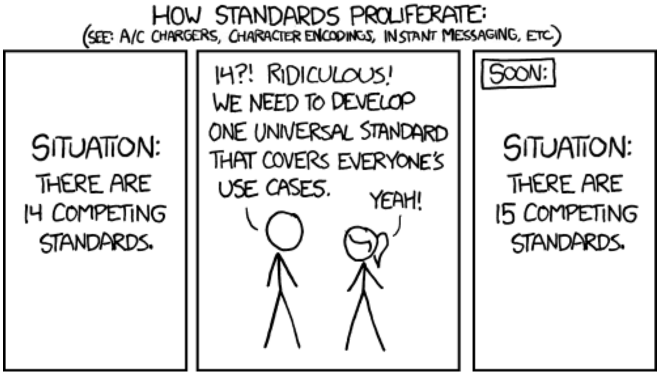
- Standard Portable Intermediate Representation (for Vulkan)
- Khronos Group standard, related to Vulkan
- Independent encoding, easy to map to/from LLVM IR
- Backed by ARM, AMD, Intel (everyone except nVidia)
- Intended as a compilation target for GLSL, OpenCL C, others

## PTX

- Parallel Thread eXecution
- IR created by nVidia
- Semantics much closer to nVidia GPUs

## HSAIL

- Heterogeneous Systems Architecture Intermediate Language
- Cross-vendor effort under the HSA umbrella
- More general than PTX (e.g. allows function pointers)


HOW STANDARDS PROLIFERATE:
(SEE: A/C CHARGERS, CHARACTER ENCODINGS, INSTANT MESSAGING, ETC)
SITUATION: THERE ARE 14 COMPETING STANDARDS.
14?! RIDICULOUS! WE NEED TO DEVELOP ONE UNIVERSAL STANDARD THAT COVERS EVERYONE'S USE CASES.
YEAH!
SOON:
SITUATION: THERE ARE 15 COMPETING STANDARDS.

## Single Instruction Multiple Thread (SIMT)

- SIMD with independent register sets, varying-sized vectors
- Program counter (PC) shared across threads
- All threads perform the same operation, but on different data
- Diverging threads get their own PC
- Only one PC used at a time
- Throughput halves for each divergent branch until only one thread is running

## Thread Groups

- GPU programs run the same code (kernel) on every element in an input set
- Threads in a group can communicate via barriers and other synchronisation primitives
- Thread groups are independent

## GPU Memory Model

- Per-thread memory (explicitly managed, equivalent to CPU cache)
- Shared memory between thread groups (equivalent to CPU shared L3 cache)
- Global memory (read-write, cache coherent)
- Texture memory (read-only or write-only, non-coherent)

## Costs for GPU Use

- Setup context (MMU mappings on GPU, command queue). Typically once per application.
- Copying data across the bus is very expensive, may involve bounce buffers
- Newer GPUs share a memory controller with the CPU (might not share an address space, setting IOMMU mappings can be expensive)
- Calling into the OS kernel to send messages (userspace command submission helps here)
- Synchronisation (cache coherency) between CPU and GPU

## Thought Experiment: memcpy(), memset()

- GPUs and DSPs are fast stream processors
- Ideal for things like memcpy(), memset()
- What bottlenecks prevent offloading all memset() / memcpy() calls to a coprocessor?
- How could they be fixed?

## Autoparallelisation vs Autovectorisation

- Autovectorisation is a special case of autoparallelisation
- Requires dependency, alias analysis between sections
- GPU SIMT processors are suited to the same sorts of workloads as SIMD coprocessors
- (Currently) only sensible when working on large data or very expensive calculations

## Loop offloading

- Identify all inputs and outputs
- Copy all inputs to the GPU
- Run the loop as a GPU kernel
- Copy all outputs back to main memory
- Why can this go wrong?

## Loop offloading

- Identify all inputs and outputs
- Copy all inputs to the GPU
- Run the loop as a GPU kernel
- Copy all outputs back to main memory
- Why can this go wrong?
- What happens if you have other threads accessing memory?
- Shared everything is hard to reason about

## Avoiding Divergent Flow Control: If Conversion

- Two threads taking different paths must be executed sequentially
- Execute both branches
- Conditionally select the result
- Also useful on superscalar architectures - reduces branch predictor pressure
- Early GPUs did this in hardware

## OpenCL on the CPU

- Can SIMD emulate SIMT?
- Hardware is similar, SIMT is slightly more flexible
- Sometimes, OpenCL code runs faster on the CPU if data is small
- Non-diverging flow is trivial
- Diverging flow requires special handling

## Diverging Flow

- Explicit masking for if conversion
- Each possible path is executed
- Results are conditionally selected
- Significant slowdown for widely diverging code
- Stores, loads-after-stores require special handling

## OpenCL Synchronisation Model

- Explicit barriers block until all threads in a thread group have arrived.
- Atomic operations (can implement spinlocks)
    - Why would spinlocks on a GPU be slow?

## OpenCL Synchronisation Model

- Explicit barriers block until all threads in a thread group have arrived.
- Atomic operations (can implement spinlocks)
  - Why would spinlocks on a GPU be slow?
  - Branches are slow, non-streaming memory-access is expensive...
  - Random access to workgroup-shared memory is cheaper than texture memory

## Barriers and SIMD

- Non-diverging flow, barrier is a no-op
- Diverging flow requires rendezvous
- Pure SIMD implementation (single core), barrier is where start of a basic block after taking both sides of a branch
- No real synchronisation required

## OpenCL with SIMD on multicore CPUs

- Barriers require real synchronisation
- Can be a simple pthread barrier
- Alternatively, different cores can run independent thread groups

Questions?