Advanced Systems Lab

Spring 2022
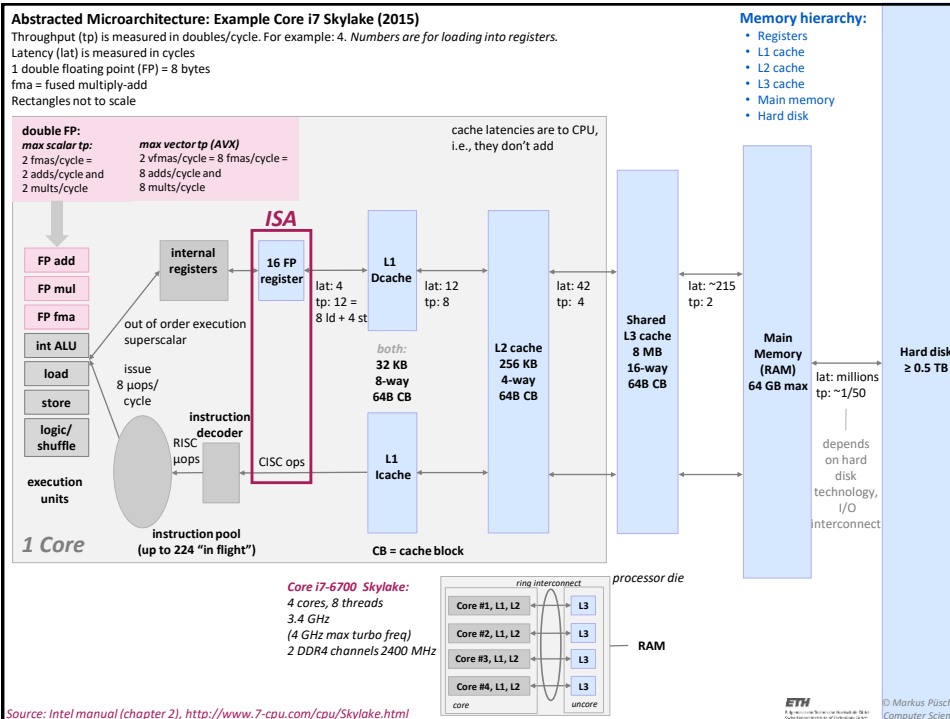*Lecture:* Optimization for Instruction-Level Parallelism

Instructor: Markus Püschel, Ce Zhang
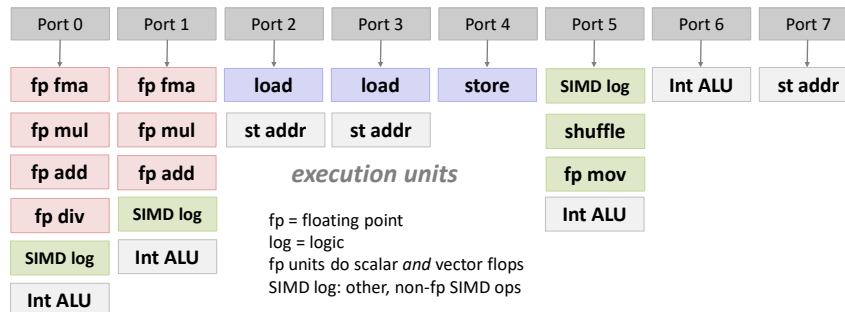
TA: Joao Rivera, several more

ETH
Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Abstracted Microarchitecture: Example Core i7 Skylake (2015)
Throughput (tp) is measured in doubles/cycle. For example: 4. *Numbers are for loading into registers.*
Latency (lat) is measured in cycles
1 double floating point (FP) = 8 bytes
fma = fused multiply-add
Rectangles not to scale

Memory hierarchy:
- Registers
- L1 cache
- L2 cache
- L3 cache
- Main memory
- Hard disk

double FP:
max scalar tp:
2 fmas/cycle =
2 adds/cycle and
2 mults/cycle

max vector tp (AVX)
2 vfmas/cycle = 8 fmas/cycle =
8 adds/cycle and
8 mults/cycle

cache latencies are to CPU,
i.e., they don't add

ISA

FP add
FP mul
FP fma
int ALU
load
store
logic/shuffle

execution units

internal registers

out of order execution
superscalar

issue
8 μops/cycle

instruction decoder

RISC μops

CISC ops

**instruction pool
(up to 224 "in flight")**

16 FP register

lat: 4
tp: 12 =
8 ld + 4 st

L1 Dcache

lat: 12
tp: 8

both:
**32 KB
8-way
64B CB**

L1 Icache

CB = cache block

**L2 cache
256 KB
4-way
64B CB**

lat: 42
tp: 4

**Shared
L3 cache
8 MB
16-way
64B CB**

lat: ~215
tp: 2

**Main
Memory
(RAM)
64 GB max**

lat: millions
tp: ~1/50

**Hard disk
≥ 0.5 TB**

depends
on hard
disk
technology,
I/O
interconnect

1 Core

Core i7-6700 Skylake:
4 cores, 8 threads
3.4 GHz
(4 GHz max turbo freq)
2 DDR4 channels 2400 MHz

ring interconnect

processor die

Core #1, L1, L2 — L3
Core #2, L1, L2 — L3
Core #3, L1, L2 — L3
Core #4, L1, L2 — L3

RAM

core   uncore

Source: Intel manual (chapter 2), http://www.7-cpu.com/cpu/Skylake.html

© Markus Püschel
Computer Science

Mapping of execution units to ports

Port 0	Port 1	Port 2	Port 3	Port 4	Port 5	Port 6	Port 7
fp fma	fp fma	load	load	store	SIMD log	Int ALU	st addr
fp mul	fp mul	st addr	st addr		shuffle		
fp add	fp add				fp mov		
fp div	SIMD log				Int ALU		
SIMD log	Int ALU						
Int ALU							

execution units

fp = floating point
log = logic
fp units do scalar *and* vector flops
SIMD log: other, non-fp SIMD ops

Execution Unit (fp)	Latency [cycles]	Throughput [ops/cycle]	Gap [cycles/issue]
fma	4	2	0.5
mul	4	2	0.5
add	4	2	0.5
div (scalar)	14	1/4	4
div (4-way)	14	1/8	8

- Every port can issue one instruction/cycle
- Gap = 1/throughput
- ***Intel calls gap the throughput!***
- Same exec units for scalar and vector flops
- Same latency/throughput for scalar (one double) and AVX vector (four doubles) flops, except for div

Source: Intel manual (Table C-8. 256-bit AVX Instructions, Table 2-1. Dispatch Port and Execution Stacks of the Skylake Microarchitecture, Figure 2-1. CPU Core Pipeline Functionality of the Skylake Microarchitecture)

How To Make Code Faster?

It depends!

Memory bound: Reduce memory traffic
- *Reduce cache misses*
- *Compress data*

Compute bound: Keep floating point units busy
- *Reduce cache misses, register spills*
- *Instruction level parallelism (ILP)*
- *Vectorization*

Next: Optimizing for ILP (an example)

Chapter 5 in Computer Systems: A Programmer's Perspective, 2nd edition, Randal E. Bryant and David R. O'Hallaron, Addison Wesley 2010
Part of these slides are adapted from the course associated with this book

4

Superscalar Processor

Definition: A superscalar processor can issue and execute *multiple instructions in one cycle*. The instructions are retrieved from a sequential instruction stream and are usually scheduled dynamically.

Benefit: Superscalar processors can take advantage of *instruction level parallelism (ILP)* that many programs have.

Deep pipelines also require ILP (explained today).

Most CPUs since about 1998 are superscalar

Intel: since Pentium Pro

Simple embedded processors are usually not superscalar

5

ILP

Code

```
t2 = t0 + t1
t5 = t4 * t3
t6 = t2 + t5
```

Dependencies

t6 = t2 + t5

t2 = t0 + t1   t5 = t4 * t3
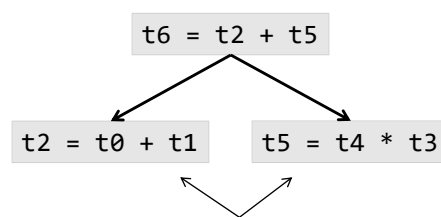
can be executed in parallel
and in any order

6

Hard Bounds: Coffee Lake and Haswell

Coffee Lake

	latency	1/tp = gap
FP Add	4	0.5
FP Mul	4	0.5
Int Add	1	0.5
Int Mul	3	1

blackboard

Haswell

	latency	1/tp = gap
FP Add	3	1
FP Mul	5	0.5
Int Add	1	0.5
Int Mul	3	1

More precisely (Int Add):
- tp = 2 if one operand is from mem
- tp = 4 if all operands in register

Only the ALU at Port 1 does Int Mults

7

Coffee Lake

	latency	1/tp = gap
FP Mul	4	0.5

ops
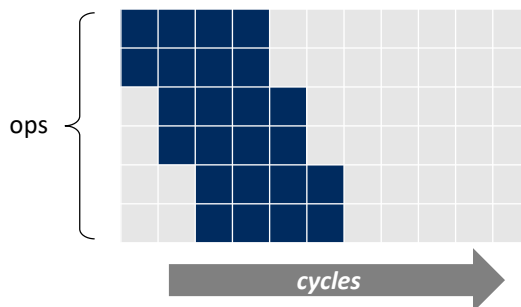
Throughput tp = 2/cycle

Gap = 1/tp = 1/2 cycles/issue

cycles

How many cycles at least for n mults?
- *ceil(n/2) (considering only throughput)*
- *ceil(n/2) + 3 (considering latency and throughput)*

8

© *Markus Püschel*
Computer Science   ETH
Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Advanced Systems Lab
Spring 2022

Example Computation: Reduction

```
void reduce(vec_ptr v, data_t *dest)
{
  int i;
  int length = vec_length(v);
  data_t *d  = get_vec_start(v);
  data_t t   = IDENT;
  for (i = 0; i < length; i++)
    t = t OP d[i];
  *dest = t;
}
```

d[0] OP d[1] OP d[2] OP … OP d[length-1]

data_t: double or int

OP: + or *
IDENT: 0 or 1

Runtime of Reduce (Coffee Lake)

```
void reduce(vec_ptr v, data_t *dest)
{
  int i;
  int length = vec_length(v);
  data_t *d  = get_vec_start(v);
  data_t t   = IDENT;
  for (i = 0; i < length; i++)
    t = t OP d[i];
  *dest = t;
}
```

Measured cycles per OP

Method	Int (add/mult)		Float (add/mult)	
reduce	1.29	2.95	3.91	3.91
bound	0.5	1.0	0.5	0.5

Questions:
- *Explain red row*
- *Explain gray row*

This and all following measurements: gcc -O3 -mavx2 -fno-tree-vectorize

Reduce = Serial Computation (here: *)
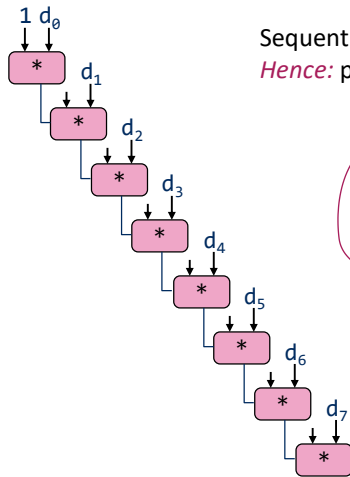
Sequential dependence = no ILP!
Hence: performance determined by latency of OP!

Method	Int (add/mult)		Float (add/mult)	
reduce	1.29	2.95	3.91	3.91
bound	0.5	1.0	0.5	0.5

Loop Unrolling

```
void unroll2(vec_ptr v, data_t *dest)
{
    int length = vec_length(v);
    int limit  = length-1;
    data_t *d  = get_vec_start(v);
    data_t x   = IDENT;
    int i;
    /* Combine 2 elements at a time */
    for (i = 0; i < limit; i += 2)
        x = (x OP d[i]) OP d[i+1];
    /* Finish any remaining elements */
    for (; i < length; i++)
        x = x OP d[i];
    *dest = x;
}
```

Perform 2x more useful work per iteration

How does the runtime change?

Effect of Loop Unrolling

Method	Int (add/mult)		Float (add/mult)	
combine4	1.29	2.95	3.91	3.91
unroll2	1.0	2.93	3.90	3.91
bound	0.5	1.0	0.5	0.5

Helps integer sum a bit

Others don't improve. *Why?*
- *Still sequential dependency*

```
x = (x OP d[i]) OP d[i+1];
```

Loop Unrolling with Separate Accumulators

```
void unroll2_sa(vec_ptr v, data_t *dest)
{
    int length = vec_length(v);
    int limit  = length-1;
    data_t *d  = get_vec_start(v);
    data_t x0  = IDENT;
    data_t x1  = IDENT;
    int i;
    /* Combine 2 elements at a time */
    for (i = 0; i < limit; i += 2) {
        x0 = x0 OP d[i];
        x1 = x1 OP d[i+1];
    }
    /* Finish any remaining elements */
    for (; i < length; i++)
        x0 = x0 OP d[i];
    *dest = x0 OP x1;
}
```

Effect on runtime?

Can this change the result of the computation?

Floating point: yes!

Effect of Separate Accumulators

Method	Int (add/mult)		Float (add/mult)	
combine4	1.29	2.95	3.91	3.91
unroll2	1.0	2.93	3.90	3.91
unroll2-sa	0.8	1.49	1.96	1.97
bound	0.5	1.0	0.5	0.5

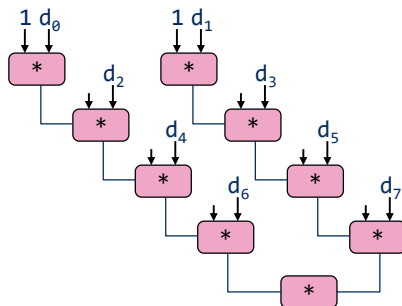Almost exact 2x speedup (over unroll2) for Int *, FP +, FP *

- *Breaks sequential dependency*

```
x0 = x0 OP d[i];
x1 = x1 OP d[i+1];
```

Separate Accumulators

```
x0 = x0 OP d[i];
x1 = x1 OP d[i+1];
```


What changed:
- Two independent "streams" of operations

Overall Performance
- N elements, D cycles latency/op
- Should be (N/2+1)*D cycles: *cycles per OP ≈ D/2*

What Now?

Unrolling & Accumulating

Idea
- *Use K accumulators*
- *Increase K until best performance reached*
- *Need to unroll by L, K divides L*

Limitations
- *Diminishing returns:*
  Cannot go beyond throughput limitations of execution units
- *Some overhead for short lengths: Finish off iterations sequentially*

17

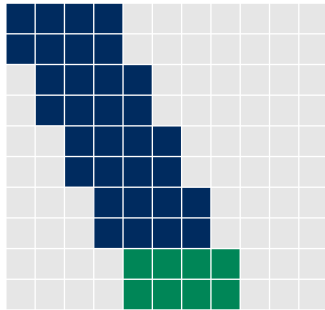
Unrolling & Accumulating: FP *

Coffee Lake: FP multiplication
- *Gap = cycles/issue = 0.5*
- *Latency = 4*

FP64 *	Unrolling Factor L							
K	1	2	3	4	6	8	10	12
1	3.91	3.91		3.91		3.91		
2		1.97		1.97		1.96		
3			1.32		1.32			
4				1.00		1.0		
6					0.70			0.70
8						0.56		
10							0.54	
12								0.54

Accumulators

Why 8?

18

Why 8?

Those have to be independent

Based on this insight: K = #accumulators = ceil(latency / cycles per issue)
                                          = ceil(latency * throughput)
Here: K = ceil(4 / 0.5) = ceil(4 * 2) = 8

Unrolling & Accumulating: FP +

Coffee Lake: FP addition

- *Gap = cycles/issue = 0.5*
- *Latency = 4*

FP64 +	Unrolling Factor L							
K	1	2	3	4	6	8	10	12
1	3.91	3.90		3.90		3.90		
2		1.96		1.96		1.96		
3			1.32		1.32			
4				1.00		1.00		
6					0.70			0.70
8						0.56		
10							0.54	
12								0.54

Accumulators

Unrolling & Accumulating: Int *

Coffee Lake: Int multiplication

- *Gap = cycles/issue = 1*
- *Latency = 3*

Int *	Unrolling Factor L							
K	1	2	3	4	6	8	10	12
1	2.94	2.94		2.93		2.93		
2		1.49		1.49		1.49		
3			1.32		1.32			
4				1.01		1.01		
6					1.01			1.00
8						1.01		
10							1.01	
12								1.01

Accumulators

Unrolling & Accumulating: Int +

Coffee Lake: Int multiplication

- *Gap = cycles/issue = 0.5*
- *Latency = 1*

Int +	Unrolling Factor L							
K	1	2	3	4	6	8	10	12
1	1.29	1.00		1.00		1.00		
2		0.80		0.58		0.52		
3			0.69		0.52			
4				0.57		0.52		
6					0.52			0.52
8						0.52		
10							0.52	
12								0.52

Accumulators

Interesting question: what exactly happens here?

ETH
Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Advanced Systems Lab
Spring 2022

Coffee Lake vs. Haswell: FP +

FP64 +	Unrolling Factor L							
K	1	2	3	4	6	8	10	12
1	3.91	3.90		3.90		3.90		
2		1.96		1.96		1.96		
3			1.32		1.32			
4				1.00		1.00		
6					0.70			0.70
8						0.56		
10							0.54	
12								0.54

Coffee Lake:
Latency = 4
Gap = 0.5

FP64 +	Unrolling Factor L							
K	1	2	3	4	6	8	10	12
1	2.95	2.95		2.95		2.95		
2		1.49		1.49		1.49		
3			1.00		1.00			
4				1.01		1.01		
6					1.01			1.01
8						1.00		
10							1.01	
12								1.01

Haswell:
Latency = 3
Gap = 1

Says something about porting processor-tuned code

23

Apple M1: FP +

FP64 +	Unrolling Factor L							
K	1	2	3	4	6	8	10	12
1	2.87	2.83		2.75		2.75		
2		1.38		1.39		1.26		
3			0.93		0.93			
4				0.71		0.71		
6					0.49			0.49
8						0.38		
10							0.32	
12								0.31

Firestorm:
Latency = 3
Gap = 0.25

24

Summary (ILP)

Deep pipelines and multiple ports require ILP for good throughput

ILP may have to be made explicit in program

Potential blockers for compilers
- *Reassociation changes result (floating point)*
- *Too many choices, no good way of deciding*

Unrolling
- *By itself does usually nothing (branch prediction works usually well)*
- *But may be needed to enable additional transformations
(here: reassociation)*

How to program this example?
- *Solution 1: program generator generates alternatives and picks best*
- *Solution 2: use model based on latency and throughput*

25