How to Write Fast Numerical Code

Spring 2017

Lecture: Optimization for Instruction-Level Parallelism

Instructor: Markus Püschel

TA: Alen Stojanov, Geoerg Ofenbeck, Gagandeep Singh

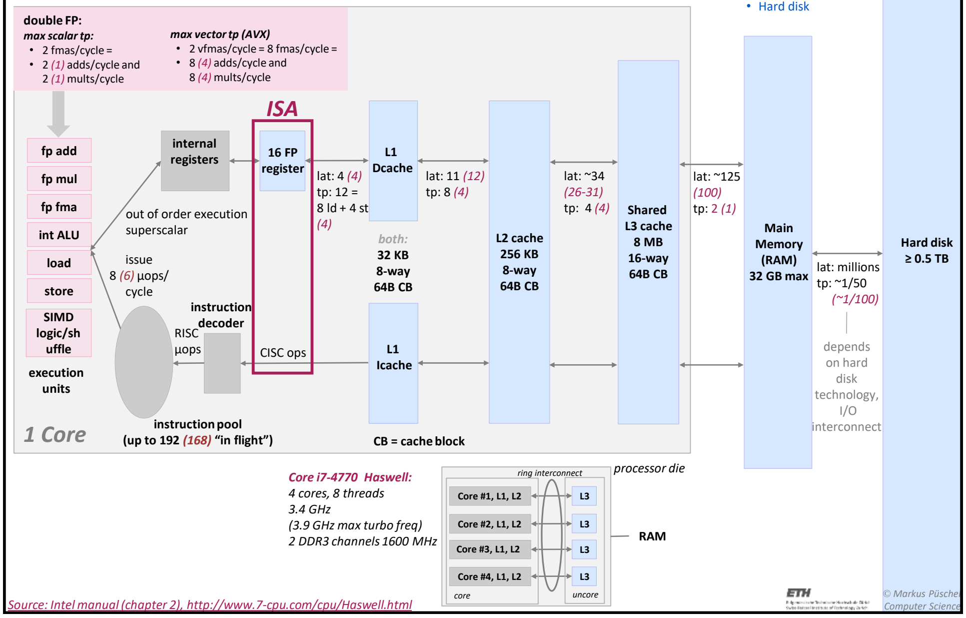
ETH
Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Abstracted Microarchitecture: Example Core i7 Haswell (2013) and *Sandybridge (2011)*

Throughput (tp) is measured in doubles/cycle. For example: 4 *(2)*
Latency (lat) is measured in cycles
1 double floating point (FP) = 8 bytes
fma = fused multiply-add
Rectangles not to scale

Haswell *Sandy Bridge*

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



Core i7-4770 Haswell:
4 cores, 8 threads
3.4 GHz
(3.9 GHz max turbo freq)
2 DDR3 channels 1600 MHz

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

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		
div	fp add				fp mov		
SIMD log	SIMD log				Int ALU		
Int ALU	Int ALU						

execution units

Execution Unit	Latency (cycles)	Throughput (ops/cycle)	Gap (cycles/issue)
fp fma	5	2	0.5
fp mul	5	2	0.5
fp add	3	1	1
div	25-35	1/27	27

Every port can issue one instruction/cycle

Gap = 1/throughput

Intel calls gap the throughput!

Same units for scalar and vector flops

Source: Intel manual (Table C-8. 256-bit AVX Instructions, Table 2-6. Dispatch Port and Execution Stacks of the Haswell Microarchitecture, Figure 2-2. CPU Core Pipeline Functionality of the Haswell Microarchitecture),

---

How To Make Code Faster?

- **It depends!**
- **Memory bound: Reduce memory traffic**
  - Reduce cache misses, register spills
  - 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

Superscalar Processor

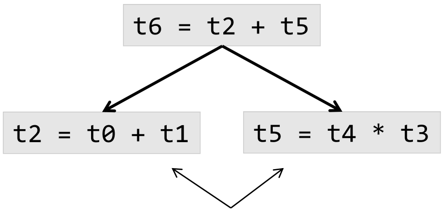
- **Definition:** A superscalar processor can issue and execute ***multiple instructions in one cycle***. The instructions are retrieved from a sequential instruction stream and are usually scheduled dynamically.
- Benefit: Superscalar processors can take advantage of ***instruction level parallelism (ILP)*** that many programs have
- Most CPUs since about 1998 are superscalar
- Intel: since Pentium Pro
- Simple embedded processors are usually not superscalar

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

Hard Bounds: Pentium 4 vs. Core 2

- **Pentium 4 (Nocona)**

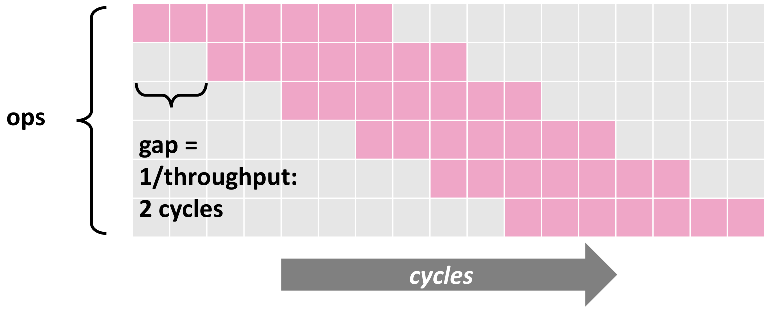
Instruction	*Latency*	*1/Throughput = Cycles/Issue*
Load / Store	5	1
Integer Multiply	10	1
Integer/Long Divide	36/106	36/106
Single/Double FP Multiply	**7**	**2**
Single/Double FP Add	**5**	**2**
Single/Double FP Divide	32/46	32/46

put on black-board

- **Core 2**

Instruction	*Latency*	*Cycles/Issue*
Load / Store	5	1
Integer Multiply	3	1
Integer/Long Divide	18/50	18/50
Single/Double FP Multiply	**4/5**	**1**
Single/Double FP Add	**3**	**1**
Single/Double FP Divide	18/32	18/32

Single/Double FP Multiply	**7**	**2**



Hard Bounds (cont'd)

- **How many cycles at least if**
  - Function requires n float adds?
  - Function requires n int mults?

Example Computation (on Pentium 4)

```
void combine4(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: float or double or int
```

```
OP:    + or *
IDENT: 0 or 1
```

Runtime of Combine4 (Pentium 4)

- **Use cycles/OP**

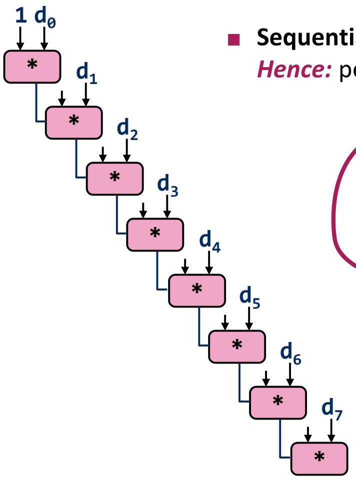
```
void combine4(vec_ptr v, data_t *dest)
{
  int i;
  int length = vec_length(v);
  data_t *d  = get_vec_start(v);
  data_t t   = IDENT;
  for (i = 0; i < length; i++)
    t = t OP d[i];
  *dest = t;
}
```

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

Cycles per OP

Method	Int (add/mult)		Float (add/mult)	
combine4	2.2	10.0	5.0	7.0
bound	1.0	1.0	2.0	2.0

Combine4 = Serial Computation (OP = *)



- **Sequential dependence = no ILP!**
Hence: performance determined by latency of OP!

Cycles per element (or per OP)

Method	Int (add/mult)		Float (add/mult)	
combine4	2.2	10.0	5.0	7.0
bound	1.0	1.0	2.0	2.0

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**

Effect of Loop Unrolling

Method	Int (add/mult)		Float (add/mult)	
combine4	2.2	10.0	5.0	7.0
unroll2	1.5	10.0	5.0	7.0
bound	1.0	1.0	2.0	2.0

- **Helps integer sum**
- **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;
}
```

- **Can this change the result of the computation?**
- ***Floating point: yes!***

Effect of Separate Accumulators

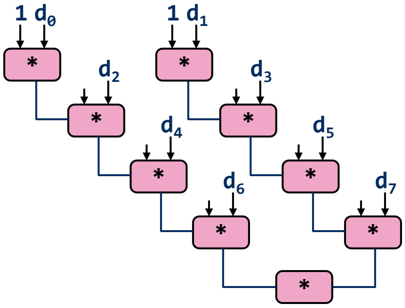
Method	Int (add/mult)		Float (add/mult)	
combine4	2.2	10.0	5.0	7.0
unroll2	1.5	10.0	5.0	7.0
unroll2-sa	1.50	5.0	2.5	3.5
bound	1.0	1.0	2.0	2.0

- **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
  - Large overhead for short lengths: Finish off iterations sequentially

Unrolling & Accumulating: Intel FP *

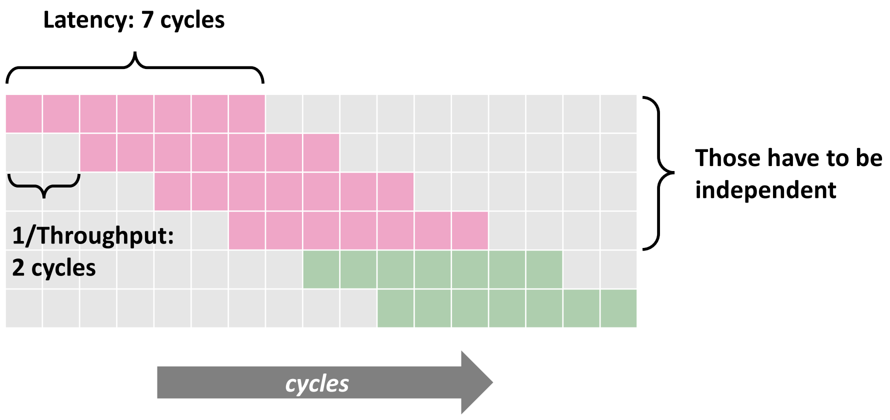
- **Case**
  - Pentium 4
  - FP Multiplication
  - Theoretical Limit: 2.00

Accumulators

FP *	Unrolling Factor L							
K	1	2	3	4	6	8	10	12
1	7.00	7.00		7.01		7.00		
2		3.50		3.50		3.50		
3			2.34					
4				2.01		2.00		
6					2.00			2.01
8						2.01		
10							2.00	
12								2.00

Why 4?

Why 4?



Latency: 7 cycles

1/Throughput: 2 cycles

Those have to be independent

cycles

Based on this insight: **K = #accumulators = ceil(latency/cycles per issue)**

Unrolling & Accumulating: Intel FP +

- **Case**
  - Pentium 4
  - FP Addition
  - Theoretical Limit: 2.00

FP +	Unrolling Factor L							
K	1	2	3	4	6	8	10	12
1	5.0	5.0		5.0		5.0		
2		2.5		2.5		2.5		
3			2.0					
4				2.0		2.00		
6					2.0			2.0
8						2.0		
10							2.0	
12								2.0

Unrolling & Accumulating: Intel Int *

- **Case**
  - Pentium 4
  - Integer Multiplication
  - Theoretical Limit: 1.00

Int *	Unrolling Factor L							
K	1	2	3	4	6	8	10	12
1	10.0	10.0		10.0		10.0		
2		5.0		5.0		5.0		
3			3.3					
4				2.5		2.5		
6					1.67			1.67
8						1.25		
10							1.1	
12								1.14

Unrolling & Accumulating: Intel Int +

- **Case**
  - Pentium 4
  - Integer addition
  - Theoretical Limit: 1.00

Int +	Unrolling Factor L							
K	1	2	3	4	6	8	10	12
1	2.2	1.5		1.1		1.0		
2		1.5		1.1		1.0		
3			1.34					
4				1.1		1.03		
6					1.0			1.0
8						1.03		
10							1.04	
12								1.1

FP *	Unrolling Factor L							
K	1	2	3	4	6	8	10	12
1	7.0	7.0		7.0		7.0		
2		3.5		3.5		3.5		
3			2.34					
4				2.0		2.0		
6					2.0			2.0
8						2.0		
10							2.0	
12								2.0

Pentium 4

FP *	Unrolling Factor L							
K	1	2	3	4	6	8	10	12
1	4.0	4.0		4.0		4.0		
2		2.0		2.0		2.0		
3			1.34					
4				1.0		1.0		
6					1.0			1.0
8						1.0		
10							1.0	
12								1.0

Core 2
*FP * is fully pipelined*

Summary (ILP)

- **Instruction level parallelism may have to be made explicit in program**
- **Potential blockers for compilers**
  - Reassociation changes result (FP)
  - Too many choices, no good way of deciding
- **Unrolling**
  - By itself does often 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