

How to Write Fast Numerical Code

Spring 2013

Lecture: Architecture/Microarchitecture and Intel Core

Instructor: Markus Püschel

TA: Daniele Spampinato & Alen Stojanov

ETH

Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Technicalities

- **Research project:**
 - Let us know once you have a partner
 - If you have a project idea, talk to me (break, after Wed class, email)
 - Deadline: March 7th
- **Finding partner:** fastcode-forum@lists.inf.ethz.ch
 - Recipients: TA + all students that have no partner yet

Today

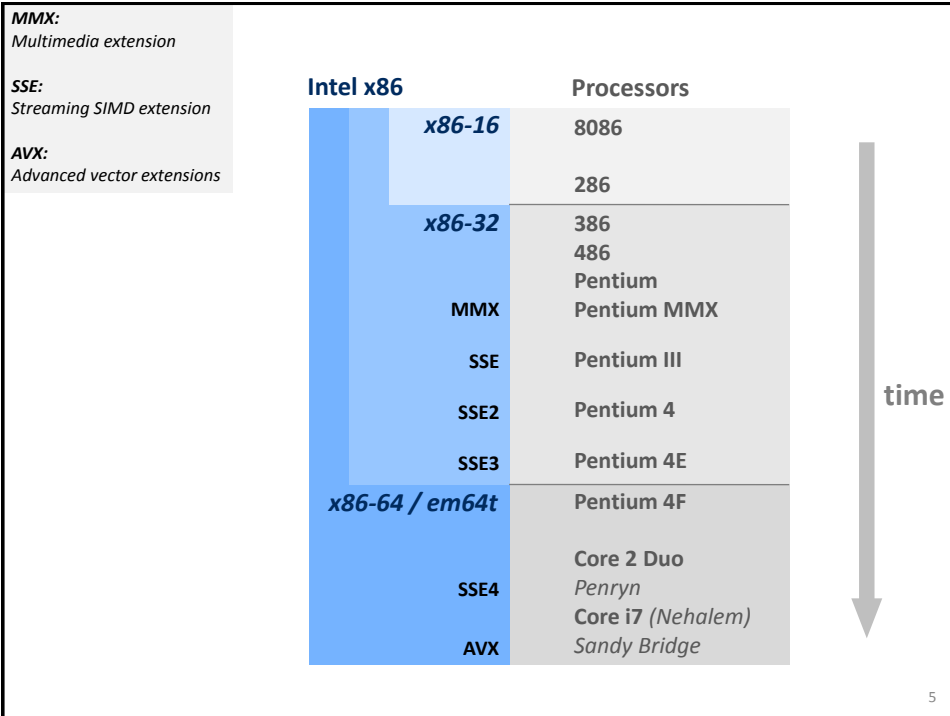
- Architecture/Microarchitecture
- In detail: Core 2/Core i7
- Crucial microarchitectural parameters
- Peak performance
- Operational intensity

3

Definitions

- **Architecture** (*also instruction set architecture = ISA*): The parts of a processor design that one needs to understand to write assembly code
- **Examples:** instruction set specification, registers
- **Counterexamples:** cache sizes and core frequency
- **Example ISAs**
 - x86
 - ia
 - MIPS
 - POWER
 - SPARC
 - ARM

4



ISA SIMD (Single Instruction Multiple Data) Vector Extensions

■ What is it?

- Extension of the ISA. Data types and instructions for the parallel computation on short (length 2-8) vectors of integers or floats

$$\begin{array}{|c|c|c|c|} \hline \color{yellow}{\square} & \color{red}{\square} & \color{green}{\square} & \color{blue}{\square} \\ \hline \end{array} + \begin{array}{|c|c|c|c|} \hline \color{yellow}{\square} & \color{red}{\square} & \color{green}{\square} & \color{blue}{\square} \\ \hline \end{array} \quad \color{black}{\times} \quad \begin{array}{|c|c|c|c|} \hline \color{yellow}{\square} & \color{red}{\square} & \color{green}{\square} & \color{blue}{\square} \\ \hline \end{array} \quad \color{black}{4\text{-way}}$$

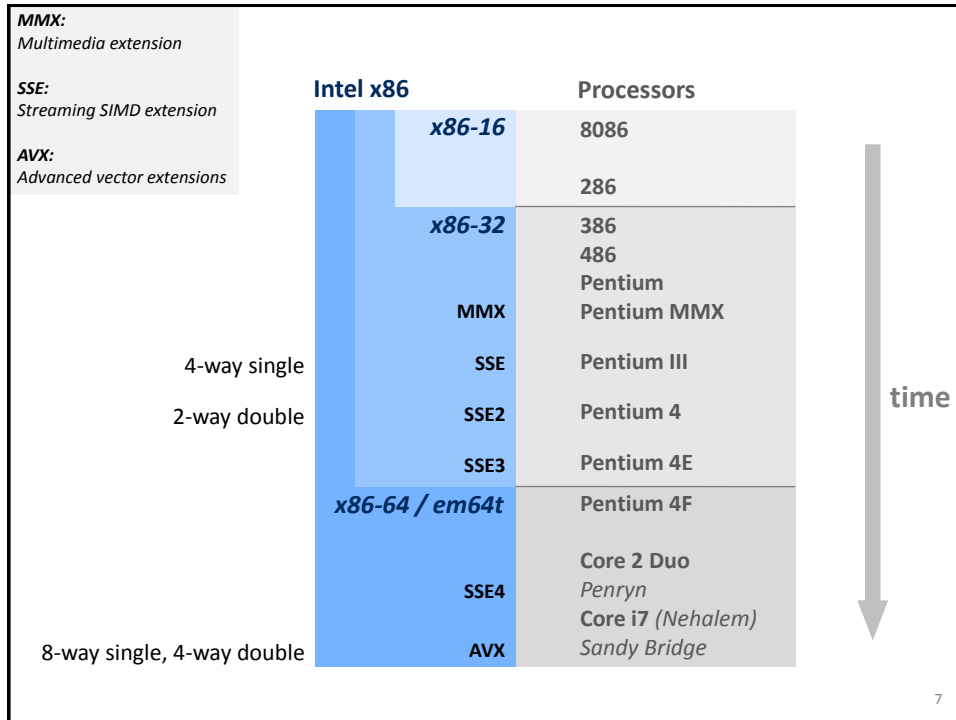
- Names: MMX, SSE, SSE2, ...

■ Why do they exist?

- Useful:** Many applications have the necessary fine-grain parallelism
Then: speedup by a factor close to vector length
- Doable:** Chip designers have enough transistors to play with

■ We will have an extra lecture on vector instructions

- What are the problems?
- How to use them efficiently

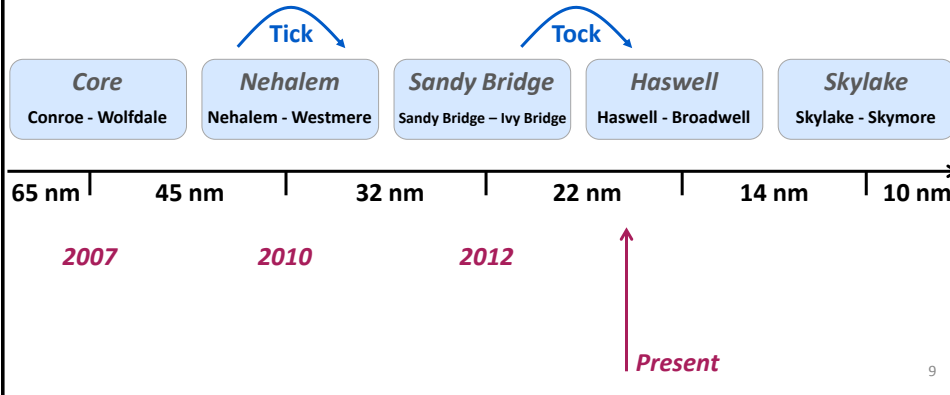


Definitions

- **Microarchitecture:** Implementation of the architecture
- **Examples:** caches, cache structure, CPU frequency, details of the virtual memory system
- **Examples**
 - Intel processors ([Wikipedia](#))
 - AMD processors ([Wikipedia](#))

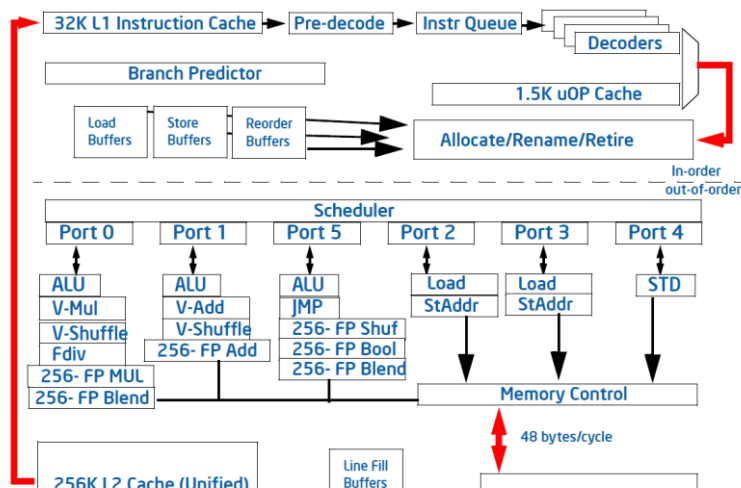
Intel's Tick-Tock Model

- **Tock:** New microarchitecture
- **Tick:** Shrink of process technology
- **Example: Core and successors**
Shown: Intel's microarchitecture code names (server/mobile may be different)



9

Microarchitecture: The View of the Computer Architect

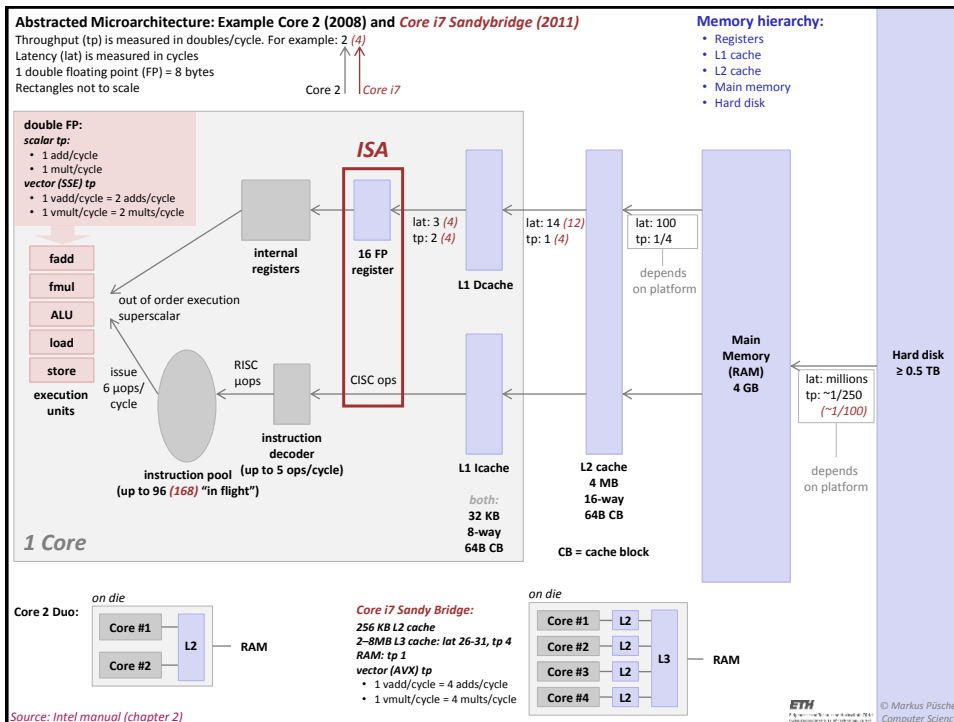


we take the software developer's view ...

Source: *Intel Architectures Optimization Reference Manual*

10

■ Distribute microarchitecture abstraction



Runtime Bounds (Cycles) on Core 2

```
/* dot product computation; x, y are vectors of doubles of length n */  
double t = 0;  
for (i = 0; i < n; i++)  
    t = t + x[i]*y[i];
```

*maximal achievable percentage
of (vector) peak performance*

- Number flops? $2n$
- Runtime bound no vector ops: n
- Runtime bound vector ops: $n/2$
- Runtime bound data in L1: n **50**
- Runtime bound data in L2: $2n$ **25**
- Runtime bound data in main memory: $8n$ **6.25**

*Runtime dominated by data movement:
Memory-bound*

Runtime Bounds (Cycles) on Core 2

```
/* matrix multiplication; A, B, C are n x n matrices of doubles */  
for (i = 0; i < n; i++)  
    for (j = 0; j < n; j++)  
        for (k = 0; k < n; k++)  
            C[i*n+j] += A[i*n+k]*B[k*n+j];
```

- Number flops? $2n^3$
- Runtime bound no vector ops: n^3
- Runtime bound vector ops: $n^3/2$
- Runtime bound data in L1: $3/2 n^2$
- Runtime bound data in L2: $3n^2$
- Runtime bound data in main memory: $12n^2$

*Runtime dominated by data operations (except very small n):
Compute-bound*

Operational Intensity

- Definition: Given a program P, assume cold (empty) cache

$$\text{Operational intensity: } I(n) = \frac{W(n)}{Q(n)}$$

#flops (input size n) ←
#bytes transferred cache ↔ memory (for input size n) ←

15

Operational Intensity (Cold Cache)

```
/* dot product computation; x, y are vectors of doubles of length n */  
double t = 0;  
for (i = 0; i < n; i++)  
    t = t + x[i]*y[i];
```

- Operational intensity:
 - Flops: $W(n) = 2n$
 - Memory/cache transfers (doubles): $\geq 2n$ (just from the reads)
 - Reads (bytes): $Q(n) \geq 16n$
 - Operational intensity: $I(n) = W(n)/Q(n) \leq 1/8$

16

Operational Intensity (Cold Cache)

```
/* matrix multiplication; A, B, C are n x n matrices of doubles */
for (i = 0; i < n; i++)
  for (j = 0; j < n; j++)
    for (k = 0; k < n; k++)
      C[i*n+j] += A[i*n+k]*B[k*n+j];
```

■ Operational intensity:

- Flops: $W(n) = 2n^3$
- Memory/cache transfers (doubles): $\geq 3n^2$ (just from the reads)
- Reads (bytes): $Q(n) \geq 24n^2$
- Operational intensity: $I(n) = W(n)/Q(n) \leq 1/12 n$

17

Operational Intensity

- Definition: Given a program P, assume cold (empty) cache

$$\text{Operational intensity: } I(n) = \frac{W(n)}{Q(n)}$$

#flops (input size n) ←
#bytes transferred cache ↔ memory (for input size n) ←

- Examples: Determine asymptotic bounds on $I(n)$

- Vector sum: $y = x + y$ $O(1)$
- Matrix-vector product: $y = Ax$ $O(1)$
- Fast Fourier transform $O(\log(n))$
- Matrix-matrix product: $C = AB + C$ $O(n)$

- Note: In the last two cases, the tightest possible bound depends on the cache size m ; more later

18

Compute/Memory Bound

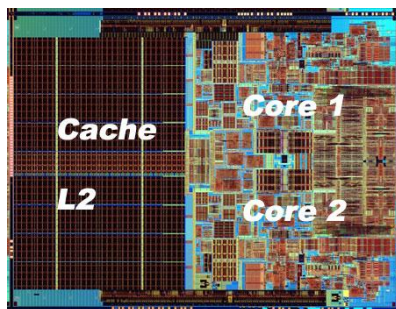
- A function/piece of code is:
 - *Compute bound* if it has high operational intensity
 - *Memory bound* if it has low operational intensity
- A more exact definition depends on the given platform
- More details later: Roofline model

19

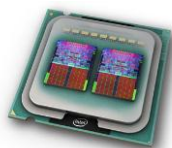
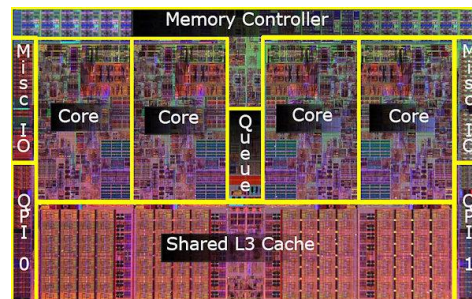
Core Processor

Pictures: Intel

Core 2 Duo



Core i7



2 x Core 2 Duo
packaged

[Detailed information about Core processors](#)

20

Floating Point Peak Performance?

Table 2-15. Issue Ports of Intel Core Microarchitecture and Enhanced Intel Core Microarchitecture

Executable operations	Latency, Throughput		Comment ¹
	Signature = 06_0FH	Signature = 06_17H	
Integer ALU	1, 1	1, 1	Includes 64-bit mode integer MUL; Issue port 0; Writeback port 0;
Integer SIMD ALU	1, 1	1, 1	
FP/SIMD/SSE2 Move and Logic	1, 1	1, 1	
Single-precision (SP) FP MUL	4, 1	4, 1	Issue port 0; Writeback port 0
Double-precision FP MUL	5, 1	5, 1	Issue port 0; Writeback port 0 FP shuffle does not handle QW shuffle.
FP MUL (X87)	5, 2	5, 2	
FP Shuffle	1, 1	1, 1	
Integer ALU	1, 1	1, 1	Excludes 64-bit mode integer MUL; Issue port 1; Writeback port 1;
Integer SIMD ALU	1, 1	1, 1	
FP/SIMD/SSE2 Move and Logic	1, 1	1, 1	
FP ADD	3, 1	3, 1	Issue port 1; Writeback port 1;
QW Shuffle	1, 1 ^c	1, 1 ³	
Integer loads	3, 1	3, 1	Issue port 2; Writeback port 2;
FP loads	4, 1	4, 1	
Store address ⁴	3, 1	3, 1	Issue port 3;
Store data ⁵			Issue Port 4;
Integer ALU	1, 1	1, 1	Issue port 5; Writeback port 5;
Integer SIMD ALU	1, 1	1, 1	
FP/SIMD/SSE2 Move and Logic	1, 1	1, 1	
QW shuffles	1, 1 ²	1, 1 ³	Issue port 5; Writeback port 5;
128-bit Shuffle/Pack/Unpack	2-4, 2-4 ⁶	1-3, 1 ⁷	

Two different processor lines

1 or 2 mults/cycle?

1 add/cycle

The Two Floating Points

```
float ipf (float x[], float y[], int n) {
    int i;
    float result = 0.0;

    for (i = 0; i < n; i++)
        result += x[i]*y[i];
    return result;
}
```

standard compilation (SSE)

```
ipf:
    xorps    %xmm1, %xmm1
    xorl    %ecx, %ecx
    jmp     .L8
.L10:
    movslq  %ecx, %rax
    incl   %ecx
    movss  (%rsi,%rax,4), %xmm0
    mulss  (%rdi,%rax,4), %xmm0
    addss  %xmm0, %xmm1
.L8:
    cmpl   %edx, %ecx
    jl    .L10
    movaps %xmm1, %xmm0
    ret
```

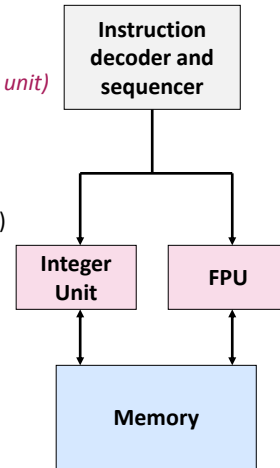
compilation for x87

```
...
cmpl %edx,%eax
jge .L3
.L5:
    flds  (%ebx,%eax,4)
    fmuls (%ecx,%eax,4)
    faddp
    incl %eax
    cmpl %edx,%eax
    jl .L5
.L3:
    movl -4(%ebp),%ebx
    movl %ebp, %esp
    popl %ebp
    ret
```

The Other Floating Point (x87)

History

- 8086: first computer to implement IEEE FP
(separate 8087 math coprocessor with floating point unit)
- Logically stack based
- 486: merged FPU and Integer Unit onto one chip
- Once SSE came out, it was used for floating point
- x87 is default on x86-32 (since SSE is not guaranteed)
- Became obsolete with x86-64



23

MMX:
Multimedia extension

SSE:
Streaming SIMD extension

AVX:
Advanced vector extensions

Intel x86	Processors
x86-16	8086
	286
x86-32 x87	386
	486
	Pentium
	Pentium MMX
MMX	
SSE	Pentium III
SSE2	Pentium 4
SSE3	Pentium 4E
x86-64 / em64t	Pentium 4F
	Core 2 Duo
	Penryn
	Core i7 (Nehalem) Sandy Bridge

time

24

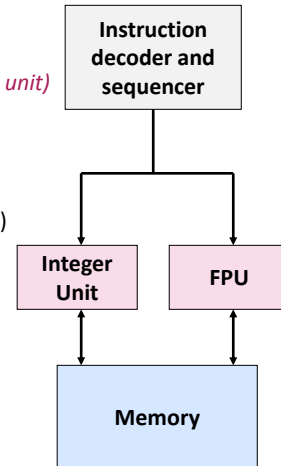
The Other Floating Point (x87)

History

- 8086: first computer to implement IEEE FP
(separate 8087 math coprocessor with floating point unit)
- Logically stack based
- 486: merged FPU and Integer Unit onto one chip
- Once SSE came out, it was used for floating point
- x87 is default on x86-32 (since SSE is not guaranteed)
- Became obsolete with x86-64

Floating Point Formats

- single precision (C **float**): 32 bits
- double precision (C **double**): 64 bits
- extended precision (C **long double**): 80 bits



25

x87 FPU Instructions and Register Stack

Sample instructions:

- flds (load single precision)
- fmls (mult single precision)
- faddp (add and pop)

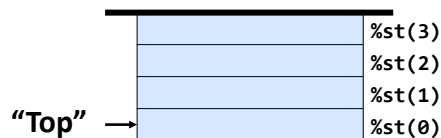
8 registers %st(0) - %st(7)

Logically form stack

Top: %st(0)

Bottom disappears (drops out) after too many pushes

Obsolete



26

FP Code Example (x87)

- Inner product of two vectors
Single precision arithmetic

```
float ipf (float x[],
          float y[],
          int n) {
    int i;
    float result = 0.0;
    for (i = 0; i < n; i++)
        result += x[i]*y[i];
    return result;
}
```

```
pushl %ebp                # setup
movl %esp,%ebp
pushl %ebx

movl 8(%ebp),%ebx        # %ebx=&x
movl 12(%ebp),%ecx       # %ecx=&y
movl 16(%ebp),%edx       # %edx=n
fldz                    # push +0.0
xorl %eax,%eax          # i=0
cmpl %edx,%eax          # if i>=n done
jge .L3

.L5:
flds (%ebx,%eax,4)      # push x[i]
fmuls (%ecx,%eax,4)     # st(0)*=y[i]
faddp                   # st(1)+=st(0); pop
incl %eax               # i++
cmpl %edx,%eax         # if i<n repeat
jl .L5

.L3:
movl -4(%ebp),%ebx      # finish
movl %ebp, %esp
popl %ebp
ret                    # st(0) = result
```

27

Core: Floating Point Peak Performance

Single-precision (SP) FP MUL	4, 1	4, 1	Issue port 0; Writeback port 0
Double-precision FP MUL	5, 1	5, 1	
FP MUL (X87)	5, 2	5, 2	Issue port 0; Writeback port 0
FP Shuffle	1, 1	1, 1	FP shuffle does not handle QW shuffle.
DIV/SQRT			

**SSE based FP
x87 FP**

- Scalar:**
 - 1 add and 1 mult / cycle
 - Assume 3 GHz:
6 Gflop/s scalar peak performance on one core
- Vector double precision (SSE2)**
 - 1 vadd and 1 vmult / cycle (2-way): 4 flops/cycle
 - Assume 3 GHz:
12 Gflop/s peak performance on one core
- Vector single precision (SSE)**
 - 1 vadd and 1 vmult / cycle (4-way): 8 flops/cycle
 - Assume 3 GHz:
24 Gflop/s peak performance on one core

28

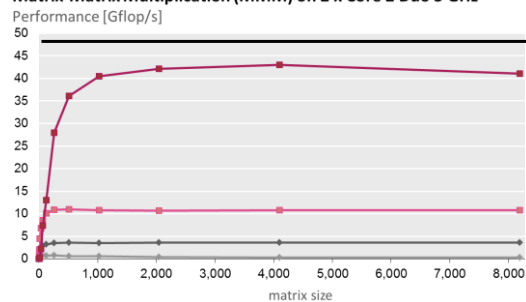
Core: Floating Point Peak Performance

- Overall peak on 3 GHz Core 2 and Core i3/i5/i7 Nehalem: (2 cores, SSE)
 - Double precision: **24 Gflop/s**
 - Single precision: **48 Gflop/s**
- Overall peak on 3 GHz Core i3/i5/i7 Sandy Bridge: (4 cores, AVX)
 - Double precision: **96 Gflop/s**
 - Single precision: **192 Gflop/s**

29

Example: Peak Performance

Matrix-Matrix Multiplication (MMM) on 2 x Core 2 Duo 3 GHz



*Peak performance
of this computer*

30

Summary

- **Architecture vs. microarchitecture**
- **To optimize code one needs to understand a suitable abstraction of the microarchitecture**
- **Operational intensity:**
 - High = compute bound = runtime dominated by data operations
 - Low = memory bound = runtime dominated by data movement