Computer Structure

Multi-Threading

Lecturer:
Roni Kupershtok

Slides prepared by Lihu Rappoport
Parallelism

- **Instruction Level Parallelism (ILP)**
  - Independent instructions within a program can run in parallel
    - A given program, with a given input data has a given parallelism
  - An OOO uses ILP to run independent instructions in parallel

- **Data Level Parallelism (DLP)**
  - Same operation executed on multiple data elements
    - E.g., adding two vectors: add operation on each one of the vector’s elements

- **Thread Level Parallelism (TLP)**
  - A threaded application: an application written to use multiple threads
    - Multi-threaded code is hard to write, to debug, and to validate
  - Different applications running simultaneously
  - Operating system services running in parallel to applications
Flynn’s Taxonomy of Parallel Machines

- Classifies machines according to two parameters
  - How many Instruction streams? single/many
  - How many Data streams? single/many

<table>
<thead>
<tr>
<th>Single Instruction stream</th>
<th>Single Data stream</th>
<th>Multiple Data streams</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>SISD</strong></td>
<td>SISD</td>
<td>SIMD</td>
</tr>
<tr>
<td>A uniprocessor</td>
<td></td>
<td>Each execution element works on its own data, but all execute the same instructions in lockstep. E.g. a vector processor.</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Multiple Instruction streams</th>
<th>Multiple Data streams</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>MISD</strong></td>
<td>MISD</td>
</tr>
<tr>
<td>Not Used</td>
<td>MIMD</td>
</tr>
<tr>
<td></td>
<td>Each processor executes its own instructions and operates on its own data. Typical multi-core processor (made using a bunch of “normal” cores).</td>
</tr>
</tbody>
</table>
Multi-Processor Memory

- **Centralized shared memory (SMP – Symmetric Multiprocessor)**
  - Uniform Memory Access (UMA)
    - All memory locations have similar latencies
    - Data sharing through memory reads/writes
  - Memory bandwidth becomes bottleneck

- **Distributed Shared-Memory (DSM)**
  - Also called NUMA (non-uniform memory access)
  - All processors can address all memory locations, like in SMP
  - Variable latency: local access faster than remote access

- **Message-Passing (multi-computers / clusters)**
  - A processor can directly address only its local memory
    - Most accesses local \(\Rightarrow\) less memory contention
  - Communicate with other processors by sending/receiving messages
    - Puts extra burden on SW for handing messages vs. shared memory systems, in which communication is done by memory read/write
  - Scales to many processors
CMP – Chip Multi Core

• Simple SMP on the same chip
  – Resources can be shared between CPUs, e.g., shared L3 caches

• Cheaper than multi-socket SMP
  – Interface logic integrated on-chip
  – Fewer total chips, single CPU socket,
  – Single interface to main memory

• Less power than multi-socket SMP
  – On-die communication more power-efficient than chip-to-chip communication

• Performance
  – On-chip communication is faster
  – Efficiency
  – Potentially better use of hardware resources than improving performance of a single-threaded CPU
Single-Core Multi-Threading

- **Multi-threading**: a single core executes multiple threads
  - When one thread is stalled (on a cache miss, branch misprediction, or a long dependency), the other thread gets to use the free resources

- **Switch-on-event multithreading (SOEMT)**
  - A single thread exists in the machine at a given moment
  - Switch threads on a long latency event, such as last level cache misses
  - Works well for server applications that have many cache misses
  - Does not cover for branch mispredictions and single-thread low ILP

- **Simultaneous multi-threading (SMT)**
  - Multiple threads execute on a single core simultaneously
    - Interleaved in the machine pipeline
  - When one thread is stalled / slowed down, other thread gets more resources
  - Makes the most effective use of processor resources
Server Systems

- Server systems typically support SMP, CMP and SMT together
  - Multiple sockets (processors) per motherboard
  - Multiple cores per die
  - Multiple thread per core
  - E.g., Intel® Xeon® E7-8890 V4
    - SMP: Up to 8 processors per board
    - CMP: 24 cores per processor
    - SMT: 2 threads per core

- Multiple blades per rack
  - Blades connect using a network, e.g., InfiniBand
Supercomputers

• Supercomputers or HPC (High Performance Computers) clusters
  – A group of servers connected with a dedicated high-speed network
  – Top500 ranks the 500 most powerful non-distributed computer systems
    ✔ Based on HPL – high-performance LINPACK benchmark
      (a software library for performing numerical linear algebra)

• Trinity system (Cray XC40) at Los Alamos National Laboratory

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Cores:</td>
<td>979,968</td>
</tr>
<tr>
<td>Processor:</td>
<td>Intel Xeon Phi 7250 68C 1.4GHz</td>
</tr>
<tr>
<td>Linpack Performance</td>
<td>14,137 TFlop/s</td>
</tr>
<tr>
<td>Theoretical Peak</td>
<td>43,902 TFlop/s</td>
</tr>
<tr>
<td>Power:</td>
<td>3,843kW</td>
</tr>
</tbody>
</table>
Core Resources Utilization

- The IPC of a program is limited due to
  - Instruction dependencies (ILP – Instruction Level Parallelism)
  - Cache misses
  - Branch misprediction

- Memory bound vs. compute bound
  - Server apps are typically memory bound, with low IPC

- Actual average IPC for a 4-wide machine is <2
  - Utilization < 50%
  - Even taking into account 10%-20% more instructions are fetched and executed than retired due to speculative execution

- Increasing single thread performance becomes harder
  - Less power efficient, less area efficient
Simultaneous multi-threading (SMT)

- **Two logical processors within one physical core**
  - Sharing most execution/cache resources using SMT
  - Look like two processors to the SW (OS and apps)

- **Each logical processor executes a software thread**
  - Threads execute simultaneously on one physical core

- **Each logical processor maintains its own arch. state**
  - Complete set of architectural registers
    - General-purpose registers, Control registers, Machine state registers, Debug registers
  - Instruction pointers

- **Each logical processor has its own interrupt controller**
  - Handles the interrupts sent to the specific logical processor
SMT: A High-level View of the Pipeline

- Each pipe-stage is occupied by one of the threads

- When one thread is stalled, the other thread can continue to make progress
  - Pipe-stages: either guaranteed not to stall, or flushed if prevent other thread from making progress
  - Buffers: partitioned / replicated / shared with guaranteed entries
Physical Resource Sharing Schemes

- **Replicated Resources**: e.g. CR3
  - Each thread has its own resource

- **Partitioned Resources**: e.g. iTLB
  - Resource is partitioned in MT mode, and combined in ST mode

- **Competitively-shared resource**: e.g. RS entries
  - Both threads compete for the entire resource

- **HT unaware resources**: e.g. execution unit
  - Both threads use the resource

- **Resource sharing may cause performance issues**
  - E.g., cache thrashing: each one of the threads may fit into the cache, but both together don’t ⇒ high miss rate
SMT: Thread Select Points

- Pipeline arbitration points select the thread gets to use a resource in a given cycle
  - “ping-pong” between the threads on a cycle-by-cycle basis
    - If one thread is stalled ⇒ give its turn to the other thread
  - Achieves fairness between the threads
    - But still gives full bandwidth to one thread in case the other thread does not require the given resource

- The RS is not a thread arbitration point
  - Dispatches µops to EUs based on readiness and age (alloc time)
    - Regardless of which thread they belong to
Front End Resource Sharing

- Branch prediction resources are mostly shared
- The Instruction Pointer is replicated
- The iTLB is partitioned
- The I$ is thread-unaware (thus competitively-shared)
- The pre-decode logic and the instruction decoder logic are used by one thread at a time
- The μop-Cache is partitioned
- The μop-Queue is partitioned
Back End Resource Sharing

- **Out-Of-Order Execution**
  - Register renaming tables are replicated
  - The reservation stations are competitively-shared
  - Execution units are SMT unaware
  - The ROB is partitioned
  - Retirement logic alternates between threads

- **Memory**
  - The Load buffers and store buffers are partitioned
  - The DTLB and STLB are competitively-shared
  - The cache hierarchy and fill buffers are competitively-shared
Single-task And Multi-task Modes

- **MT-mode (Multi-task mode)**
  - Two active threads, with some resources partitioned as described earlier

- **ST-mode (Single-task mode)**
  - ST0 / ST1 – only thread 0 / 1 is active
  - Resources that are partitioned in MT-mode are re-combined to give the single active logical processor use of all of the resources

- **Moving the processor from between modes**

```
+------------------+        +-------------------+        +-------------------+
|                  |        |                  |        |                  |
|                  |        |                  |        |                  |
|  ST0              |        |  ST1              |        |  MT               |
|                  |        |                  |        |                  |
|                  |        |                  |        |                  |
| Thread 0 executes HALT |    | Thread 1 executes HALT |    | Thread 0 executes HALT |
|                   |        |                  |        |                   |
|                   |        |                  |        |                   |
|                  |        |                  |        |                   |
| Interrupt        |  Low  | Thread 1 executes HALT |    | Thread 0 executes HALT |
|                  | Power |                  |        |                   |
```

From the Optimization Manual

Computer Structure 2018 – Intel® Core™ µArch
Thread Optimization

The OS should implement two optimizations:

- Use HALT if only one logical processor is active
  - Allows the processor to transition to either the ST0 or ST1 mode
  - Otherwise the OS would execute on the idle logical processor a sequence of instructions that repeatedly checks for work to do
  - This so-called “idle loop” can consume significant execution resources that could otherwise be used by the other active logical processor

- On a multi-processor system
  - OS views logical processors similar to physical processors
    - But can still differentiate and prefer to schedule a thread on a new physical processor rather than on a 2nd logical processors in the same physical processor
  - Schedule threads to logical processors on different physical processors before scheduling multiple threads to the same physical processor
  - Allows SW threads to use different physical resources when possible
Computer Structure

Intel® Core™ μArch

Lecturer:
Roni Kupershtok

Slides prepared by Lihu Rappoport
## Tick/Tock Development Model

<table>
<thead>
<tr>
<th>Generation</th>
<th>Year</th>
<th>Architecture</th>
<th>Microarchitecture</th>
<th>Node Size</th>
</tr>
</thead>
<tbody>
<tr>
<td>1st Generation</td>
<td>2006</td>
<td>Merom</td>
<td>Intel® Core2™ Duo</td>
<td>65nm</td>
</tr>
<tr>
<td>1st Generation</td>
<td>2007</td>
<td>Penryn</td>
<td>Intel® Core2™ Duo</td>
<td>45nm</td>
</tr>
<tr>
<td>1st Generation</td>
<td>2008</td>
<td>Nehalem</td>
<td>Intel® Core™</td>
<td>32nm</td>
</tr>
<tr>
<td>2nd Generation</td>
<td>2009</td>
<td>Westmere</td>
<td>Intel® Core™</td>
<td></td>
</tr>
<tr>
<td>3rd Generation</td>
<td>2010</td>
<td>Sandy Bridge</td>
<td>Intel® Core™</td>
<td></td>
</tr>
<tr>
<td>4th Generation</td>
<td>2011</td>
<td>Ivy Bridge</td>
<td>Intel® Core™</td>
<td></td>
</tr>
<tr>
<td>5th Generation</td>
<td>2012</td>
<td>Haswell</td>
<td>Intel® Core™</td>
<td></td>
</tr>
<tr>
<td>6th Generation</td>
<td>2013</td>
<td>Brodwell</td>
<td>Intel® Core™</td>
<td></td>
</tr>
<tr>
<td>6th Generation</td>
<td>2014</td>
<td>Skylake</td>
<td>Intel® Core™</td>
<td></td>
</tr>
</tbody>
</table>

Timeline:

- **2006**: 1st Generation Intel® Core2™ Duo (Merom) and Intel® Core2™ Duo (Penryn)
- **2007**: 1st Generation Intel® Core™ (Nehalem)
- **2009**: 2nd Generation Intel® Core™ (Westmere)
- **2010**: 3rd Generation Intel® Core™ (Sandy Bridge)
- **2011**: 4th Generation Intel® Core™ (Ivy Bridge)
- **2012**: 5th Generation Intel® Core™ (Haswell)
- **2013**: 6th Generation Intel® Core™ (Brodwell)
- **2014**: 6th Generation Intel® Core™ (Skylake)

Node Sizes:

- 65nm
- 45nm
- 32nm
- 22nm
- 14nm
6th Generation Intel Core™ – Skylake

- 14nm Process Technology
- Quad core die, with Intel HD Graphics 530
SkyLake Core

32KB L1 I$ → Pre decode → Inst Q → Decoders → μop Cache

Branch Prediction Unit

Allocate/Rename/Retire/ME/ZI

Scheduler

Port 0: ALU, Shift, JMP 2, MUL
Port 1: ALU, LEA
Port 5: ALU, LEA
Port 6: ALU, Shift, JMP 1

Port 4: Store Data
Port 2: Load/STA
Port 3: Load/STA
Port 7: STA

Memory Control

256KB L2$

Load Buffers → 32KB L1 D$

Load Data 2

Load Data 3

μop Cache

Port 4

Port 2

Port 3

Port 7

μop Queue

In order

OOO

Foil taken from IDF 2012
**The Front End**

**Legacy decode pipeline**

- **32KB L1 I-Cache** → **Pre decode** → **Inst Q** → **Decoders** → **μop-Queue**

- **Branch Prediction Unit**
  - Predicts 32B/cyc – $2 \times$ wider than the legacy decode pipeline
    - Enables taken branches to be predicted with no penalty
    - Initiate TLB and cache misses speculatively to hide miss latency

- **Legacy decode pipeline: fetch and decode instructions**
  - Instruction cache: 32KB 8 ways, fetches aligned 16 bytes/cycle
    - Typical programs average ~4 bytes per instruction
  - Pre-decode unit: instruction length and attributes decode
  - Decoders: decode instructions into μops

- **μop-Queue** – caches μops coming from the decoder
Micro Operations (μops)

- Each X86 inst. is broken into one or more RISC μ-ops
  - Each μop is (relatively) simple

- Simple instructions translate to a few μops
  - μop count for typical instructions
    - Reg-Reg ALU/Mov inst: 1 μop
    - Mem-Reg Mov (load) 1 μop
    - Mem-Reg ALU (load + op) 2 μops
    - Reg-Mem Mov (store) 2 μops (st addr, st data)
    - Reg-Mem ALU (ld + op + st) 4 μops
**Macro-Fusion**

- The IQ sends up to 5 inst. / cycle to the decoders
- **Merge two adjacent instructions into a single \( \mu \text{op} \)**
  - A macro-fused instruction executes with a single dispatch
    - Reduces latency and frees execution resources
    - Increased decode, rename and retire bandwidth
    - Power savings from representing more work in fewer bits
- **The 1\textsuperscript{st} instruction modifies flags**
  - CMP, TEST, ADD, SUB, AND, INC, DEC
- **The 2\textsuperscript{nd} instruction pair is a conditional branch**
- **These pairs are common in many types of applications**
Stack Pointer Tracker

- **PUSH, POP, CALL, RET implicitly update ESP**
  - Add or sub an offset, which would require a dedicated µop
  - The Stack Pointer Tracker performs implicit ESP updates

\[
\begin{align*}
\text{PUSH EAX} & : & \text{ESP} & \rightarrow \text{ESP} - 4 & \Delta = \Delta - 4 & \Delta = 0 \\
\text{STORE [ESP], EAX} & : & \text{STORE [ESP-4], EAX} & & \Delta = - 4 & \Delta = - 8 \\
\text{PUSH EBX} & : & \text{ESP} & \rightarrow \text{ESP} - 4 & \Delta = \Delta - 4 & \Delta = 0 \\
\text{STORE [ESP], EBX} & : & \text{STORE [ESP-8], EBX} & & \Delta = - 8 & \Delta = 0 \\
\text{INC ESP} & : & \text{ESP} & \rightarrow \text{ADD ESP, 1} & \text{Need to sync ESP!} & \Delta = 0 \\
\end{align*}
\]

- **Provides the following benefits**
  - Improves decode BW: PUSH, POP and RET become single µop instructions
  - Conserves rename, execution and retire resources
  - Remove dependencies on ESP – can execute stack operations in parallel
  - Saves power
• **Caches the μops coming out of the decoders**
  - Up to 1.5K μops (32 sets × 8 ways × 6 μops/way)
  - Next time μops are taken from the μop Cache
  - ~80% hit rate for most applications
  - Included in the IC and iTLB, flushed on a context switch

• **Higher Bandwidth and Lower Latency**
  - More cycles sustaining 4 instruction/cycle
    - In each cycle provide μops for instructions mapped to 32 bytes
    - Able to ‘stitch’ across taken branches in the control flow
Decoded μop-Cache

- Decoded μop Cache lets the normal front end sleep
  - Decode one time instead of many times

- Branch misprediction penalty reduced
  - The correct path is also the most efficient path

Save Power while Increasing Performance
Loop Stream Detector (LSD)

- **LSD detects small loops that fit in the µop queue**
  - The µop queue streams the loop, allowing the front-end to sleep
  - Until a branch miss-prediction inevitably ends it

- **Loops qualify for LSD replay if all following conditions are met**
  - Up to 64 µops, with ≤8 taken branches, ≤ 8 32-byte chunks
  - All µops are also resident in the µop-Cache
  - No CALL or RET
  - No mismatched stack operations (e.g., more PUSH than POP)
The Renamer

- **Moves $\leq 4\mu$ops/cycle from the $\mu$op-queue to the OOO**
  - Renames architectural sources and destinations of the $\mu$ops to micro-architectural sources and destinations
  - Allocates resources to the $\mu$ops, e.g., load or store buffers
  - Binds the $\mu$op to an appropriate dispatch port
  - Up to 2 branches each cycle

- **Some $\mu$ops are executed to completion during rename, effectively costing no execution bandwidth**
  - A subset of register-to-register MOV and FXCHG
  - Zero-Idioms
  - NOP

From the Optimization Manual
Dependency Breaking Idioms

- **Zero-Idiom** – an instruction that zeroes a register
  - Regardless of the input data, the output data is always 0
  - E.g.: XOR REG, REG and SUB REG, REG
  - No μop dependency on its sources

- **Zero-Idioms are detected and removed by the Renamer**
  - Do not consume execution resource, have zero exe latency

- **Zero-Idioms remove partial register dependencies**
  - Improve instruction parallelism

- **Ones-Idiom** – an instruction that sets a register to “all 1s”
  - Regardless of the input data the output is always "all 1s"
  - E.g., CMPEQ XMM1, XMM1;
    - No μop dependency on its sources, as with the zero idiom
    - Can execute as soon as it finds a free execution port
  - As opposed to Zero-Idiom, the Ones-Idiom μop must execute
Out-of-Order Execution

- The Scheduler queues \( \mu \)ops until all source operands are ready
  - Schedules and dispatches ready \( \mu \)ops to the available execution units in as close to a first in first out (FIFO) order as possible

- Physical Register File (PRF)
  - Instead of centralized Retirement Register File
    - Single copy of every data with no movement after calculation
  - Allows significant increase in buffer sizes
    - Dataflow window \(~33\%\) larger

- Retirement
  - Retires \( \mu \)ops in order and handles faults and exceptions
Intel® Advanced Vector Extensions

- Vectors are a natural data-type for many apps
  - Extend SSE FP instruction set to 256 bits operand size
  - Extend all 16 XMM registers to 256 bits

- Non-destructive source syntax
  - E.g., VADDPS ymm1, ymm2, ymm3

- New Operations to enhance vectorization
  - Broadcasts, masked load & store

Wide vectors + non-destructive source: more work with fewer instructions
Extending the existing state is area and power efficient
FMA – Fused Multiply Add

- FMA performs $C += A \times B$, where $A$, $B$, and $C$ are Vectors
  - Common operation, e.g., in matrix multiplication

<table>
<thead>
<tr>
<th></th>
<th>255..124</th>
<th>223..192</th>
<th>191..160</th>
<th>159..128</th>
<th>127..96</th>
<th>95..64</th>
<th>63..32</th>
<th>31..0</th>
</tr>
</thead>
<tbody>
<tr>
<td>ymm1</td>
<td>a7</td>
<td>a6</td>
<td>a5</td>
<td>a4</td>
<td>a3</td>
<td>a2</td>
<td>a1</td>
<td>a0</td>
</tr>
<tr>
<td>ymm2</td>
<td>b7</td>
<td>b6</td>
<td>b5</td>
<td>b4</td>
<td>b3</td>
<td>b2</td>
<td>b1</td>
<td>b0</td>
</tr>
<tr>
<td>ymm3</td>
<td>c7</td>
<td>c6</td>
<td>c5</td>
<td>c4</td>
<td>c3</td>
<td>c2</td>
<td>c1</td>
<td>c0</td>
</tr>
<tr>
<td>ymm3</td>
<td>c7+b7×a7</td>
<td>c6+b6×a6</td>
<td>c5+b5×a5</td>
<td>c4+b4×a4</td>
<td>c3+b3×a3</td>
<td>c2+b2×a2</td>
<td>c1+b1×a1</td>
<td>c0+b0×a0</td>
</tr>
</tbody>
</table>

- Dedicated FMA execution units
  - 2× peak FLOPs/cycle
  - 5-cycle FMA latency same as an FP multiply
AVX-512

2011: Sandy Bridge (AVX1)
- 256-bit
- 16 SP / 8 DP
- Flops/Cycle

2013: Haswell (AVX2)
- 256-bit
- 32 SP / 16 DP
- Flops/Cycle (FMA)

Skylake Server (AVX-512)
- 512-bit
- Server: 64SP / 32 DP
- Client: 32 SP / 16 DP
- Flops/Cycle (FMA)

<table>
<thead>
<tr>
<th>AVX</th>
<th>AVX2</th>
<th>AVX512</th>
</tr>
</thead>
<tbody>
<tr>
<td>256-bit basic FP</td>
<td>Float16 (IVB 2012)</td>
<td>512-bit FP/Integer</td>
</tr>
<tr>
<td>16 registers</td>
<td>256-bit FP FMA</td>
<td>32 registers</td>
</tr>
<tr>
<td>NDS (and AVX128)</td>
<td>256-bit integer</td>
<td>8x64bit mask registers</td>
</tr>
<tr>
<td>Improved blend</td>
<td>PERMD</td>
<td>Embedded broadcast/rounding</td>
</tr>
<tr>
<td>MASKMOV</td>
<td>Gather</td>
<td>New permute instructions</td>
</tr>
<tr>
<td>Implicit unaligned</td>
<td></td>
<td>Scatter</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Element size</th>
<th>Vector Length</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>128 bits</td>
</tr>
<tr>
<td>Byte</td>
<td>16</td>
</tr>
<tr>
<td>Word</td>
<td>8</td>
</tr>
<tr>
<td>Dword/SP</td>
<td>4</td>
</tr>
<tr>
<td>Qword/DP</td>
<td>2</td>
</tr>
</tbody>
</table>

Computer Structure 2018 – Intel® Core™ μArch
Vector Registers

XMM = 128-bits

YMM = 256 bits

ZMM = 512 bits
AVX-512 Mask Registers

- 8 Mask registers of size 64-bits
  - k1-k7 can be used for predication
    - k0 can be used as a destination or source for mask manipulation operations
    - Mask can blend \{k1\} or zero \{k1\}{z}

```
VADDPPD zmm1 {k1}, zmm2, zmm3
```

<table>
<thead>
<tr>
<th>zmm1</th>
<th>a7</th>
<th>a6</th>
<th>a5</th>
<th>a4</th>
<th>a3</th>
<th>a2</th>
<th>a1</th>
<th>a0</th>
</tr>
</thead>
<tbody>
<tr>
<td>zmm2</td>
<td>b7</td>
<td>b6</td>
<td>b5</td>
<td>b4</td>
<td>b3</td>
<td>b2</td>
<td>b1</td>
<td>b0</td>
</tr>
<tr>
<td>zmm3</td>
<td>c7</td>
<td>c6</td>
<td>c5</td>
<td>c4</td>
<td>c3</td>
<td>c2</td>
<td>c1</td>
<td>c0</td>
</tr>
<tr>
<td>k1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>zmm1</td>
<td>b7+c7</td>
<td>a6</td>
<td>b5+c5</td>
<td>b4+c4</td>
<td>b3+c3</td>
<td>b2+c2</td>
<td>a1</td>
<td>a0</td>
</tr>
</tbody>
</table>
```
void foo(const double* a, const double* b, double* c, int n) {
    for(int i=0; i<n; ++i)
        c[i] = a[i] + b[i];
}

void foo(const double* a, const double* b, double* c, int n) {
    #pragma omp simd
    for(int i=0; i<n; ++i)
        c[i] = a[i] + b[i];
}

The compiler uses AVX
And also may break loop iterations to threads
Execution Units

- Improved throughput and latency for divide/sqrt and approximate reciprocals.
- Identical latency and throughput for all operations running on FMA units.
- Latency improvements for a significant portion of the SSE, AVX and general-purpose instructions.
• The L1 D$ handles 2× 256-bit loads + 1× 256-bit store per cycle

• The ML$ can service one cache line (64 bytes) each cycle

• 72 load buffers keep load μops from allocation till retirement
  – Re-dispatch blocked loads

• 42 store buffers keep store μops from allocation till the store value is written to L1-D$
  – or written to the line fill buffers – for non-temporal stores

• Support Store-to-load forwarding and Memory Disambiguation
Memory Subsystem

- Handles up to 10 outstanding cache misses in the LFBs
  - Continues to service incoming stores and loads

- Two hardware prefetchers load data to the L1 D$
  - DCU prefetcher (the streaming prefetcher)
    - Triggered by an ascending access to very recently loaded data
    - Fetches the next line, assuming a streaming load
  - Instruction pointer (IP)-based stride prefetcher
    - Tracks individual load instructions, detecting a regular stride
      - Prefetch address = current address + stride
      - Detects strides of up to 2K bytes, both forward and backward
# Skylake Cache Parameters

<table>
<thead>
<tr>
<th>Level</th>
<th>Capacity / ways</th>
<th>Line Size (bytes)</th>
<th>Fastest Latency1</th>
<th>Peak Bandwidth (bytes/cyc)</th>
<th>Sustained Bandwidth (bytes/cyc)</th>
<th>Update Policy</th>
</tr>
</thead>
<tbody>
<tr>
<td>L1 data cache</td>
<td>32 KB/ 8</td>
<td>64</td>
<td>4 cycles</td>
<td>2×32B Load+ 1×32B Store /cyc</td>
<td>~81 B/cyc</td>
<td>Writeback</td>
</tr>
<tr>
<td>Instruction cache</td>
<td>32 KB/8</td>
<td>64</td>
<td>N/A</td>
<td>N/A</td>
<td>N/A</td>
<td>N/A</td>
</tr>
<tr>
<td>L2 cache</td>
<td>256KB/4</td>
<td>64</td>
<td>12 cycle</td>
<td>64 B/cyc</td>
<td>~29 B/cyc</td>
<td>Writeback</td>
</tr>
<tr>
<td>L3 cache</td>
<td>Up to 2MB per core/Up to 16 ways</td>
<td>64</td>
<td>44 cycles</td>
<td>32 B/cyc</td>
<td>~18 B/cyc</td>
<td>Writeback</td>
</tr>
</tbody>
</table>
## Skylake TLB Parameters

<table>
<thead>
<tr>
<th>Level</th>
<th>Page Size</th>
<th>Entries</th>
<th>Associativity</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Instruction TLB</strong></td>
<td>4KB</td>
<td>128</td>
<td>8</td>
</tr>
<tr>
<td></td>
<td>2MB/4MB</td>
<td>8 per thread</td>
<td>8</td>
</tr>
<tr>
<td><strong>Data TLB</strong></td>
<td>4KB</td>
<td>64</td>
<td>4</td>
</tr>
<tr>
<td></td>
<td>2MB/4MB</td>
<td>32</td>
<td>4</td>
</tr>
<tr>
<td></td>
<td>1GB</td>
<td>4</td>
<td>4</td>
</tr>
<tr>
<td><strong>Second Level TLB</strong></td>
<td>Shared by 4KB and 2/4MB pages</td>
<td>1536</td>
<td>12</td>
</tr>
<tr>
<td></td>
<td>1GB</td>
<td>16</td>
<td>4</td>
</tr>
</tbody>
</table>
## Buffer Sizes

Extract more parallelism in every generation

<table>
<thead>
<tr>
<th></th>
<th>Nehalem</th>
<th>Sandy Bridge</th>
<th>Haswell</th>
<th>Skylake</th>
</tr>
</thead>
<tbody>
<tr>
<td>Out-of-order Window</td>
<td>128</td>
<td>168</td>
<td>192</td>
<td>224</td>
</tr>
<tr>
<td>In-flight Loads</td>
<td>48</td>
<td>64</td>
<td>72</td>
<td>72</td>
</tr>
<tr>
<td>In-flight Stores</td>
<td>32</td>
<td>36</td>
<td>42</td>
<td>56</td>
</tr>
<tr>
<td>Scheduler Entries</td>
<td>36</td>
<td>54</td>
<td>60</td>
<td>97</td>
</tr>
<tr>
<td>Integer Register File</td>
<td>N/A</td>
<td>160</td>
<td>168</td>
<td>180</td>
</tr>
<tr>
<td>FP Register File</td>
<td>N/A</td>
<td>144</td>
<td>168</td>
<td>168</td>
</tr>
<tr>
<td>Allocation Queue</td>
<td>28/thread</td>
<td>28/thread</td>
<td>56</td>
<td>64/thread</td>
</tr>
</tbody>
</table>