CPU architecture

October 2016 - Romain Dolbeau
Global description
Example: processor Intel « Sandy Bridge »

<table>
<thead>
<tr>
<th>Core 0</th>
<th>Core 1</th>
<th>Core 2</th>
<th>Core 3+</th>
</tr>
</thead>
<tbody>
<tr>
<td>2x32KB L1 Cache</td>
<td>2x32KB L1 Cache</td>
<td>2x32KB L1 Cache</td>
<td>2x32KB L1 Cache</td>
</tr>
<tr>
<td>256KB L2 Cache</td>
<td>256KB L2 Cache</td>
<td>256KB L2 Cache</td>
<td>256KB L2 Cache</td>
</tr>
</tbody>
</table>

8MB L3 Cache

DDR3 memory controllers

QuickPath interconnect

4x12.8GB/s

2x32GB/s
CPU architecture

Instructions & pipelines

01-10-2016
Instruction pipelining (1)

- Traditional example: 5 execution steps
  - Instruction Fetch (IF)
  - Instruction Decode (ID)
  - Execution (EX)
  - Memory Read/Write (MEM)
    - For Load/Store only
  - Result Writeback (WB)

- Sequential execution: at best 4 cycles (not including memory) per instruction
Instruction pipelining (2)

- Pipelining allows to lower the average to one cycle per instruction by executing different steps simultaneously
  - “scalar” pipeline

- During any given cycle, several instructions are in progress
  - each at a different step

![Pipeline stages diagram]

From wikimedia commons
**Superscalar processor**

- All the modern processors are « superscalar », with several pipelines processing several instructions per cycle.
  - During any given cycle, several instructions are in progress.
    - More than one instruction can be at the same stage.

<table>
<thead>
<tr>
<th>IF</th>
<th>ID</th>
<th>EX</th>
<th>MEM</th>
<th>WB</th>
</tr>
</thead>
<tbody>
<tr>
<td>IF</td>
<td>ID</td>
<td>EX</td>
<td>MEM</td>
<td>WB</td>
</tr>
<tr>
<td>IF</td>
<td>ID</td>
<td>EX</td>
<td>MEM</td>
<td>WB</td>
</tr>
<tr>
<td>IF</td>
<td>ID</td>
<td>EX</td>
<td>MEM</td>
<td>WB</td>
</tr>
</tbody>
</table>

*From wikimedia commons*
Throughput and instructions latency

Two important notions:

- **Throughput**: rate at which an instruction may be executed, in the best case without data dependency;
- **Latency**: time required for an instruction to finish and give its result.

---

**Timing (cycles)**

- **Inst r0, r1**: 1 instruction per cycle, 5 cycles of latency
- **Inst r2, r1**: 1 instruction per cycle, 5 cycles of latency
- **Inst r0, r1**: 2 instructions per cycle, 4 cycles of latency
- **Inst r2, r0**: 2 instructions per cycle, 4 cycles of latency
- **Inst r3, r1**: 2 instructions per cycle, 4 cycles of latency
- **Inst r4, r3**: 2 instructions per cycle, 4 cycles of latency
Producer - consumer latency (1/2)

- The « producer - consumer » is the minimum delay between the instruction that « produces » a result and the instruction that « consumes » it
- The higher the latency, the greater the number of « bubbles » (lost execution opportunities)
  - If no instructions executed between the producer and the consumer
- Loss of efficiency
  - Decreased by other mechanisms such « out-of-order » execution

Note on next page, for the latencies:
Producer - consumer latency (2/2)

Example

- 2 floating-point multiplication instructions
- If one independent from the other on Haswell: 1 cycle \( \square \) (2 pipelines)
  - \texttt{mulpd \%xmm0, \%xmm1 ; mulpd \%xmm2, \%xmm3}
  - the 1\textsuperscript{st} reads & writes \%xmm0, reads \%xmm1 ; the 2\textsuperscript{nd} reads & writes \%xmm2, writes \%xmm3
- If one dependent on the other on Haswell: 6 cycles \( \square \) (latency of 5 cycles \[1\])
  - \texttt{mulpd \%xmm0, \%xmm1 ; mulpd \%xmm0, \%xmm3}
  - Both read & write \%xmm0

Left: independent ; right: dependent

<table>
<thead>
<tr>
<th>Cycle</th>
<th>Pipe 0</th>
<th>Pipe 1</th>
<th>Pipe 0</th>
<th>Pipe 1</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>\texttt{mulpd %xmm0,%xmm1}</td>
<td>\texttt{mulpd %xmm2,%xmm3}</td>
<td>\texttt{mulpd %xmm0,%xmm1}</td>
<td>(idle)</td>
</tr>
<tr>
<td>1-4</td>
<td>(whatever follows)</td>
<td>(whatever follows)</td>
<td>(idle)</td>
<td>(idle)</td>
</tr>
<tr>
<td>5</td>
<td>(whatever follows)</td>
<td>(whatever follows)</td>
<td>\texttt{mulpd %xmm0,%xmm3}</td>
<td>(whatever follows)</td>
</tr>
</tbody>
</table>
Out of Order (OoO) execution

- Out of order execution consist in execution of the instructions in an order different from the one of the binary program
  - But maintaining semantics: registers, memory (more on memory later in this presentation)
- It allows to fill the « bubbles » by executing instructions whose operands are already available
- Example
  - mulpd %xmm0,%xmm1 ; mulpd %xmm0,%xmm2 ; mulpd %xmm3,%xmm4 ; mulpd %xmm3,%xmm5

<table>
<thead>
<tr>
<th>Cycle</th>
<th>Pipe 0</th>
<th>Pipe 1</th>
<th>Pipe 0</th>
<th>Pipe 1</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>mulpd %xmm0,%xmm1</td>
<td>(idle)</td>
<td>mulpd %xmm0,%xmm1</td>
<td>mulpd %xmm3,%xmm4</td>
</tr>
<tr>
<td>1-4</td>
<td>(idle)</td>
<td>(idle)</td>
<td>(idle)</td>
<td>(idle)</td>
</tr>
<tr>
<td>5</td>
<td>mulpd %xmm0,%xmm2</td>
<td>mulpd %xmm3,%xmm4</td>
<td>mulpd %xmm0,%xmm2</td>
<td>mulpd %xmm3,%xmm5</td>
</tr>
<tr>
<td>6-9</td>
<td>(idle)</td>
<td>(idle)</td>
<td>(whatever follows)</td>
<td>(whatever follows)</td>
</tr>
<tr>
<td>10</td>
<td>mulpd %xmm3,%xmm5</td>
<td>(whatever follows)</td>
<td>(whatever follows)</td>
<td>(whatever follows)</td>
</tr>
</tbody>
</table>
Pipelining lessons

► Filling floating-point pipelines
  - Independent instructions
  - Available data
  - Hence simultaneously several independent calculations
    • Unrolling
    • Software pipelining
    • Simultaneous Multithreading / Hyperthreading (later in the presentation)
  - Parallelism exploitation at instructions level
    • Compete with vectorization or threading

► If one have 16 independent multiplications
  - 2 consecutive AVX-512 instructions simultaneously executable
    • If enough registers are available...
  - 1 scalar instruction executed by 16 threads
  - Better having 256 multiplications and combining both approaches...
  - Memory has to follow!
Consequences for scientific computing

▶ Sandy Bridge is able to do a (floating) multiplication and an add per cycle
  – Vectorization taken on later
▶ But the latency of a multiplication is 5 cycles, whereas the latency of an add is 3 cycles
▶ If instructions are dependent (producer – consumer), a lot of « bubbles » within the pipeline(s)
  – Efficiency loss 
▶ Needs for independent instructions, but then needs to access to many operands
▶ Haswell introduces the Fused Multiply Add (FMA, D = A*B+C)
  – Haswell:
    • MUL has a latency of 5 cycles
    • ADD has a latency of 3 cycles
    • FMA has a latency of 5 cycles
  – Broadwell:
    • MUL falls to 3 cycles
    • FMA improved by 1 cycle
CPU architecture

Additional considerations
Branch & prediction

► When there is a change in the flow control
  – Return backward in the loop
  – Conditional
  – Call to function

► Next instruction loaded from memory, decoded, sent in the pipeline...
  – Introduce a lot of « bubbles » in the pipelines

► The processor « predicts » direction, predicts loading & execution
  – « Speculative » execution
  – Wins if true forecast, otherwise loses a little
  – Very good success rate with numerical programs
    • N iterations loop, forecast branch used: 1 error for N attempts

► First approximation
  – Cost null in the loops
  – Cost low for conditionals, but problem with vectorization
    • Avoid conditionals in loops
Simultaneous Multi-Threaded (SMT) or HyperThreading

- 2 CPUs in 1! (4 in 1 for KNL)

- Ports (columns)
- Structural hazard
- Data dependency
- Consequences of structural hazard
- Cycles (lines)
CPU architecture

Caches
Caches justification

- A lot of operations need a lot of operands, so a lot of memory
- Memory latency is very high – therefore a lot of latency in the producer – consumer link
  - Between a « producer » load and the « consumer » instruction
  - Efficiency loss
- Caches are small, but fast, memories that decrease access latency to operands
- To get a gain of performance, data has to be in the cache at load time
- **Mechanisms to « anticipate » the load**
  - Locality
  - Prefetching
Memory hierarchy

- **64 bits** Registers
- **32 kB/256 kB/2MB** Cache
- **Local memory** Cache line
- **Distant memory** message
- **Disk/Storage** pages

Bandwidth size

- **xGB**
- **xGB → xPB**

Latency
### Latencies

#### Relevant benchmarks:
- `lat_mem_rd` (LMBench)
  - [http://www.bitmover.com/lmbench/get_lmbench.html](http://www.bitmover.com/lmbench/get_lmbench.html)

#### Hardware counters

<table>
<thead>
<tr>
<th>Latency (cycles)</th>
<th>Theo. (1)</th>
<th>Measured</th>
</tr>
</thead>
<tbody>
<tr>
<td>DCU</td>
<td>4-5</td>
<td>4</td>
</tr>
<tr>
<td>MLC</td>
<td>12</td>
<td>12</td>
</tr>
<tr>
<td>LLC (avg)</td>
<td>41</td>
<td>40</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Latency (cycles)</th>
<th>NHM</th>
<th>WST</th>
<th>SNB</th>
</tr>
</thead>
<tbody>
<tr>
<td>DCU</td>
<td>4</td>
<td>4</td>
<td>4</td>
</tr>
<tr>
<td>MLC</td>
<td>10</td>
<td>10</td>
<td>12</td>
</tr>
<tr>
<td>LLC</td>
<td>38</td>
<td>43</td>
<td>40</td>
</tr>
</tbody>
</table>
“Pinning” / “Binding”

- « Pinning » (affinity) consists in the restriction of the bitmask, allowing to limit a thread (or a process) execution to a subset of cores

- It allows to improve and stabilize the performance by not disturbing caches, and the NUMA behavior
  - NUMA details in another part of the training

- A lot of methods to reach the goal
  - Batch manager (process)
  - MPI library (process)
  - OpenMP (native since v4, or depending on implementation)
    - e.g. KMP_AFFINITY with Intel OpenMP
  - Command numactl (process)
  - Command taskset (process)
  - …
Cache: principles of operations

▶ Exploitation of locality to decrease latency

1. Spatial locality
   - Data close in memory are often used together
   - Structures, arrays, ...

2. Temporal locality
   - Data frequently used

▶ Use of the « cache line »
   - A contiguous subset of the memory
   - Typically 32 or 64 or 128 bytes
     • 64 bytes on current Intel processors including Knights Corner
Spatial locality: example

- We assume that an object of type “struct position3d” is at address X
  - With a size of 12 bytes here
- If the cache line is 64 bytes, then the object is loaded in one go
  - Plus 52 other bytes...
- If one uses successively x, y, and z, then two loads are saved – y & z are already present in the cache, loaded at the same time than x
- But 52 bytes are loaded for nothing...
- Except if for example one have an array of “position3d” – then the 4 next objects of the array are already loaded
- But if one uses only one of the three coordinates...
  - All the y and z are a waste of bandwidth
  - Having the x side by side guarantees the spatial locality in the case of an array!
  - We will see for vectorization...

```c
struct position3d {
    int x;
    int y;
    int z;
};
```
Temporal locality: example

- More natural than spatial locality…
  - … let the audience judges
- Close temporal use of the same memory address
- In example #1, locality on the 2 arrays
  - “coef” is used at each iterations “i”
  - 4 over 5 elements of X used at iteration “i” are used again at “i+1”
  - As a corollary, iteration “i” reuse 4 over 5 elements of X used at iteration “i-1”
- In example #2, locality on X is broken
  - Each iteration “i” uses a subset of X disjointed from the subset of the iterations “i-1” and “i+1”
  - Locality between the iterations “k”, but if N is big compared to cache size then no benefit

- Those examples will show up again in the talk about vectorization…

```c
#1
for (i = 2 ; i < N-2 ; i+)
{ 
  Y[i] = 0.;
  for (k = 0 ; k < 5 ; k+)
  { 
    Y[i] += X[i-2+k]*coef[k];
  }
}
```

```c
#2
for (i = 2 ; i < N-2 ; i+)
{ 
  Y[i] = 0.;
  for (k = 0 ; k < 5 ; k+)
  { 
    for (i = 2 ; i < N-2 ; i+)
    { 
      Y[i] += X[i-2+k]*coef[k];
    }
  }
}
```
Consequence for scientific computing

Must exploit properly the caches in order to not be limited by memory bandwidth

1. Spatial locality
   - Structure of arrays rather than array of structures
     • Unless using all the members of the structure
   - Access to arrays in the order of storage
   - But see also vectorization

2. Temporal locality
   - All the processing over a data rather than a processing over all the data
   - But see also vectorization
Prefetching

- Prefetching consists in loading data before needing them
- Allows to hide memory latency by anticipating the loads
- But it should be done appropriately
  - Loading an unused data consumes memory bandwidth uselessly
  - Data loaded too early can be ejected from the cache before its use
  - Data loaded too late do not hide (all) the latency
- Two approaches
  - Software: instructions prefetching in the code
  - Hardware: mechanism of accesses detection & anticipation
- Both are available on x86-64
  - Knights Corner is software-only!
- Very dependent on the micro-architecture
  - Hardware prefetcher, specific latencies for caches and memories, cache sizes, etc.
When a cache « line » is loaded, where will it be stored?
- 64 bytes to put in 32 KiB, 512 possibilities

Anywhere: « fully associative »
- Ideal for cache hit, no placement conflict
- Complex & costly to implement
- High latency when checking if a line is present

Single place: « direct mapped »
- Trivial check & insertion: memory address modulo cache size
- Simple, low cost, low latency
- Two addresses used simultaneous mapped at the same place…
  • “ping-pong” effect: one ejects another, another ejects one, etc.

A small finite number of places: “set associative”
- Each memory address can go in 2, 4, … places (“associativity”)
- Compromise between complexity and efficiency
- Most common solution
Caches organization (2/2)

▶ Beware: in cache mode, fast memory (MCDRAM) of Knight Landings is in mode « direct mapped » !

▶ Be careful to the congruence of the modulos of arrays addresses...
Cache: line replacement

- When there is no more space in the cache and that a line has to be stored inside, another one has to be removed
  - Choose the line to replace (replacement policy)
  - Do not lose data (« flush » modified data to memory)
- In « direct-mapped » no choice – one single possible place
- In « set associative » or « fully associative » – decision to make
  - LRU (« Least recently used »), Pseudo-LRU, Random, LFU (« Least frequently used »), ...
- If the line was not modified (identical to memory), nothing to do
- Otherwise, the line has to be copied towards a more remote cache and/or the memory
- Very low level behavior, low influence from the programmer
  - Except for « direct-mapped », which can cause problems
  - And in some extreme case, “ping-pong” effect over a set of values larger than the associativity
A write (store) is often smaller than the entire cache line
- line = 64 bytes, double = 8 bytes

Then for a write it is needed to...
- Load the entire line from the memory to the cache
- Replace a subset of the data in the line by the data in the store

But if several consecutive writes replace all the data in the line, then the memory to cache load is wasted time (and bandwidth)

Tell to the processor « these data are part of a sequence » (stream) to avoid the load: « streaming stores » or « nontemporal stores »

code: #pragma vector nontemporal [A]
compiler: -opt-streaming-stores always, -opt-streaming-cache-evict=X

code: _mm_stream_pd, _mm256_stream_pd, _mm512_stream_pd, ...

It also exists cache indications for the loads
Cache & multi-cores

- Reminder: the spatial locality is obtained by the cache line, a group of data read or written simultaneously
  - 64 bytes on x86-64
- If only one core uses the line, no difference with mono-core
- Otherwise - coherence issue
  - Each cache can have a copy of the line
  - The memory also has a copy
- And sharing issue – «true» or «false»
  - True: required by the algorithm
    • e.g. synchronization variable
  - False: induced by the implementation and not by the algorithm
    • Two data in the same cache line
    • Used by two different cores
    • “Ping-pong” effect – both cores need to use the line, even though they don’t share data
typedef struct {
    int neg;
#ifdef NOCONFLICT
    int pad[15];
#endif
    int pos;
} counters;

#pragma omp parallel
{
#pragma omp sections
{
    /* first openmp section: count negative values */
    #pragma omp section
    {
        for (i = 0 ; i < SIZE ; i ++) {
            if (input[i] < 0.)
                c.neg ++;
        }
    }
    /* second openmp section: count positive values */
    #pragma omp section
    {
        for (i = 0 ; i < SIZE ; i ++) {
            if (input[i] > 0.)
                c.pos ++;
        }
    }
} /* end sections */
} /* end parallel */
## False sharing (2) [Nehalem]

```plaintext
#undef NOCONFLICT

<table>
<thead>
<tr>
<th>Event</th>
<th>core 0</th>
<th>core 1</th>
</tr>
</thead>
<tbody>
<tr>
<td>INSTR_RETIRED_ANY</td>
<td>1.20797e+09</td>
<td>1.47641e+09</td>
</tr>
<tr>
<td>CPU_CLK_UNHALTED_CORE</td>
<td>9.41486e+08</td>
<td>9.41545e+08</td>
</tr>
<tr>
<td>L1D_REPL</td>
<td>2.45502e+07</td>
<td>2.45488e+07</td>
</tr>
<tr>
<td>L1D_M_REPL</td>
<td>3.12177e+06</td>
<td>3.13701e+06</td>
</tr>
</tbody>
</table>

#define NOCONFLICT

<table>
<thead>
<tr>
<th>Event</th>
<th>core 0</th>
<th>core 1</th>
</tr>
</thead>
<tbody>
<tr>
<td>INSTR_RETIRED_ANY</td>
<td>1.20798e+09</td>
<td>1.47641e+09</td>
</tr>
<tr>
<td>CPU_CLK_UNHALTED_CORE</td>
<td>8.51338e+08</td>
<td>8.51594e+08</td>
</tr>
<tr>
<td>L1D_REPL</td>
<td>1.68129e+07</td>
<td>1.68122e+07</td>
</tr>
<tr>
<td>L1D_M_REPL</td>
<td>547</td>
<td>574</td>
</tr>
</tbody>
</table>
```
CPU architecture

NUMA system
UMA: Unified Memory Access

- Example of multi-core mono-socket systems
  - Mostly, for Intel
- Usually identical cores
- Same memory access (bandwidth, latency) for all the cores
- Also CC-UMA
  - Cache Coherent UMA
NUMA: Non-Uniform Memory Access (1/3)

- Generally the union of several UMA basic blocks, typically a socket
- A core can access to the memory of another group of cores
- The access through the link (QPI for Intel) is less efficient
  - Higher latency
  - Lower bandwidth
- Now, NUMA inside a socket
  - AMD Bulldozer & offsprings
  - Intel Knights Landing
    - In some configurations
Above: 2 NUMA zones, 2 latencies ("local", "remote")

Right: 4 NUMA zones, 3 latencies
- "local"
- "remote 1 hop"
- "remote 2 hop" (diagonal)
The communication internal to a processor can create a “NUMA” effect depending on the combination cores – memory controller.

- Right: a simplified view of Intel Knights Corner
  - First generation Xeon Phi
  - Core-to-memory must traverse the ring
NUMA configuration

Intel “Sandy Bridge”

- Xeon® E5-2600
- Cores
- Patsburg
- DDR3
- QPI

AMD “Interlagos”

- Opteron 6200
- Cores
- Chipset
- HT3
- PCIe gen3
- DDR3
- QPI
NUMA & Linux

Where (physically, with NUMA meaning) is memory allocated?
- `malloc()` in C, `allocate()` in Fortran, etc.

Depends on the operating system & on the method of allocation

Let’s consider Linux...
- Memory is *virtually* allocated only during an anonymous `malloc()` or `mmap()`
- Memory is *physically* allocated during first access
- First access might be implicit (`calloc()`, …)
- The page is located on the NUMA node closest to the core causing the physical allocation

Careful: a sequential physical allocation implies all the pages on the same NUMA node!
- Very bad for multi-thread performance – good with MPI

Need for an alternative behavior in some cases
Reminder: Virtual vs. physical memory

▶ Physical memory: memory modules/sticks
  – Electric charge inside transistors
  – « Physically » addressed by the memory controller
    • And some peripheral devices such as IB HBA
  – On 40, 44, 48 bits...
  – Normally, never seen by an « application » programmer

▶ Virtual memory: what is seen by the code
  – Abstract address on 32 (obsolete) or 64 bits, i.e. « pointer »
    • The same virtual address may be used by several processes!
  – Gathered in zone called « page » of 4 KiB, 2 MiB or 1 GiB (on x86-64)
  – To each virtual « page » is associated a « physical » page
  – This association can change with time (« paging » or « swapping ») and is handled by the operating system
    • But it can be locked for peripheral devices
  – The same physical page can be seen by several virtual pages
    • Shared memory
NUMA & libnuma

libnuma is useful for an alternative allocation
- To specify a different default behavior
  • numa_set_interleave_mask, numa_set_localalloc, ...
- To do a specific allocation
  • numa_alloc_interleaved, numa_alloc_local, numa_alloc_onnode, ...
- To shift pages in memory
  • numa_move_pages, numa_migrate_pages, ...
- And various support functions

numactl is a “prefix” command to enforce a default policy
- E.g. numactl --interleave 0,2 ./mybinary

Alternatively the library “hwloc”
- Very advanced management of CPU/memory placement, topology, etc.
NUMA: exploitation

- Must use memory on all the NUMA nodes to maximize the bandwidth
  - Since “Nehalem”, the bandwidth is “by socket”
- But avoid distant accesses to minimize latency
  - See graph later
- Allocation of a physical page on the node that will use it the most
  - ▶
- If at all possible, the physical allocation (« first touch ») should use the same parallelism as the computational parts
  - Should be enough to get an ideal behavior
  - Difficult without a fine control of physical allocation
NUMA: recipes

▶ Single process, only one thread (non-parallel)
  – By default 50% of the bandwidth (dual socket), minimal latency
  – Use of numactl -interleave=all increases the available bandwidth but degrades (increases) the latency

▶ Several processes, each with one thread (MPI)
  – Default behavior is excellent 😊
  – Each process will allocate in the closest memory

▶ Single process, several threads (OpenMP)
  – Parallel initialization
  – Otherwise, try to use numactl -interleave=all

▶ Several processes, each with several threads (MPI + OpenMP)
  – Default behavior is excellent 😊
  – ...with at least one process per socket (NUMA zone)
CPU Architecture

x86-64 Vectorization

01-10-2016
Flynn Taxonomy

First computer architecture classification proposed by Michael J. Flynn (1966)

Four main categories according to data and instructions streams
- SISD (Single Instruction stream, Single Data stream)
- SIMD (Single Instruction stream, Multiple Data streams)
- MISD (Multiple Instruction streams, Single Data stream)
- MIMD (Multiple Instruction streams, Multiple Data streams)
  - Shared memory (CPU cores)
  - Distributed memory (clusters)
Flynn Taxonomy (1/4)

- SISD: single instruction, single data
  - Sequential

```
Control | Processor | Memory
```

Data
Instruction
Flynn Taxonomy (2/4)

- SIMD: single instruction, multiple data
  - Vectorization

- Data
- Instruction
Flynn Taxonomy (3/4)

- MIMD: multiple instructions, multiple data
  - MPI
  - Separate process
Taxinomie de Flynn (4/4)

- MISD: multiple instructions, single data
x86-64 Architecture

- 16 general purpose registers (8 in 32 bits mode)
- Variable-length instructions, rich semantic (CISC)
- Various memory addressing schemes for memory operands
  - Register1 + Register2 * scale + offset
  - Designed for direct access to array of structure
    - Register_1[Register_2].shift (in C code)
- Instructions use 0-2 operand(s)
  - Operations necessarily erase one register (A <- A + B )
  - Mnemonic instructions does not explicitly show all registers impacted
    - May implicitly write to general-purpose registers or status register
- Support unaligned memory access
  - memory address is not a multiple of the size of the object addressed
- Little Endian
  - Order of bytes in memory
Vectorization principle / SIMD

▶ The ideal case: SIMD afford to accelerate N times a SISD code
### Performance Loss of non-vectorized code

<table>
<thead>
<tr>
<th></th>
<th>Single Precision</th>
<th></th>
<th>Double Precision</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Elements/vector</td>
<td>Loss</td>
<td>Elements/vector</td>
<td>Loss</td>
</tr>
<tr>
<td>SSE</td>
<td>4</td>
<td>75%</td>
<td>2</td>
<td>50%</td>
</tr>
<tr>
<td>AVX</td>
<td>8</td>
<td>87.5%</td>
<td>4</td>
<td>75%</td>
</tr>
<tr>
<td>AVX-512</td>
<td>16</td>
<td>93.75%</td>
<td>8</td>
<td>87.5%</td>
</tr>
</tbody>
</table>

- “Loss” represents the percentage of the peak performance unexploited by a scalar code
- Calculated by:
  \[(\text{Elements/vector} - 1) / \text{Elements/vector}\]
## Vector data type overview: MMX

### 8 x 64-bits registers with 4 types (Archaic)

- **Packed byte**: 8x8 bits elements (char)
  - Elements: 0, 1, 2, 3, 4, 5, 6, 7

- **Packed word**: 4x16 bits elements (short)
  - Elements: 0, 1, 2, 3

- **Packed doubleword**: 2x32 bits elements (int)
  - Elements: 0, 1

- **Packed quadword**: 64 bits (long)
  - Elements: 0
## Vector data type overview: SSE

- **16x 128-bits registers and 6 types [outdated?]**
  - Packed byte: 16x8bits elem. (char)
    - Packed word: 8x16bits elem. (short)
    - Packed doubleword: 4x32bits elem. (int / float)
    - Packed quadword: 2x64bits (long / double [SSE2])
## Vector data type overview: AVX

16 x 256-bits registers with 6 types [current]

- **Packed byte:** 32x8bits elem. (char) [AVX2]
  
<table>
<thead>
<tr>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
<th>10</th>
<th>11</th>
<th>12</th>
<th>13</th>
<th>14</th>
<th>15</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>5</td>
<td>6</td>
<td>7</td>
<td>8</td>
<td>9</td>
<td>10</td>
<td>11</td>
<td>12</td>
<td>13</td>
<td>14</td>
<td>15</td>
</tr>
</tbody>
</table>

- **Packed word:** 16x16bits elem. (short) [AVX2]
  
<table>
<thead>
<tr>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
<th>10</th>
<th>11</th>
<th>12</th>
<th>13</th>
<th>14</th>
<th>15</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>5</td>
<td>6</td>
<td>7</td>
<td>8</td>
<td>9</td>
<td>10</td>
<td>11</td>
<td>12</td>
<td>13</td>
<td>14</td>
<td>15</td>
</tr>
</tbody>
</table>

- **Packed doubleword:** 8x32bits (int [AVX2] / float)
  
<table>
<thead>
<tr>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>5</td>
<td>6</td>
<td>7</td>
</tr>
</tbody>
</table>

- **Packed quadword:** 4x64bits (long [AVX2] / double)
  
<table>
<thead>
<tr>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
<td>2</td>
<td>3</td>
</tr>
</tbody>
</table>
Vector data type overview: AVX-512

- 32 x 512-bits registers with 4 types [next]
  - Packed doubleword: 16x32bits (int / float)
  - Packed quadword: 8x64bits (long / double)
- No type for 8 / 16 bits (char, short)
  - Supported in Skylake
- More registers
- And masking registers
  - Allow partial operations on vectors
  - Skylake will introduce AVX-512 on 128/256 bits vectors
SSE / AVX : Registers (x86-64) (1/2)

- **SSE**
  - 16 registers xmm0 à xmm15 (128 bits)
  - Floating-point unit (replace x87 coprocessor in 64 bits mode)

- **AVX**
  - 16 registers ymm0 à ymm15 (256 bits)
  - Lower part (128 bits) of AVX registers correspond to SSE registers (xmm0 to xmm15)

- **SSE & AVX encoding are different**
  - AVX instructions are based on the “VEX” scheme
  - AVX-512 instructions are based on “EVEX” scheme
  - AVX & AVX-512 are fully compatible
  - Avoid mixing SSE and AVX instructions (transition penalties)
    - Pay attention to libraries and which set they use
SSE / AVX : Registers (x86-64) (2/2)

▶ Similar to the Xeon Phi “Knights Corner” specific instructions.
  - KNC also has “swizzle” of one register in instructions

▶ AVX-512 (upcoming in Knights Landing, Xeon Skylake)
  - 32 registers (zmm0 to zmm3)
  - Lower 256 bits correspond to ymm0 to ymm31 registers (!)
  - Lower 128 bits correspond to xmm0 to xmm31 registers (!)
  - Masking registers allowing partial calculation/load/store operations
    • k0-k7, with k0 hardcoded to all-1

▶ Instructions encoded with “EVEX” working on 128 bits or 256 bits will arrive in Skylake
  - As a stopgap, AVX2 can be used in Knight Landings but without access to ymm16-ymm31 or xmm16-xmm31 or masking
  - Of course, masking can be used to work on partial registers

▶ More details on available instruction in the Intel Intrinsics Guide:
SSE / AVX : Instructions (1/4)

▶ SSE

– Similar to conventional x86, use 0 to 2 operands
  • Store output result in an input register (A <- A+B)
– Non-orthogonal instructions set ▶
  • Not all operations available for all data types
  • In particular for integer operations, e.g. multiplications
– Aligned memory access only in operands
  • Register size, so 16 bytes alignment
– Special instructions for unaligned operation
  • Only pure load/store
– Floating point double precision support from SSE2 onward
  • SSE2 required for all x86-64 CPUs
▶ Baseline for all x86-64 CPUs and compilers !

– Not in KNC, which is distinct from x86-64 for this reason
SSE / AVX : Instructions (2/4)

- **AVX (introduced with Sandy Bridge)**
  - Wider registers (256 bits)
  - Add one operand for instructions (C <- A + B)
    - No need to preserve inputs by explicit copy to another register
  - Support unaligned memory access in operands
    - But still less efficient than aligned accesses
  - No integer operations, only floating-point
    - Except with AVX-128

- **The overlooked aspect: AVX-128**
  - Almost all SSE instructions re-encoded with 3 operands on 128 bits registers !
  - Including integers operations
    - “Keccak”, a.k.a. SHA-3: +25% performance !
  - 100% Compatible with AVX operation on 256 bits
    - No transition loss, unlike SSE<->AVS
SSE / AVX : Instructions (3/4)

AVX2 / FMA (introduced with « Haswell »)
- Integers operation on 256 bits wide vectors
- Add « gather » instruction
  - Load vector registers from non-contiguous memory
- Add arbitrary intra-registers permutations
- Add FMA instructions set: a*b+c and variation (a*b-c, ...)

Sometimes, names are confusing:
- Inside CPUID register identification:
  - Flag “AVX2” corresponds to 256 bits integers operations & “gather” support
  - Specific Flag “FMA” for FMA instructions
  - AVX2 is not FMA – they just showed up together!
SSE / AVX : Instructions (4/4)

AVX-512 (upcoming in Knights Landing “KNL”, “Skylake”)
- Double both register width and number of registers
- Masking instructions
  - Simplify the vectorization of conditional loop, etc.
- Some instructions support registers xmm16-xmm31 or ymm16-ymm31
  - `vpbroadcastd %xmm19, %zmm20`
- Add « scatter » operation
  - Write vector register to non consecutive memory addresses

Possible to emulate the instructions set

And now, possible to run native code on Xeon Phi 72xx ☑
Vectorization by the compiler (1/2)

Ideally:
- Inner loop obviously parallel
- No indirect addressing
- Aligned access as far as possible
- Enough loop iterations to justify overheads
  • And cost of remainder/tail loop when number of iterations not an integer multiple of the vector size
- Sequential/continuous write operation
- No/few conditional statements

Then Compiler can automatically vectorize for you:

fun.f90(78): (col. 10) remark: LOOP WAS VECTORIZED
Vectorization by the compiler (2/2)

► But also...
  – Sometimes, block-based vectorization
    • Bunch of operations merged together
  – Loops can automatically be rewritten
    • permutation, fusion, ...
  – Indirect memory access can produce vectorized code
    • If the compiler accepts the case and deems it cost-effective
    • Is facilitated by the gather instruction (for reading)
    • Is facilitated by the scatter instruction for writing (AVX-512 only)
  – Some directives can help the compiler in the analysis
    • ivdep, simd, vector, ...

► Masking instruction (KNC, AVX-512) will also help
  – In particular, combined with OpenMP4 “omp simd” directives
Vectorization and divergence

```c
#pragma omp simd reduction(+:...)
for (p=0; p<N; p++)
{
    // Blue work
    if(...)
    {
        // Green work
    }
    else
    {
        // Red work
    }
    while(...)
    {
        // Gold work
        // Purple work
    }
    y = foo (x);
    Pink work
}
```

©Intel
Vectorization by programmer, a simple example

- Simple case of a trivial loop
  1. Analyze dependencies
     • No dependency between successive iterations
     • No aliasing between A, B and C
        – Issue in C/C++, less so Fortran
  2. Unroll the loop (virtually) to explicitly express multiple operations
     • « i » is an **induction variable**
  3. Create the vector loop by merging similar operations

```plaintext
loop i in [0 : N]
{
    A[i] = B[i] + C[i]
}
```

```plaintext
loop i in [0 : N : 4]
{
    A[i] = B[i] + C[i]
    A[i+1] = B[i+1] + C[i+1]
}
```

```plaintext
loop i in [1 : N / 4]
{
    t1, t2, t3: SIMD_word
    simd_load(t2, B[i*4])
    simd_load(t3, C[i*4])
    simd_add(t1, t2, t3)
    simd_store(t1, A[i*4])
}
```
Identify the instructions set (1/3)

➤ Useful to check what the compiler did (statically in the binary) and what is really executed by the machine (i.e. using a debugger or a profiler or an emulator/simulator, etc.)
  – Use the “-S” option to the compiler to generate the assembly
  – Use a breakpoint in a function to check with a debugger

➤ The x87 FPU
  – Very old, very obsolete, purely scalar, stack-based
  – Useful to know to identify compilation issue in 64 bits
  – Most of x87 instructions use the hardware stack (%st) with an optional suffix to access inside the stack, i.e. %st(4)
    • fmul %st(4),%st (multiplication)
    • fxch %st(1) (exchange top element with another, very common)
  – Avoid this unit like the plague
    • Standard FPU in 32 bits mode
    • In 64 bits mode, the standard FPU is SSE/SSE2
Identify the instructions set (2/3)

- For vector operations (SSE, AVX, ...), two things have to be identified:
  1. Register width (128, 256, 512 bits)
  2. Instruction encoding (SSE, AVX aka VEX, AVX-512 aka EVEX)

- FP vector assembly instructions are based on a pattern:
  - An optional prefix "v" means AVX / AVX-512
  - A mnemonic showing operation type (mul, add, ...)
  - Two one-letter suffixes:
    1. « s » for scalar or « p » for packed (a.k.a. vector)
    2. « s » or « d » for simple or double precision

- Registers name show register width:
  - xmm : 128 bits register (and scalar ss/sd, using only 32/64 bits)
  - ymm : 256 bits register
  - zmm : 512 bits register

- AVX-512 instructions also can use an optional mask register %k
## Identify the instructions set (3/3)

<table>
<thead>
<tr>
<th>Assembly instruction</th>
<th>Meaning</th>
<th>Meanings</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>addsd %xmm0,%xmm1</code></td>
<td>SSE-encoded scalar</td>
<td></td>
</tr>
<tr>
<td><code>addpd %xmm0,%xmm1</code></td>
<td>SSE-encoded 128 bits (2x64)</td>
<td></td>
</tr>
<tr>
<td><code>vaddsd %xmm0,%xmm1,%xmm1</code></td>
<td>AVX-encoded scalar</td>
<td></td>
</tr>
<tr>
<td><code>vaddpd %xmm0,%xmm1,%xmm1</code></td>
<td>AVX-encoded 128 (2x64) bits</td>
<td></td>
</tr>
<tr>
<td><code>vaddpd %ymm0,%ymm1,%ymm1</code></td>
<td>AVX-encoded 256 (4x64) bits</td>
<td></td>
</tr>
<tr>
<td><code>vaddpd %zmm0,%zmm1,%zmm2</code></td>
<td>AVX512-encoded 512 (8x64) bits, unmasked</td>
<td></td>
</tr>
<tr>
<td><code>vaddpd %zmm0{%k1},%zmm1,%zmm2</code></td>
<td>AVX512-encoded 512 (8x64) bits, masked</td>
<td></td>
</tr>
<tr>
<td><code>vaddpd %ymm0{%k1},%ymm1,%ymm1</code></td>
<td>AVX512-encoded 512 (4x64) bits, masked [1]</td>
<td></td>
</tr>
<tr>
<td><code>vaddpd %xmm0,%xmm1,%xmm1</code></td>
<td>AVX512-encoded 128 (2x64) bits, unmasked [1] indistinguishable from AVX except by binary opcode? [2]</td>
<td></td>
</tr>
<tr>
<td><code>vaddsd %xmm0,%xmm1,%xmm1</code></td>
<td>AVX512-encoded scalar, unmasked indistinguishable from AVX except by binary opcode? [2]</td>
<td></td>
</tr>
</tbody>
</table>

[1] not available in Knights Landing  
[2] Mask register %k0 is always all-one
Vectorization & dependencies

- To be vectorized, a loop must be « vectorizable »
- The data from merged iterations must be “sufficiently” independent
  - So that it is legitimate to compute them simultaneously
- Pay attention to dependencies
  - Read-after-Write (RAW)
  - Write-after-Read (WAR)
  - Write-after-Write (WAW)
  - No issues for Read-after-Read (RAR), but they should be noted as they might prove useful when thinking about the cache hierarchy
    - RAR = reuse = temporal locality
- Simplest case:
  - Output arrays are not read
  - Indices in output arrays are linear functions of the induction variable
  - No aliasing between output arrays and input arrays/other output arrays
Vectorization & dependencies: RAW & WAR

for (i = 1 ; i < N ; i++) {
    A[i] = A[i-1]
}

▶ A[i] is written at step i
▶ A[i] is read at step i+1
▶ RAW dependency
  – i+1 comes after i in loop order

▶ In practice: the loop copies A[0] in N consecutive element

▶ This “broadcast” operation can be vectorized despite the RAW dependency

for (i = 0 ; i < N-1 ; i++) {
    A[i] = A[i+1]
}

▶ A[i] is written at step i
▶ A[i] is read at step i-1
▶ WAR dependency
  – i-1 comes before i in loop order

▶ In practice: the loop shifts all values in A[] by one element to the start

▶ Shifting operation can be vectorized
Vectorization & dependencies : WAW (1/2)

```c
for (i = 0 ; i < N ; i++) {
    A[B[i]] = C[i]
}
```

- Undetermined access to A[*] since values off B[i] unknown at compilation
- Potentially dependency WAW if B[] contains duplicate values
- If B[i] is an injection from [0-N[ to [size of A[, no dependency WAW
  - “Scatter” can exploit vectorization if no aliasing between A, B and C
    • Or if we have AVX512CD ❧
  - Compiler cannot automatically determine this property and will not vectorize
- Possible to vectorize with a weaker property
Vectorization & dependencies : WAW (2/2)

for (j = 0 ; j < N ; j+=V) {
    for (i = 0 ; i < V ; i++) { // loop vectorized by V
        int idx = i+j;
        A[B[idx]] = C[idx];
    }
}

▶ Outer loop « j » is executed in the original order
▶ Inner loop « i » is executed in vectorized mode
  – If loop operation can be vectorized...
▶ If B[j..j+V[ is an injection in A, inner loop could be vectorized
  – If two elements of B has the same value for different j, the original write order is respected
▶ Again, the compiler cannot prove the property
  – But as we’ll see later, that is the idea behind AVX512CD
Scatter / Gather (1/2)

- Assume \textbf{idx} is an index list in vector \textbf{x}
- Gather:

  \begin{verbatim}
  for(i=0; i<N; ++i)
  x[i]=y[idx[i]];
  \end{verbatim}

- Scatter:

  \begin{verbatim}
  for(i=0; i<N; ++i)
  y[idx[i]]=x[i];
  \end{verbatim}

- In some cases, it is possible to use specific instructions such as permutations if \textbf{idx} has known properties
  - Ex: \( \forall i \in [0, N - 1], idx(i + 1) - idx(i) \in [0,1] \)
    - Dense data loading and permutations
    - The compiler cannot know the properties
Scatter / Gather (2/2)

Scatter / Gather support in hardware
- Better than on current Haswell, or the loop in Knights Corner

- Can optionally be masked, so some values are not updated
Vectorization & caches

Revisiting the example about temporal locality

Example #1 (was good for locality)

- Inner loop of 5 iterations
  - Too small, not a multiple of the vector size
- Reduction sum in Y
  - Suboptimal

Example #2 (was bad for locality)

- Long loop of N-4 iterations
  - Good for vectorization if N is large
- Linear access to X
  - Easy to vectorize
- Coef[k] is constant inside the inner loop
  - Easy to vectorize
- No reduction
  - Easier to vectorize

---

#1
for (i = 2 ; i < N-2 ; i++) {
    Y[i] = 0.;
    for (k = 0 ; k < 5 ; k++)
        Y[i] += X[i-2+k]*coef[k];
}

#2
for (i = 2 ; i < N-2 ; i++)
    Y[i] = 0.;
for (k = 0 ; k < 5 ; k++) {
    for (i = 2 ; i < N-2 ; i++)
        Y[i] += X[i-2+k]*coef[k];
}
Vectorization & cache, a compromise

- Keep the inner loop “vectorizable”
  - To exploit the peak performance of the processor

- Keep the temporal locality as well
  - To avoid memory bandwidth becoming a bottleneck

- Split X in tiles of size M element
  - Choose the tile size so that the data in the innermost loop (X,Y, coef) fit in some level of cache memory (L1, L2, L3)

- Each tile will be efficiently processed by a vectorized loop

```cpp
#3
for (b = 2 ; b < N-2 ; b+=M) {
    for (i = b ; i < b+M ; i+)
        Y[i] = 0.;
    for (k = 0 ; k < 5 ; k+)
        for (i = b ; i < b+M ; i++)
            Y[i] += X[i-2+k]*coef[k];
}
```
Vectorization & data alignment (1/4)

▶ Data loaded from memory, in particular vector data, should be “naturally aligned”
  – Stored at an address an integer multiple of the data size
▶ For vector, that is aligned on a 16 (SSE/AVX-128), 32 (AVX) or 64 (KNC, AVX-512) boundary
▶ This is faster for the CPU, and guarantee the load (or store) does not “cross a cache line” and does not “cross a page boundary” in memory
  – Data that overlaps two cache lines or two memory pages are less efficient to load/store and creates more conflicts

1. Data should be properly aligned whenever possible

2. And the compiler should know that the data are properly aligned
Vectorization & data alignment (2/4)

1. Aligning the data

1. C, static
   ```c
   float A[1000] __attribute__((aligned(64)));
   ```
2. C, dynamic
   ```c
   posix_memalign(), memalign(), aligned_alloc(), __mm_malloc(), ...
   ```
3. Fortran, static
   ```fortran
   real :: A(1000)
   !dir$ attributes align: 64 :: A
   ```
4. Fortran, dynamic
   ```fortran
   real, allocatable :: a(:)
   !dir$ attributes align:64 :: a
   ```
Vectorization & data alignment (3/4)

2. Informing the compiler about the data (ICC, IFORT)
   1. C, static
      • Usually directly visible using the attributes
   2. C, dynamic
      • Can use attributes __assume_aligned & __assume

```c
void f (float * restrict a, int n) {
    int i;
    __assume_aligned(a, 64);
    __assume (n % 16 == 0);
}
```

3. Fortran, static
   • Usually directly visible using the attributes

4. Fortran, dynamic
   subroutine f(a,n)
   real*4 :: a(*)
   integer n
   !DIR$ ASSUME_ALIGNED A: 64
   !DIR$ ASSUME (mod(n,16) .eq. 0)
Vectorization & data alignment (4/4)

- The “vector aligned” directive (ICC, IFORT)
  - Applied to a loop, inform the compiler all accesses in the loop are aligned
  - Should be used carefully

- OpenMP 4 directives “simd” and “declare simd” contain an “aligned clause”
  - Similar to __assume_aligned but as a directive, not an attribute
Vectorization: compilation vs. exploitation

- Compilation: compiler succeeds in generating a vectorized version (or multiple versions) of the original code
  - Indicated by compiler report
  - Indicated by static binary analysis
    • Maqao, objdump, etc.

- Exploitation: This version (or some of those versions) is/are exploited during code execution
  - Constraints on data alignment, loop size, etc.
  - Pay attention to masking!
    • Vector instruction with all but one element disabled are no better than scalar
  - Metric: ratio between number of iterations executed in vector mode compared to total number of iterations
Vectorization & numerical accuracy

- SSE/AVX/AVX-512, KNC are IEEE 754-2008 compliant
  - And IEEE 754-2008 includes single-rounding fused-multiply-add for AVX2+
- For faster computations when denormals are not useful, use flush-to-zero (FTZ) or denormals-are-zero (DAZ)
- If some functions are sensitive and require less optimizations, when using GCC? it is possible to (not) optimize a subset of the code rather than the whole file / all the code:

```c
#pragma GCC optimize("-O3")
#pragma GCC target("arch=haswell,tune=haswell")
void f2(const int n, float *restrict c, const float *restrict a, const float *restrict b) {
    int i;
    for (i = 0 ; i < n ; i++)
        c[i] = a[i] + b[i];
}
#pragma GCC reset_options
```


Thanks

For more information please contact:

M+ 33 6 40 19 64 42
romain.dolbeau@atos.net

Atos, the Atos logo, Atos Consulting, Atos Worldgrid, Worldline, BlueKiwi, Bull, Canopy the Open Cloud Company, Yunano, Zero Email, Zero Email Certified and The Zero Email Company are registered trademarks of the Atos group. September 2016. © 2016 Atos. Confidential information owned by Atos, to be used by the recipient only. This document, or any part of it, may not be reproduced, copied, circulated and/or distributed nor quoted without prior written approval from Atos.

01-10-2016