

# **Performance Tuning for CPU**

## Part 3: Memory Optimizations

Marat Dukhan

## Example 0: List Traversal

```
size_t traverse_list(  
    const list_node* list)  
{  
    size_t list_length = 0;  
    while (list != 0) {  
        list = list->next;  
        list_length += 1;  
    }  
    return list_length;  
}
```

## Example 0: List Traversal

```
size_t traverse_list(  
    const list_node* list)  
{  
    size_t list_length = 0;  
    while (list != 0)  
        list = list->next;  
    list_length += 1;  
}  
return list_length;  
}
```

**O(N)?**

## Example 0: List Traversal

```
size_t traverse_list(  
    const list_node* list)
```

```
{
```

```
    size_t list_length = 0;
```

```
    while (list != 0) {
```

```
        list = list->next;
```

```
        list_length++;
```

```
    }
```

```
    return list_length;
```

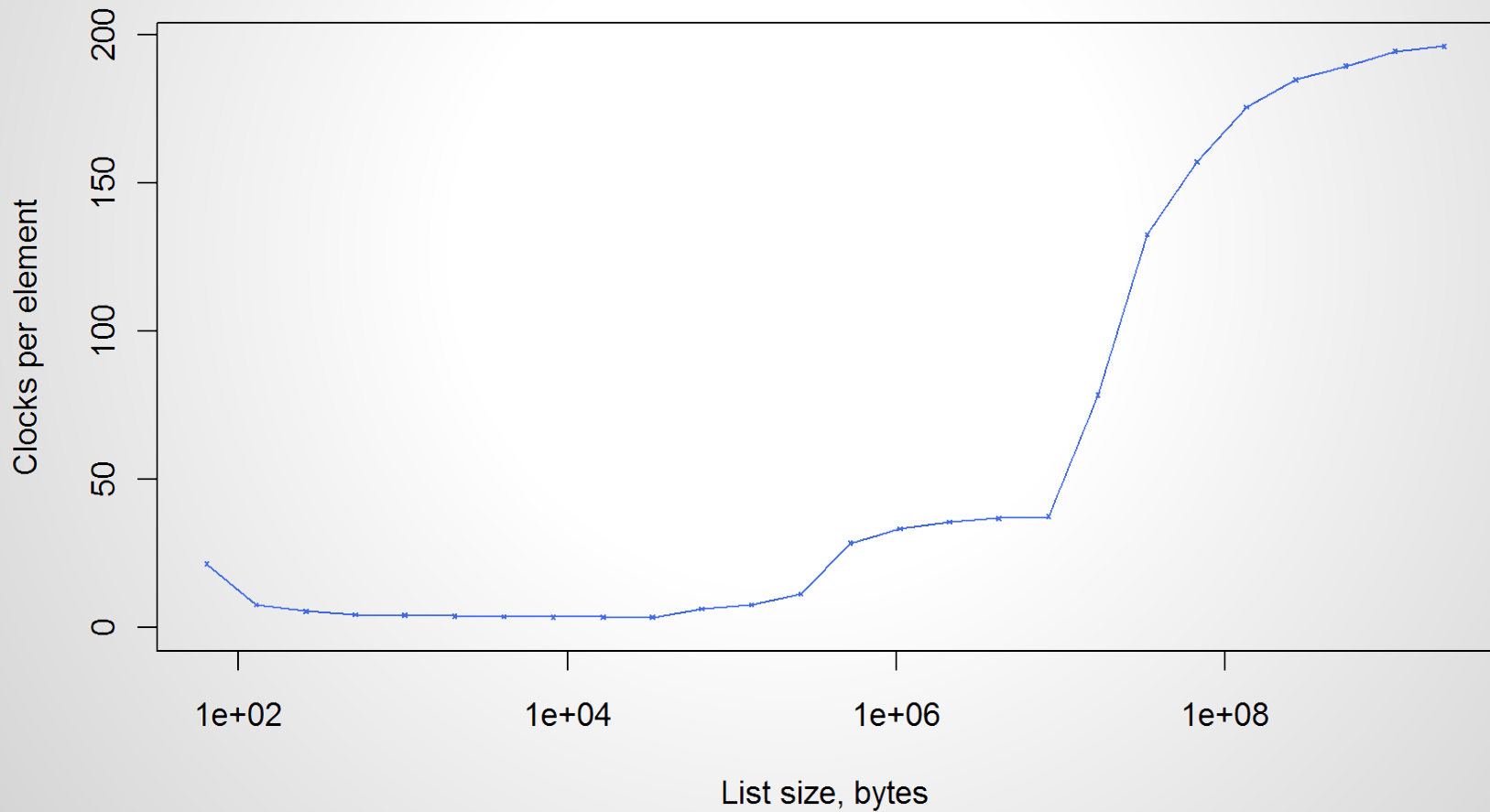
```
}
```

$O(N)$ ?

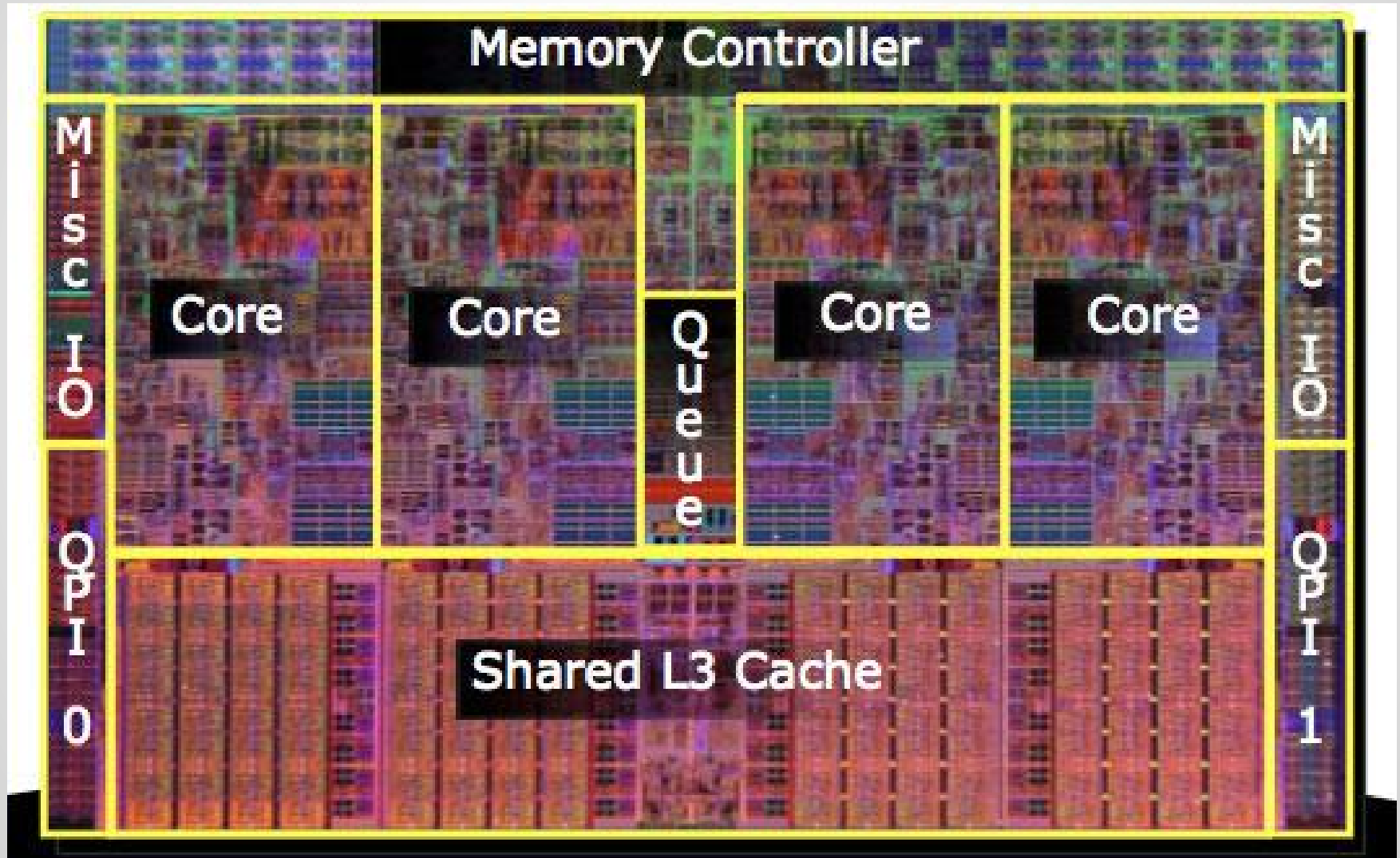
$O(\text{anything})!$

# Example 0: List Traversal

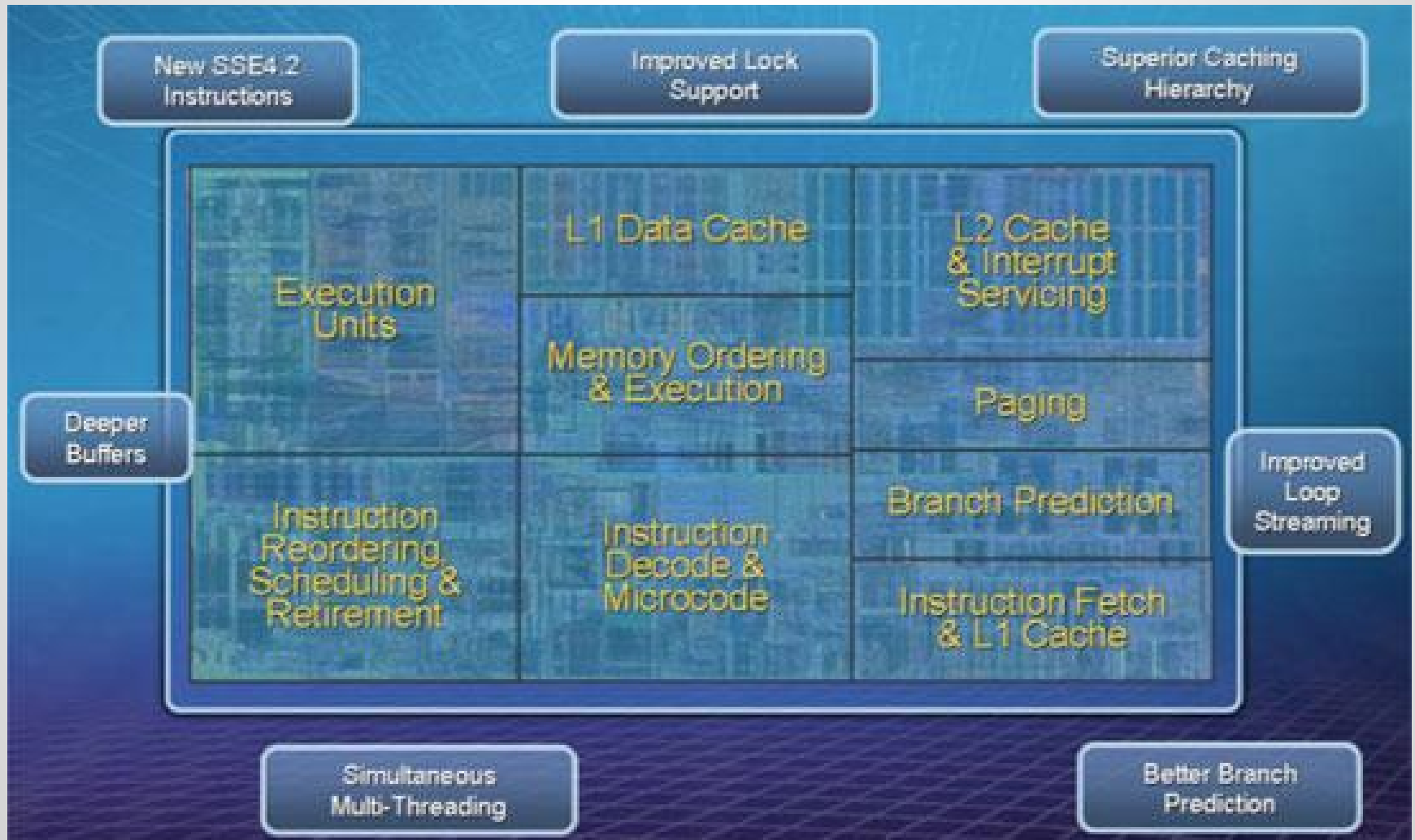
Traversing a random list



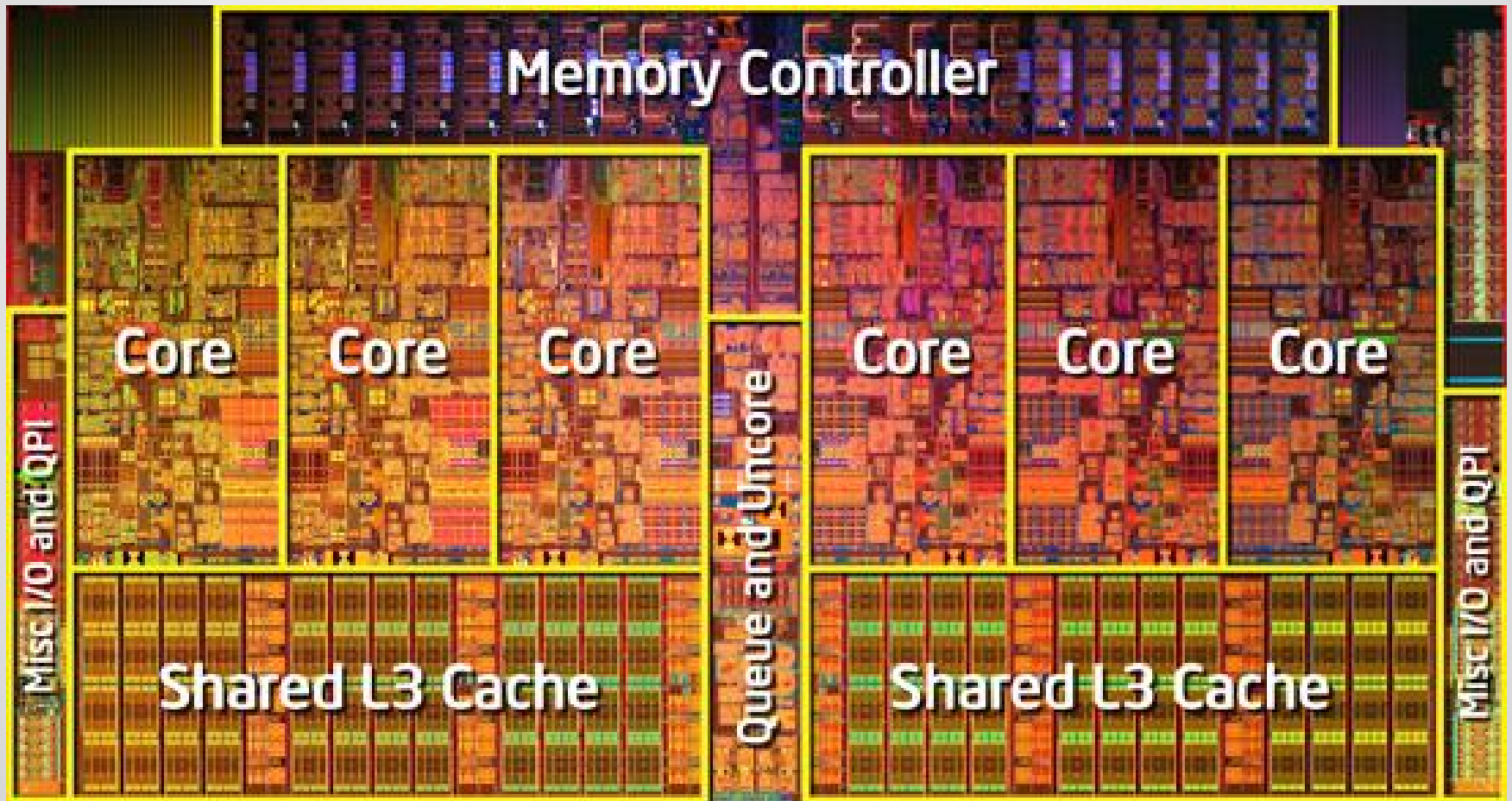
# Memory Subsystem: Overview



# Memory Subsystem: Overview



# Memory Subsystem: Overview





# Cache Hierarchy: Overview

Three levels of cache

- Level 1 (the fastest, per core)
- Level 2 (per core)
- Level 3 (aka Last-Level Cache in Intel docs, shared between all cores)



# Cache Hierarchy: Level 1

- Separate data and instruction caches
- L1 data cache:
  - Almost as fast as registers
  - The lowest latency of all caches
  - The highest bandwidth
  - Throughput: at least 1 load per cycle
    - Often more
      - 1 load + 1 store on Nehalem
  - Latency: 3-4 cycles
- L1 instruction cache:
  - Intended only for machine code (not data)
    - Read-only
    - Do not mix code and data in the same section

# Cache Hierarchy: Performance

Remember latency and throughput

- Latency = how long to wait for the result
- Throughput = how much work per second or CPU cycle can be done

Back to the memory hierarchy

- Latency = the number of cycles (or nanoseconds) to bring the requested data
- Throughput = how many megabytes per second can be read or written

# Cache Hierarchy: Latency

Latency always increases from lower-level cache to RAM. E.g. on Nehalem:

- 4 cycles for L1 cache
- 10 cycles for L2 cache
- 17 cycles for L3 cache
- 198 (!) cycles for RAM

# Cache Hierarchy: Throughput

Throughput decreases from lower-level cache to RAM. However, the changes are not as sharp as with latency. E.g. on Nehalem:

- 32 bytes/cycle for L1
  - 1 16-byte read + 1 16-byte write
- 32 bytes/cycle for L2
  - On average
  - Delivers 64 bytes every other cycle
- Unknown for L3
  - 1 load + 1 store every **cache** cycle
    - Shared between all cores!
  - Runs at different frequency than CPU cores

# Cache Hierarchy: Throughput

Lavalys EVEREST Cache & Memory Benchmark

|  | Read  | Write          | Copy        | Latency |
|--|---|----------------|-------------|---------|
| Memory   | 14253 MB/s  | 9043 MB/s      | 13247 MB/s  | 57.5 ns |
| L1 Cache   | 53186 MB/s  | 53166 MB/s     | 106281 MB/s | 1.2 ns  |
| L2 Cache   | 35392 MB/s  | 31547 MB/s     | 41007 MB/s  | 3.0 ns  |
| L3 Cache   | 19333 MB/s  | 12167 MB/s     | 18079 MB/s  | 4.8 ns  |
| CPU Type   | HexaCore Intel Core i7 Extreme 980X (Gulftown, LGA1366) |                |             |         |
| CPU Clock  | 3324.9 MHz (original: 3333 MHz)                         |                |             |         |
| CPU FSB  | 133.0 MHz (original: 133 MHz)                           |                |             |         |
| CPU Multiplier   | 25x   | CPU Stepping   |             | B0      |
| Memory Bus   | 798.0 MHz   | DRAM:FSB Ratio |             | 6:1     |
| Memory Type  | Triple Channel DDR3-1600 SDRAM (9-9-9-24 CR1)           |                |             |         |
| Chipset  | Intel Tylersburg X58, Intel Westmere                    |                |             |         |
| Motherboard  | Gigabyte GA-X58A-UD5                                    |                |             |         |
| EVEREST v5.30.2054 Beta / BenchDLL 2.4.273.0 (c) 2003-2010 Lavalys, Inc. |   |                |             |         |

Save

Start Benchmark

Close

Image from [http://www.xbitlabs.com/articles/cpu/display/intel-core-i7-980x\\_3.html](http://www.xbitlabs.com/articles/cpu/display/intel-core-i7-980x_3.html)

# Cache Blocking

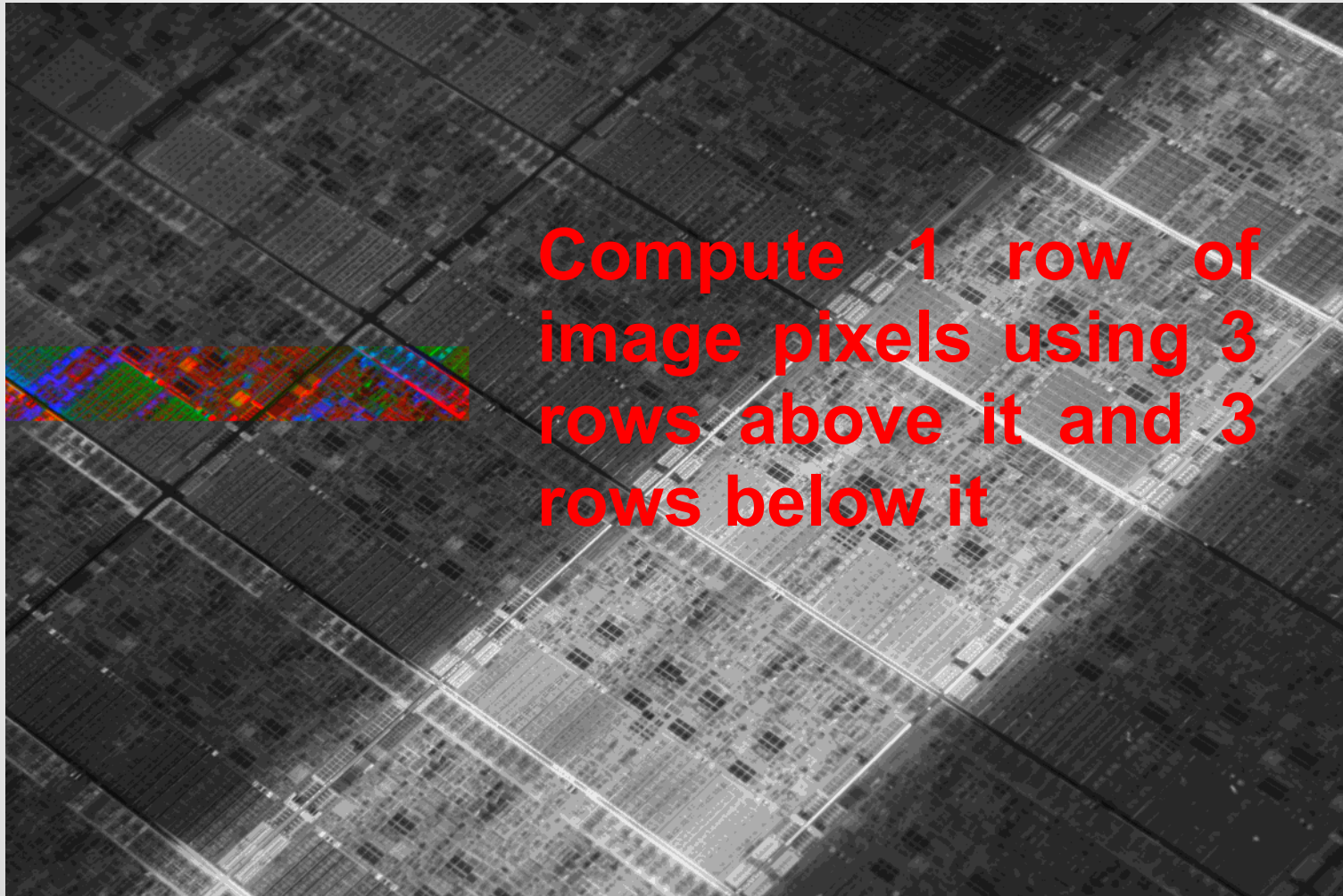
Example: Image processing

- Input: 7680 x 4320 pixels image (8K UHDTV)
- Operation: 7-point 1D Gaussian blur

$$\begin{aligned} \circ \quad a[i][j] &= a[i-3][j] * c_{-3} \\ &+ a[i-2][j] * c_{-2} \\ &+ a[i-1][j] * c_{-1} \\ &+ a[i][j] * c_0 \\ &+ a[i+1][j] * c_1 \\ &+ a[i+2][j] * c_2 \\ &+ a[i+3][j] * c_3 \end{aligned}$$



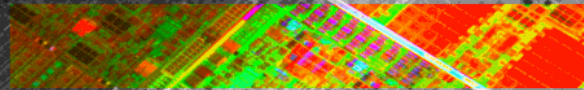
# Example 1: Cache Blocking





# Example 1: Cache Blocking

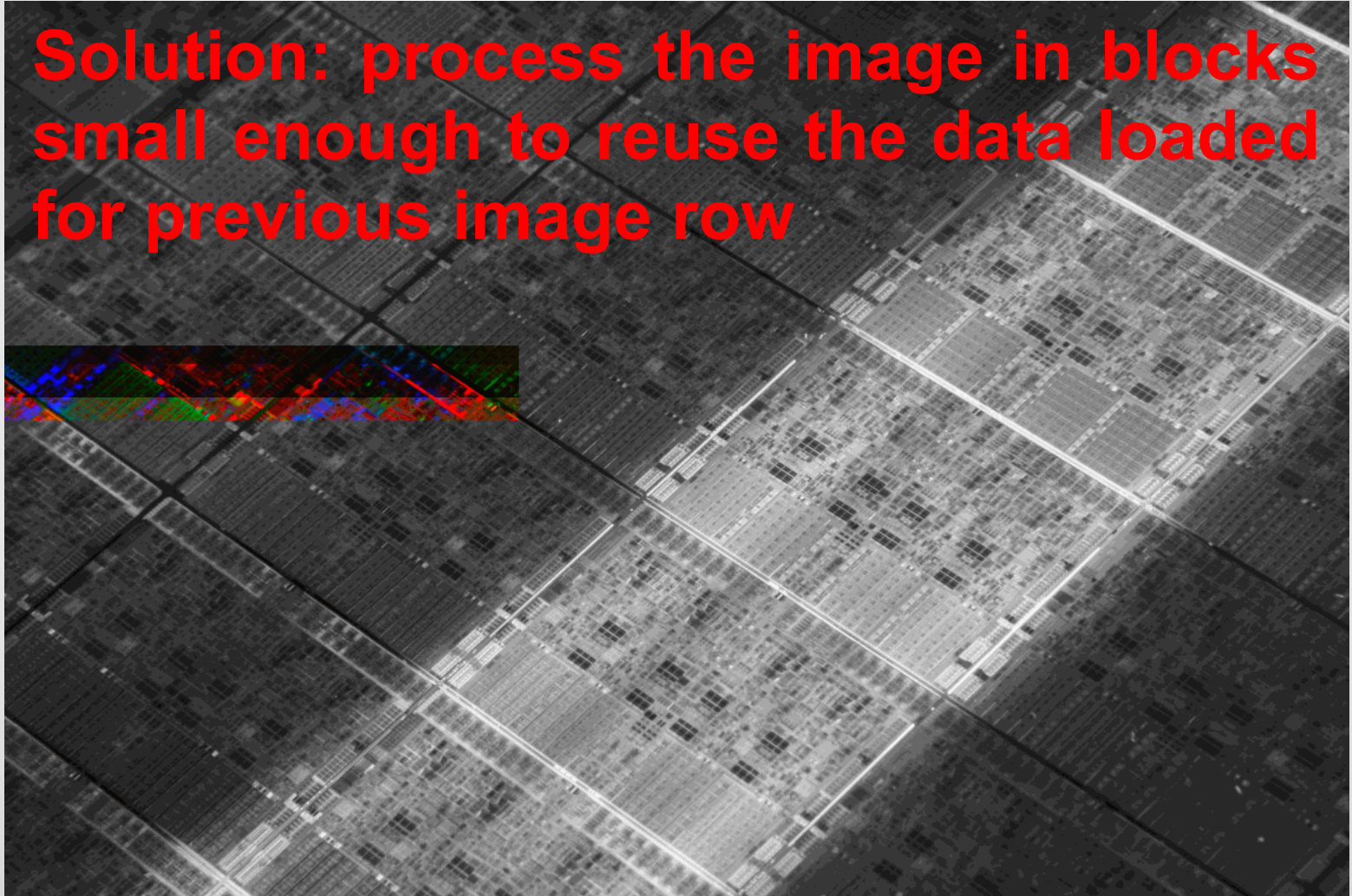
As the processing goes on through the image, the faraway parts of the image lines are evicted from the cache



When we will process the next line, we will have to load them again

# Example 1: Cache Blocking

**Solution: process the image in blocks small enough to reuse the data loaded for previous image row**



# Cache Associativity

Associativity = the maximum number of independent arrays which are guaranteed to not interfere with each other

- If you work with more independent arrays than ways of associativity, they can evict each other from the cache
  - Consider about any array pointer as independent
    - E.g. pointers to different lines of an image
- Direct mapped cache = 1 way of associativity
- Fully associative means  $(\text{cache size} / \text{cache line size})$  ways of associativity

# Cache Associativity

Nehalem (and other server processors from Intel) has quite high associativity:

- 8-way for L1D cache
- 8-way for L2 cache
- Variable associativity for L3 cache
  - Usually 16 to 20-way
  - 16-way on Jinx

# Cache Line

Cache Line is a quant of cache

- The minimum amount of cache which can be loaded or stored to memory
- 64 bytes in all modern x86 CPUs
- 32 bytes in some ARM CPUs
  - E.g. ARM Cortex-A5, Cortex-A9, Qualcomm Scorpion

# Cache Line: Memory writes

What happens if your program stores 1 byte?

1. 64-byte cache line will be loaded from RAM.
2. 1 byte will be written to the cache line.
3. When the cache line will be evicted from the cache, the 64 bytes will be written to the memory.

**64 bytes were read and 64 bytes written to memory to store just 1 byte**

**Random writes are extremely expensive**



# Cache Line: RFO

Only one core can write to a cache line

- The "owner" of the cache line
- What happens if some core wants to write to the same cache line?
  - The current owner of the cache evicts the cache line from its cache
    - If it was modified, it is stored back to memory
  - The core which initiated the operation obtains exclusive ownership of the cache line
  - Now it can write to the cache line
- What if two cores contend for a cache line?

# Example 2.1: RFO

```
int a;  
int b;  
  
void thread1_proc() {  
    for (...)  
        a = calc(a);  
}  
  
void thread2_proc() {  
    for (...)  
        b = calc(b);  
}
```



## Example 2.1: RFO

```
int a __attribute__((aligned(64))) ;
int b __attribute__((aligned(64))) ;

void thread1_proc() {
    for (...)
        a = calc(a) ;
}

void thread2_proc() {
    for (...)
        b = calc(b) ;
}
```

## Example 2.1: RFO

```
int results[2];
```

```
void thread0_proc() {  
    const int threadId = 0;  
    for (...)  
        results[threadId] = calc();  
}
```

```
void thread1_proc() {  
    const int threadId = 1;  
    for (...)  
        results[threadId] = calc();  
}
```

# Line Fill Buffers

When a load, store, or prefetch instruction incurs an L1 cache miss, a Line Fill Buffer is allocated

- Line Fill Buffer accumulates parts of the cache line while they are coming from other levels of memory hierarchy
- The number of Line Fill Buffers limits how many cache misses the CPU may handle concurrently
  - Nehalem has 10 LFBs

# Preloading and Prefetching

We may cause a cache miss in advance to guarantee that the data is in cache by the time we need it. Three ways to do it:

- Use special prefetch instructions
  - May be ignored by processor
- Load one element in a batch ahead of others
  - May stall the pipeline
- Relax and let the CPU do it for you
  - Not guaranteed to work

# Example 3.0: Vector sum

```
double vector_sum(const double *data, size_t length) {  
    double sum0 = 0.0, sum1 = 0.0;  
    double sum2 = 0.0, sum3 = 0.0;  
    for (; length >= 4; length -= 4) {  
        sum0 += data[0];  
        sum1 += data[1];  
        sum2 += data[2];  
        sum3 += data[3];  
        data += 4;  
    }  
    ... process last elements ...  
    return sum0 + sum1 + sum2 + sum3;  
}
```

# Example 3.1: Preloading

```
double vector_sum(const double *data, size_t length) {  
    double sum0 = 0.0, sum1 = 0.0;  
    double sum2 = 0.0, sum3 = 0.0;  
    ... process first elements ...  
    for (; length > 4 * N; length -= 4) {  
        sum0 += data[4 * N];  
        sum1 += data[1];  
        sum2 += data[2];  
        sum3 += data[3];  
        data += 4;  
    }  
    ... process last elements ...  
    return sum0 + sum1 + sum2 + sum3;  
}
```

## Example 3.2: Prefetching

```
double vector_sum(const double *data, size_t length) {
    double sum0 = 0.0, sum1 = 0.0;
    double sum2 = 0.0, sum3 = 0.0;
    for (; length >= 4; length -= 4) {
        _mm_prefetch((char*)&data[4 * N], _MM_HINT_T0);
        sum0 += data[0];
        sum1 += data[1];
        sum2 += data[2];
        sum3 += data[3];
        data += 4;
    }
    ... process last elements ...
    return sum0 + sum1 + sum2 + sum3;
}
```

## Example 3.2: Prefetching

**void** **\_mm\_prefetch**(**char\*** **address**, **int** **hint**);

- **address** specifies the memory location to be loaded to memory
  - The whole cache line will be loaded
- **hint** specifies the level of cache hierarchy
  - **\_MM\_HINT\_T0**, **\_MM\_HINT\_T1**, **\_MM\_HINT\_T2** correspond to L1, L2, and L3 caches
  - **\_MM\_HINT\_NTA** means load data into L1 cache and mark as the first candidate to be evicted
  - Practice: use **\_MM\_HINT\_T0** and don't care
- Nehalem supports 4 outstanding **\_mm\_prefetch** instructions



# Hardware Prefetching

Nehalem uses two algorithms to automatically prefetch to L1 cache

- Simple algorithm: monitor increasing memory access addresses
- Complex algorithm: uses a buffer of 256 entries
  - Each entry remembers the address of memory read instruction, its previous memory read address, and read sequence stride (up to  $\pm 2048$ )
  - On every read the algorithm tries to find the pattern
    - If succeeds, prefetches the next portion

# Hardware Prefetching

Nehalem uses two algorithms to automatically prefetch to L2 and K3 caches

- Simple algorithm: load adjacent cache line
  - The two cache lines form an aligned 128-bit chunk
- Complex algorithm: track the requests from lower-level caches
  - Tries to find a sequence of requests with the same stride
  - For L2 it also maintains a 2-bit counter of the usefulness of prefetched data:
    - Continues to prefetch if it useful 2/3 times.

# Compiler prefetching

GCC has built-in prefetch intrinsic:

- **void \_\_builtin\_prefetch(const void\* address)**
  - Available on all architectures

With **-fprefetch-loop-arrays** option GCC tries to insert prefetch instructions itself

- This option works better than auto-parallelization, but still slightly worse than handwritten code

# Pages: Overview

The memory is divided into pages

- Page is the quant of memory on system level
  - Can not allocate less than one page
  - Memory attributes are set for individual pages
    - I.e. read, write, execute
- Page maps to physical memory
  - Sometimes also to devices or disk files
- Modern processor architectures support pages of different sizes
- On x86 the default page size is **4 kilobytes**.

# Pages: Page Descriptors

Pages are described by a page descriptor

Page descriptor keeps information about

- The physical address of the page
- Page attributes
- No page descriptor = no memory page
  - I.e. memory for this page was not allocated

# Pages: Page Descriptors

Each time your code accesses memory, the processor looks up page descriptor for the accesses page

- If no page descriptor found, it raises hardware exception
  - System can catch and process the exception
- If page descriptor found, but access is of wrong type, it raises hardware exception
- If none of the above occurs, the memory operation succeeds
  - The physical memory accesses is determined by the page descriptor

## Example 4.0: strlen

```
size_t strlen(const char *str) {  
    const char *cur = str;  
    while (*cur != 0) {  
        cur += 1;  
    };  
    return size_t(cur - str);  
}
```

# Example 4.1: SSE strlen

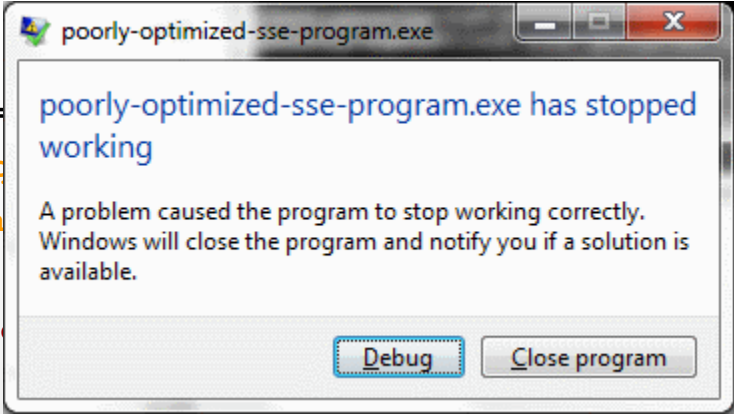
```
size_t strlen(const char *str) {
    const __m128i *cur = (const __m128i*)str;
    __m128i str_vector = _mm_loadu_si128(cur);
    int str_bitmask = _mm_movemask_epi8(
        _mm_cmpeq_epi8(str_vector, _mm_setzero_si128()));
    unsigned long index;
    _BitScanForward(&index, str_bitmask);
    while (str_bitmask == 0) {
        cur += 1;
        str_vector = _mm_loadu_si128(cur);
        str_bitmask = _mm_movemask_epi8(
            _mm_cmpeq_epi8(str_vector, _mm_setzero_si128()));
        _BitScanForward(&index, str_bitmask);
    }
    return size_t((((const char*)cur) + index) - str);
}
```



# Example 4.1: SSE strlen

```
size_t strlen(const char *str) {  
    const __m128i *cur = (const __m128i*)str;  
    __m128i str_vector = _mm_loadu_si128(cur);  
    int str_bitmask = _mm_movemask_epi8(  
        _mm_cmpeq_epi8(str_vector, _mm_setzero_si128()));  
    unsigned long index;  
    _BitScanForward(&index, str_bitmask);  
    while (str_bitmask == 0) {  
        cur += 1;  
        str_vector =  
        str_bitmask =  
        _mm_cmpeq_epi8(str_vector, _mm_setzero_si128());  
        _BitScanForward(&index, str_bitmask);  
    }  
    return size_t(index);  
}
```

**Suddenly...**



poorly-optimized-sse-program.exe has stopped working

A problem caused the program to stop working correctly. Windows will close the program and notify you if a solution is available.

Debug Close program

# Example 4.2: SSE strlen

```
size_t strlen(const char *str) {
    static const __m128i str_mask[16] = {
        _mm_set_epi32(0x00000000, 0x00000000, 0x00000000, 0x00000000),
        _mm_set_epi32(0xFF000000, 0x00000000, 0x00000000, 0x00000000),
        ...
        _mm_set_epi32(0xFFFFFFFF, 0xFFFFFFFF, 0xFFFFFFFF, 0xFFFFFFFF00)
    };

    const size_t str_misalignment = ((size_t)str) % ((size_t)16);
    const __m128i *str_aligned = (const __m128i*)((size_t)str - str_misalignment);
    __m128i str_vector = _mm_load_si128(str_aligned);
    str_vector = _mm_or_si128(str_vector, str_mask[str_misalignment]);
    int str_bitmask = _mm_movemask_epi8(_mm_cmpeq_epi8(str_vector, _mm_setzero_si128()));
    unsigned long index;
    _BitScanForward(&index, str_bitmask);
    while (str_bitmask == 0) {
        str_aligned += 1;
        str_vector = _mm_load_si128(str_aligned);
        str_bitmask = _mm_movemask_epi8(_mm_cmpeq_epi8(str_vector, _mm_setzero_si128()));
        _BitScanForward(&index, str_bitmask);
    }
    return size_t((((const char*)str_aligned) + index) - str);
}
```

# Pages: TLB

TLB = Translation Lookaside Buffer

- Cache for page descriptors
- Often multilevel
- Often separate code and data TLB
- Nehalem:
  - L1 DTLB: 64 entries, 4-way set-associative
  - L1 ITLB: 128 entries, 4-way set-associative
  - L2 TLB: 512 entries, 4-way set-associative
- If TLB miss occurs, processor looks up page descriptor in memory subsystem
  - Page descriptors can be cached themselves

# Pages: Huge TLB

x86 processors additionally support large pages of 2 MB or 1 GB size

- Good idea jeopardized by OS devs
- No interface is provided for allocating 1 GB pages on Windows and Linux
- For 2 MB cache the interface implies that all large pages must be allocated on startup
  - Helps to solve memory fragmentation issues, but renders large pages useless in many use-cases

# NUMA

In systems with several Nehalem processors (e.g. Jinx nodes) different parts of memory are physically attached to different processors

- Each processor has local memory (which it can access directly) and remote memory (which is only accessible through other processors)
- Functions exist to control NUMA allocations
  - [VirtualAllocExNuma](#) (on Windows)
  - [set\\_mempolicy](#) and [mbind](#) (on Linux)

# Bypassing the cache

Remember the problem with writing to memory

- Normally all memory writes operate on cache
  - A line of cache will be loaded from memory
  - Store instruction will write into this line of cache
- Even if we want to write the whole 64 bytes, they will first be loaded into cache
  - Effectively halves the available memory bandwidth for write operations
- Solution: special instructions which write bypassing the cache
  - Introduced in SSE

# Bypassing the cache

- `_mm_stream_si128`
  - Like `_mm_store_si128`
- `_mm_stream_pd`
  - Like `_mm_store_pd`
- `_mm_stream_ps`
  - Like `_mm_store_ps`
- Few more `_mm_stream_xx` intrinsics
- Require an aligned address (!)

# Bypassing the cache

`_mm_stream_xx` intrinsics write to special write-combining buffers

- When the buffers are filled (64 byte), they are disposed directly to memory
- `_mm_sfence` intrinsic disposes the buffers
  - Required before you access the stored data with other instructions
- On Nehalem the write-combining buffers are physically the same as line fill buffers
  - Max 10 output streams



## Example 5.0: memset

```
void* memset(void *ptr, int value,
size_t num)
{
    const char *cur = ptr;
    for (size_t i = 0; i < num; i++) {
        cur[i] = value;
    };
    return ptr;
}
```

## Example 5.1: SSE memset

```
void* memset(void *ptr, int value, size_t num)
{
    const char *cur = ptr;
    ... align pointer on 16 bytes ...
    __m128i mm_value = _mm_set1_epi8(value);
    for (size_t i = 0; i < num; i += 16) {
        _mm_stream_si128((__m128i*)cur, mm_value);
        cur += 16;
    };
    _mm_sfence();
    ... process the remainder ...
}
```

# Memory Optimization: Resources

- Ulrich Drepper: "What Every Programmer Should Know About Memory"
  - Available: [akkadia.org/drepper/cpumemory.pdf](http://akkadia.org/drepper/cpumemory.pdf)
- Kris Kaspersky: "Code Optimization: Efficient Memory Usage"

# Performance Testing: Issues

- Core migration
  - Bind your main thread to a system thread
- Hyperthreading
  - Disable in BIOS
- Turboboost
  - Disable in BIOS
- OS interrupts
  - Set high priority for your thread and program

# Performance Testing: Tradeoffs

## Median vs Min

- Run your code multiple times
- If your code has chance not to be interrupted by system or hardware events, use min
- If interruptions are inevitable, use median

## CPU clocks vs real time

- If you only work with data in L1 and L2 caches, use CPU ticks
- If you work with RAM, network, hard drive, use real time

# Performance Testing: Setup

Recreate the same experimental environment each time before you time the code

- Caches
  - Either flush or fill with useful data
- CPU power mode
  - Warmup might be needed
  - A better solution: disable all power-saving features
- Branch predictor history
  - Run some other branchy code
    - E.g. `std::sort` on various random data
  - Create a code consisting of jump instructions only, and feed to the CPU

# Questions

