Highly efficient screening for point-like targets via concentric shells

Jan Wassenberg
Fraunhofer IOSB
jan.wassenberg@iosb.fraunhofer.de

Wolfgang Middelmann
Fraunhofer IOSB

Peter Sanders
KIT

September 8, 2010

We consider the problem of automatically screening for man-made objects (MMO) in infrared (IR) videos and synthetic aperture radar (SAR) imagery. Since hard targets are often highly reflective in SAR and also have an IR signature that differs from their surroundings, both problems reduce to finding point-like objects. Thresholding (usually locally adaptive) only utilizes the radiometric information and ignores the maximum target size, which means reflection artifacts or large regions are often returned as false alarms. Recently, a level-set approach has been proposed that takes speckle into account and reliably separates targets from the background [1]. However, its computational cost is almost certainly too high for large datasets or real-time video analysis. An alternative model called the "hotspot transform" was developed for IR Search and Track applications [2]. This operator (defined in Sec. 1) searches for local maxima that are entirely surrounded by a ring of darker pixels, thus suppressing bright but non-point-shaped regions. Its computational cost for N pixels and maximum target size R is $O(N \cdot R \cdot R)$. We believe this technique to be suitable for screening in both IR and SAR data and have developed a novel algorithm that reduces its complexity to the optimal $O(N \cdot R)$. Our sophisticated implementation, described in Sec. 2, avoids redundant computations with a divide and conquer scheme, ensures the working set fits in caches via pipelining, and achieves an additional 27-fold speedup via vectorization and parallelization. The attained processing rate of 72 MPixel/s on a single workstation enables screening entire satellite datasets within seconds (c.f. Sec. 3). Results are given for airborne SAR images and the MSTAR dataset in Sec. 4. The algorithm is shown to be suitable for detection of MMO and as a pre-processing step for multi-class target recognition via support vector machine (SVM).

1 Hotspot Operator

The hotspot operator for extracting point-like regions and suppressing background pixels was introduced in [2]. Since the point texture and shape are generally highly variable, template-based pattern matching cannot be applied. Instead our model considers interest points to be pixels that are (without loss of generality) brighter than their surroundings. With the point size unknown (bounded only by a maximum), we consider multiple neighborhoods of concentric 'shells'

$$S(x_c, y_c, r) = \{\mathcal{I}(y, x) \mid \|(x_c, y_c) - (x, y)\|_\infty = r\}$$

Their maximum pixel values are compared with the central pixel $\mathcal{I}(y_c, x_c)$. Negative differences indicate the pixel is surrounded by uniformly darker pixels, thus attesting to a point region within that shell. The hotspot filter is defined by the largest of these values for all shells up to a maximum radius R (clamping negative values to zero):

$$\mathsf{minMax}(x_c, y_c) = \min_{r=1..R} \max S(x_c, y_c, r)$$

$$\mathsf{hotspot}(x_c, y_c) = \max\left[\mathcal{I}(y_c, x_c) - \mathsf{minMax}(x_c, y_c), 0\right]$$

This operator suppresses background pixels and thus enhances freestanding point-like regions as desired. It is simple and intuitive, requiring no parameters other than R, which is defined by the target size and sensor resolution. Unfortunately a naive implementation has complexity proportional to R^2. A first improvement takes advantage of a property of the minimum and clamping operations shown in Lemma 1:

$$\begin{aligned} \exists b \in S(x_c, y_c, r) > \mathcal{I}(y_c, x_c) \Rightarrow \\ \mathsf{hotspot}(x_c, y_c) = 0 \vee \\ \mathsf{minMax}(x_c, y_c) < b \leq \max S(x_c, y_c, r) \end{aligned} \tag{1}$$

If a shell contains a pixel brighter than the central pixel, then it will not affect the hotspot value and the rest of its pixels can be skipped. This optimization yields a measured (data-dependent) speedup of about 18 versus a naive implementation. While the worst-case quadratic complexity remains unchanged, it is difficult to construct such inputs and they will certainly not be encountered in practice. A drawback of this algorithm is that it cannot make effective use of vectorization due to its reliance on conditional branches.[1]

2 Improved Algorithm

We will now build upon related theoretical work to engineer a new and improved algorithm for computing hotspots.

Recall the computation of the maximum of the $8 \cdot r$ pixels that constitute a shell of radius r. By maintaining a transposed copy of the image, this operation reduces to

[1] Accumulating shell maxima via 16-way SIMD only resulted in a speedup of two due to unaligned memory access penalties and the overhead of copying ranges into registers.

four "Range Maximum Queries" $\mathsf{RMQ}(i,j) = \max_{k=i\ldots j} A[k]$ in an array or image row/column A. Alon and Schieber have shown that such queries (generalizable to any semigroup) can be answered in $O(1)$ time after $O(n \log n)$ preprocessing [3]. The hotspot operator's complexity is therefore bounded by $O(n \log n + n \cdot R)$, a significant improvement versus the previous algorithm's $O(n \cdot R^2)$ cost.

We refer to [4] for a complete presentation of the RMQ algorithm. The basic idea is to pre-calculate the maxima of power-of-two intervals. Each query can be split into two such intervals; the result is the larger of the two maxima. Katriel et al. suggest an efficient scheme for preprocessing that computes prefix and suffix maxima and interleaves them into a single array [5]. This only requires $O(n \log R)$ preprocessing time and space, since the query lengths are bounded by $2 \cdot R + 1$. Bender and Farach-Colton also describe an scheme that first divides the input array into blocks of size $O(\log n)$ [4]. While reducing the preprocessing time to $O(n)$, this comes at the price of more complicated queries with separate handling of inter- or intra-block queries. Fischer and Heun have recently introduced a similar succinct algorithm with optimal space requirements [6], but its queries are also too expensive in practice.

A disadvantage shared by all of these RMQ-based approaches is their mediocre locality – both interval length and the query indices affect the location of the preprocessed value, which makes for non-sequential accesses. One alternative would be to cast the hotspot operator as a stencil computation, maintaining four separate maximum accumulators for overlapping left, right, up, down intervals. Hotspot values would be computed as the maximum of these shell components, thus achieving the desired and optimal complexity of $O(n \cdot R)$. A disadvantage of this method lies in its high space requirements.

To bridge the gap between the redundant calculations of the existing method and the practical costs of theoretically motivated approaches, we have engineered a new algorithm that combines ideas from RMQ and stencil computation. The first key change is to store only a single set of row- and column interval maxima. These are used to generate all shells of a certain range of sizes and are then combined in-place to yield intervals of twice the length. Besides folding preprocessing into the main algorithm and reducing memory use, this also improves locality. The second important step is to organize the algorithm as a pipeline such that the working set fits entirely into common L2 caches. We iterate over image rows exactly once; starting from the current row, previously calculated interval maxima of successively increasing lengths are used to compute the shells for previous rows. The resulting tentative shell maxima are accumulated into the output buffer. Since only the last $4 \cdot R + 2$ rows are accessed, a cache of that size can entirely absorb the cost of repeated accesses. The algorithm is described by the following pseudocode:

Algorithm 1: Hotspot ($\mathcal{I} \mapsto \mathcal{H}$)

for (x, y) **do** $\mathsf{minMax}\,[y, x] := \infty$;
`MinMaxima`($\mathcal{I}$);
for (x, y) **do**
  $\mathcal{H}\,[y, x] := \max(\mathcal{I}\,[y, x] - \mathsf{minMax}\,[y, x]\,, 0)$;

Algorithm 2: MinMaxima

```
// Compute length 2 interval maxima
RM := I, CM := I;
for y := 1 to height do CombineIntervalMaxima(y, 1);
// Pipelined iteration over rows
for wavefront := 1 to height do
    row := wavefront;
    for L := 1 to ⌈log2 R⌉ do
        IL := 2^L // intervalLength
        for x := 1 to width do ShellMinMaxima ((row, x), IL)
        oldestRow := row − IL/2;
        CombineIntervalMaxima(oldestRow, IL);
        row := oldestRow − IL · 2;
```

Algorithm 3: ShellMinMaxima

```
Input: pos, IL
// Compute min S for interval maxima of length IL
minMax [pos] := min(minMax [pos] , ShellMax4 (pos, IL));
for r := IL/2 + 1 to IL − 1 do
    minMax [pos] := min (minMax [pos] , ShellMax8 (pos, r));
```

`ShellMax`$\{4, 8\}$ computes the maximum pixel value on a shell from row- and column interval maxima, as shown in Fig. 1. In this case, $r = 2$ and $\mathsf{IL} = 4$. Since a radius-r shell consists of $8 \cdot r$ pixels and interval lengths are powers of two, it is easy to see that this scheme applies to all shells of radius $r = 2^n (n \in \mathbb{N}_0)$. Each of the remaining $R - \log_2 R$ shells requires eight interval maxima – their four sides are pieced together from the maxima of two overlapping intervals.

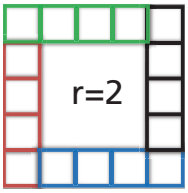


Figure 1: Assembling a shell from four 1-D intervals.

2.1 Analysis

Our new scheme requires $2 \cdot n$ values of auxiliary storage for the row- and column interval maxima. Since the inputs are copied there and not used afterwards, their storage can be reused for accumulating the minMaxima outputs. The pipelined nature of the algorithm enables a further reduction to $4 \cdot R + 2$ rows by organizing them as a sliding

Algorithm 4: CombineIntervalMaxima

Input: y, IL
for $x := 1$ **to** width **do**
  RM $[y, x] := \max(\mathsf{RM}[y, x], \mathsf{RM}[y, x + \mathsf{IL}])$;
  CM $[y, x] := \max(\mathsf{CM}[y, x], \mathsf{CM}[y + \mathsf{IL}, x])$;
`// Postcondition: IL now doubled`

window, but this would come at the price of more complex addressing.

We now examine the running time of the algorithm, which is somewhat obscured due to the four nested loops: height $\times \lceil \log_2 R \rceil \times$ width $\times$ numIM(IL). Note that loop interchange is possible because the innermost loop does not depend on width, so we combine that and height into a factor n. The number of interval maxima accesses is defined by `ShellMinMaxima`: numIM(IL) $= 4 + 8(\mathsf{IL}/2 - 1) = 4\mathsf{IL} - 4$, so:

$$\text{timePerPixel} = \sum_{L=1}^{\lceil \log_2 R \rceil} 4 \cdot 2^L - 4 = O(R)$$

The total complexity is therefore $O(n \cdot R)$, which is optimal because the filter must examine each shell and pixel.

2.2 Further Improvements

While the new algorithm is asymptotically optimal, there remains significant room for improvement. The RAM (Random-Access Machine) model underlying typical complexity measures has the virtue of simplicity but often mischaracterizes the real-world performance [7]. With cache misses now two orders of magnitude more expensive than basic operations [2], these effects can no longer be ignored. We will discuss some low-level issues in the context of the hotspot operator, but the existence of such techniques and the magnitude of the resulting improvements are likely to be of independent interest.

The past two decades have seen vast increases in CPU performance by means of higher clock speeds, better IPC (Instructions per Clock) and larger caches. [9] demonstrates that this trend cannot continue and that concurrency is already the key to performance. The first such development came in the mid 1990s with the introduction of vector SIMD (Single Instruction Multiple Data) instruction sets for several general-purpose microprocessor families [10]. These provide for storing multiple values in wide (e.g. 128-bit) registers and concurrently applying an operation to each of them. While rather limited, this form of programming can accelerate straight-line numerical applications without much cost. A second use for increasing transistor budgets has been to package multiple logical processors per socket, with quad-core multiprocessor systems widely available in 2008.

Fully utilizing the available hardware therefore calls for both vectorisation and parallelisation. In this work, a combined speedup of 27 has been achieved! Local filters

[2] DDR3 memory modules' 60 ns latency equates to 160 CPU cycles at 2.66 GHz [8].

are generally suitable for data-parallel processing, but the hotspot operator is limited by memory bandwidth due to its numerous and non-sequential memory accesses. Fig. 2 shows the scalability of the new algorithm on three different SMP systems. Parallel efficiency is only 50 % on a 16-core Intel machine. The memory bottleneck hypothesis is confirmed by better scalability on an AMD machine with multiple memory controllers and correspondingly higher bandwidth. Note that such systems have NUMA (Non-Uniform Memory Access) characteristics, which requires care to ensure each thread's working set is in local memory [11].

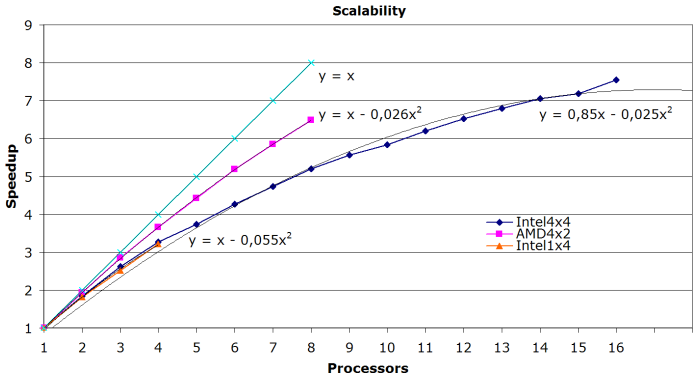


Figure 2: Scalability of the new algorithm on three SMP systems. Memory bandwidth is the limiting factor and is more plentiful on the AMD system.

The next step is vectorisation, which is possible because the per-pixel computations are independent and can be mapped to the SSE2 instruction set. We obtain an additional speedup of 3.6 via 8-way SIMD, which is helpful but surprisingly low. It turns out that the cause is a limitation in the Intel Core 2 microarchitecture regarding the handling of unaligned loads, an issue that will be discussed in depth in Appendix A. The takeaway is that the new algorithm will benefit from improvements in this area and the move towards multiple memory controllers, further improving its performance and scalability.

Another detail that has been considered is the overhead of so-called page walks. Each memory access requires virtual-to-physical address translation in the memory mapping unit, which involves examining multi-level page tables. A TLB (Translation Lookaside Buffer) serves to decrease this overhead by storing the result of the translation for a small number of recently-accessed memory pages. This specialized cache has strict latency requirements and can therefore only accomodate a few entries. If it is overloaded by random accesses in a large memory region, overhead increases dramatically since several accesses to memory are needed [12]. One means of avoiding this problem lies in the use of large memory pages (e.g. 4 MB instead of 4 KB on x86 architectures), thus increasing the coverage of the TLB. However, this turns out to be unnecessary in the case of the new algorithm as it is designed to operate in-cache and is therefore insensitive to memory latency.

One final microarchitectural issue that has affected the design of the algorithm is

also cache-related. The Intel i7 and AMD family 10h processors include a shared L3 cache, while Intel Core 2 CPUs consist of logical processor pairs sharing an L2 cache. In both cases, the caches are unpartitioned; unnecessary evictions can result from threads stealing each other's space. Having processors that share a cache work together on a task is about 7 % faster in some cases due to the reduction in contention. Even if partitioning strategies are improved, the cooperative scheme has the advantage of avoiding replication of common data and increasing the effective size of the cache. For working sets approaching a logical processor's share of the cache, the cache-aware method achieves a speedup of 1.45 due to its avoidance of thrashing.

3 Performance

The point of developing a new algorithm for the hotspot operator was to enable near-real-time processing of large datasets. Its success is determined by a performance comparison with the previous 'skip-shell' algorithm, which depends on the properties of the input data. To ensure relevant findings, we measure run times[3] for a set of four 'typical' high-resolution SAR images of different areas captured by three different sensors. The results are shown in Fig. 3 and indicate a very satisfying overall speedup factor of 7.3 (geometric mean). Note that the total input sizes (28.5 to 84.6 MPixels) do not appear to affect the processing rate. When run on a more recent workstation with dual W5580 CPUs, our implementation reaches 72 MPixel/s (chiefly due to the i7 family's higher memory bandwidth and better handling of unaligned memory accesses, c.f. Sec. A).

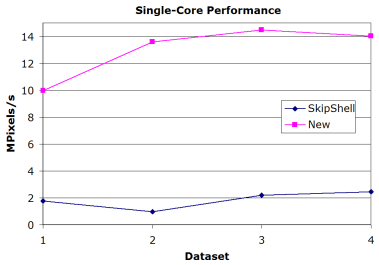


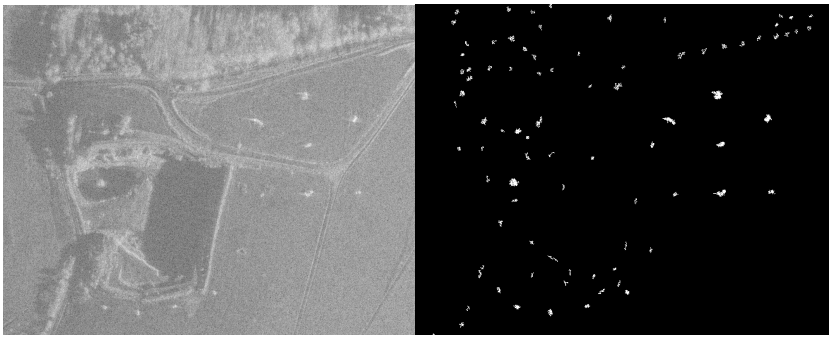
Figure 3: Comparison of single-core performance of the new vs. previous algorithm. The speedup factor ranges from 5 to 13.

4 Results

We show the results of the hotspot filter on a Dornier-SAR image of Kühlsheim (Fig. 4(a)), a scene containing both man-made objects and vegetation. We are particularly interested in vehicles and other compact objects. The hotspot filter (radius $R = 32$) sup-

[3]System specifications: Xeon X5365 CPU, 32 GB RAM, Windows XP x64, ICC 10.1.689.2008 settings: `/Ox /Og /Ob2 /Oi /Os /Oy /GF /EHsc /MD /GS- /GR- /Gd /Qopenmp /QxB /Qparallel /Qprof_use`

presses uniformly bright regions, because such pixel's shells are generally not darker than the center pixel. After the hotspot transformation, vehicle pixels and the remaining background pixels differ by three orders of magnitude (10^7 vs. 10^4). To improve the visualization, we compute connected components of nonzero pixels and discard objects smaller than an arbitrary cutoff of $12.7m^2$. The result is shown in Fig. 4(b). Subsequent steps in the image processing pipeline examine the candidate regions, e.g. classifying them via SVM.



(a) Logarithm of input (b) Hotspot-filtered

Figure 4: Airborne SAR image of Kühlsheim (65 cm resolution) and the result of the hotspot transformation.

5 Conclusion

Automatic screening for man-made objects in SAR or IR datasets entails detecting compact pixel clusters. The hotspot filter successfully suppresses other pixels, but is computationally expensive. We have introduced a new algorithm with linear complexity in the pixel count and object size, which is asymptotically optimal. Our sophisticated implementation avoids redundant computations by means of a divide and conquer scheme and organizes its memory accesses so the working set fits in cache. The overall 27-fold speedup is enabled by parallelization and vectorization. A single workstation is able to process 72 MPixel/s, which allows rapid screening of large datasets. The algorithm is used as a pre-processing step for multi-class target recognition in MSTAR SAR data via support vector machine.

A Unaligned Memory Accesses

It was mentioned that vectorisation of the hotspot operator yields a surprisingly low speedup and that the cause is related to Intel CPUs' poor handling of unaligned memory accesses. Since this issue seriously impacts performance and is likely to affect other applications as well, we will now delve into the details. An analysis of the Intel

Core 2 microarchitecture has found delays of 12 clocks in SIMD load operations that cross a cache line boundary [13, p. 83]. This issue is documented in [14, p. 5-38], which recommends using `LDDQU` to load two aligned vectors and shift the data into place, thus avoiding a cacheline split. An unfortunate design tradeoff in the Core 2 microarchitecture has replaced the implementation of this instruction with that of the architecturally equivalent `MOVQDU`, which remains affected by splits. The newer Intel i7 microarchitecture reduces the cost of splits to 2 clocks.

In the meantime, several workarounds have been attempted for the hotspot operator: substituting two 64-bit loads to decrease the probability of splits is consistently 4 % slower. Using `PALIGNR` to emulate `LDDQU` works but requires the misalignment to be known at compile-time. Realizing that access patterns for each interval length are fixed, several `ShellMax` functions were generated via templates and called through function pointers. This turns out to be 20 % slower, probably due to mispredicted indirect branches. A final alternative lies in manually aligning accesses, which is feasible because shell maxima computations only require three distinct misalignments. Unfortunately the SSE instruction set does not allow variable shifts of full registers and restricting all operations to the lower halves of registers costs about 25 % performance. Regardless, the overhead of two aligned loads, two shift and one OR-operation vastly outweighs the cost of cacheline splits. It appears that straightforward use of `MOVDQU` is currently the best option, especially because AMD microarchitectures also handle unaligned loads with only slight penalties.

We now show the performance impact of cacheline and page splits on Core 2 CPUs in the context of the hotspot operator. Assuming 2-byte values and 64-byte L1D cache lines, 7 out of the 32 possible misalignments should cross a cache line boundary. Instrumentation shows that the actual number is 22.13 %; this slightly higher number is due to the not quite uniform distribution of the misalignments. Similar arguments apply for page splits; assuming sizes of 4 KiB, we expect a ratio of 7 out of 2048 and observe 0.34 %, which is in good agreement. Using the per-split costs of 12 and 224 cycles given in [15] and supposing a 3 GHz processor, we therefore expect 1.42 seconds of CPU time to be lost due to the splits. A variant of the hotspot algorithm that rounds down all addresses to their natural alignment runs 1.33 seconds faster than the normal single-core version. This measurement matches the above prediction save for a slight difference due to the overhead of masking the lower address bits. Cacheline- and page split penalties have therefore been shown to be responsible for increasing total computation from 2223 ms to 3641 ms, i.e. a factor of 1.63!

To gain a better understanding of the cause, we have used the VTune profiler to observe certain CPU performance counters. The first surprising observation is a large amount of L1D misses despite the fact that these accesses are local. This and a cacheline split penalty equal to the L2 access latency leads to the presumption that such loads are simply not serviced by the L1 cache and must go through L2. Page splits apparently have a different effect because they do not cause an excessive amount of L2 misses. Instead we note a significant number of DTLB misses even though large pages are used and working set does not exceeed TLB capacity. This seems to point towards page splits requiring a page walk, especially because the overhead is similar to that reported in [12, p. 21]. These findings are in accord with [15].

While the above discussion may be deemed highly system-specific, it is also quite

relevant for real-world performance. It is safe to say that processors will generally — and perhaps to a surprising degree — penalize unaligned memory accesses. Since access patterns are intimately tied to the design of algorithms, this issue must be kept in mind during their design.

References

[1] R. Marques, F. de Medeiros, and D. Ushizima. Target detection in SAR images based on a level set approach. *IEEE Trans. Systems, Man and Cybernetics*, 39(2):214–222, March 2009.

[2] A. Kohnle, R. Neuwirth, W. Schuberth, K. Stein, D. Hoehn, R. Gabler, L. Hofmann, and W. Euing. Evaluation of essential design criteria for IRST systems. *Infrared Technology XIX*, 2020:76–92, 1993.

[3] N. Alon and B. Schieber. Optimal preprocessing for answering on-line product queries. Technical Report TR 71/87, Tel Aviv University, 1987. Preprint.

[4] M. Bender and M. Farach-Colton. The LCA problem revisited. In *Proc. of the 4th Latin American Symp. on Theoretical Informatics*, volume 1776 of *LNCS*, pages 88–94. Springer, 2000.

[5] I. Katriel, P. Sanders, and J. Träff. A practical minimum spanning tree algorithm using the cycle property. In *European Symposion on Algorithms*, volume 2832 of *LNCS*, pages 679–690. Springer, 2003.

[6] J. Fischer and V. Heun. A new succinct representation of RMQ-information and improvements in the enhanced suffix array. In *Combinatorics, Algorithms, Probabilistic and Experimental Methodologies*, volume 4614 of *LNCS*, pages 459–470. Springer, 2007.

[7] C. McGeoch. Experimental analysis of algorithms. *NOTICES: Notices of the American Mathematical Society*, 48(3):304–311, 2001.

[8] W. Fink. DDR3 vs. DDR2. http://www.anandtech.com/memory/showdoc.aspx?i=2989, May 2007.

[9] H. Sutter. The free lunch is over: A fundamental turn toward concurrency. *Dr. Dobb's Journal*, March 2005.

[10] R. Fisher. *General-purpose SIMD within a Register: Parallel Processing on Consumer Microprocessors*. PhD thesis, Purdue University, January 01 2003.

[11] D. an Mey and C. Terboven. Affinity matters! OpenMP on multicore and ccNUMA architectures. In *Parallel Computing: Architectures, Algorithms and Applications*, volume 15. Forschungszentrum Jülich and RWTH Aachen University, Febuary 2008.

[12] U. Drepper. What every programmer should know about memory. http://people.redhat.com/drepper/cpumemory.pdf, November 2007.

[13] A. Fog. *The Microarchitecture of Intel and AMD CPUs*. Copenhagen University, January 2008.

[14] Intel Corporation. *Intel 64 and IA-32 Architectures Optimization Reference Manual*, November 2007.

[15] Cache/page lines and LDDQU. http://softwarecommunity.intel.com/isn/Community/en-US/forums/thread/30244059.aspx, March 2008.