UNIVERSITAT POLITÈCNICA DE CATALUNYA
Department of Computer Architecture
Master in Computer Architecture, Networks and Systems

Exploiting contemporary architectures for fast nearest neighbor classification

Author: Isaac Jurado Peinado
Advisor: José Ramón Herrero

Barcelona, January 2010

Contents

1 Introduction **1**
- 1.1 Base algorithm . . . 1
- 1.2 Related work . . . 3

2 Optimizing the nearest neighbor search **5**
- 2.1 Execution environment . . . 5
  - 2.1.1 Hardware platform . . . 5
  - 2.1.2 Analytical model . . . 6
  - 2.1.3 Selected databases . . . 6
- 2.2 Memory optimizations . . . 7
  - 2.2.1 Stabilizing memory access . . . 7
  - 2.2.2 Reducing memory footprint . . . 12
- 2.3 Parallelization . . . 14
  - 2.3.1 Instruction level parallelism . . . 14
  - 2.3.2 Exploiting SIMD extensions . . . 17
  - 2.3.3 Going multicore . . . 19
- 2.4 Optimization summary . . . 21

3 Porting to GPGPU **23**
- 3.1 OpenCL programming overview . . . 23
- 3.2 Hardware platform . . . 25
- 3.3 Nearest neighbor in OpenCL . . . 25
  - 3.3.1 Multiple GPUs . . . 27
  - 3.3.2 Limitations . . . 28
- 3.4 Performance results . . . 29

4 Other classification methods **31**
- 4.1 k nearest neighbors . . . 31
- 4.2 Support Vector Machines . . . 31
- 4.3 Classification accuracy . . . 32
- 4.4 Performance comparison . . . 33

5 Conclusions **35**
- 5.1 Future work . . . 36

A Database properties **37**

B NN Executions results **39**

C Project resources **47**

Bibliography **47**

List of Figures

2.1 Memory access pattern for algorithm 2 8
2.2 Memory access pattern for algorithm 3 10
2.3 Blocking algorithm gains, from narrowest to widest 11
2.4 Memory usage growth for increasing data type sizes 12
2.5 Performance of algorithm 2 for all data types 13
2.6 Results of loop unrolling with blocking 16
2.7 Example of vectorized computation 18
2.8 Vectorization improvements, with unrolling and blocking . . . 19
2.9 Scalability of accumulated optimizations 20
2.10 Scalability of accumulated optimizations except blocking . . . 21
2.11 Average NCs per optimization and data type 22

3.1 OpenCL logical structure 24
3.2 Relation between the database matrices and the distances matrix 26
3.3 Work group layouts 27
3.4 Data separation to compute with two GPU devices 28
3.5 Performance comparison between CPU and GPU 29

4.1 SVM implementations comparison 33
4.2 Comparison between different classification methods 34

List of Tables

2.1 Characteristics of the selected databases 7
2.2 Synthetic database results 11
2.3 Performance of algorithm 3 relative to float 14
2.4 Differences in optimization results for database extreme widths 19
2.5 Average speedups for different data types 22

3.1 Average speedups on the GPU 29

4.1 Accuracy results 32

Chapter 1

Introduction

Modern computers provide excellent opportunities for performing fast computations. They are equipped with powerful microprocessors and large memories. However, programs are not necessarily able to exploit those computer resources effectively. This document presents various ways to implement and optimize nearest neighbor classification, as well as show how performance can be improved by exploiting the ability of superscalar processors to issue multiple instructions per cycle, by using the memory hierarchy adequately and by taking advantage of additional processor extensions.

The motivation of this work is exploiting contemporary hardware in a similar fashion as in [1]. Not only current CPU architectures, but also testing emerging new architectures such as GPGPU. After developing and benchmarking different nearest neighbor implementations, a selection of them will be compared against other classification approaches.

Finally, a brief summary of conclusions and future work will be discussed.

1.1 Base algorithm

The initial algorithm for finding the nearest neighbor is the most naive implementation. The expected input of the algorithm is a database consisting of two matrices: TRAIN and TEST that represent the training and testing sets respectively[1]. Each row represents a complete element of the set, while each column represents a single feature of an element. Therefore, both matrices shall have the same number of columns. When referring to a single feature/column of a particular element/row, the notation used will be a_i for a feature i of element a.

Apart from the matrices, which contain the actual data, the databases also contain some related metadata: the class labels. For convenience, the notation has been simplified to the following expression:

[1] Along this document, the term matrix refers to a single set of elements, whereas the term database includes the training set, the testing set and the related metadata

$$\text{classof}(a)$$

Where a is a complete row of either matrix TRAIN or TEST. The expression is mutable and will also be used as a left value to set the class label of an element.

In brief, the assumed parameters of all nearest neighbor algorithm listings are:

- D: number of dimensions/features of data element.
- N: number of test elements.
- M: number of training elements.
- TEST: test element set, a matrix of D columns by N rows.
- TRAIN: training element set, a matrix of D columns by M rows.

The task is to compare each element from the test set, i.e. each row of the TEST matrix, to all elements of the training set. Each comparison consists on calculating the euclidean distance of each pair. After comparing a test element against every training element, the training element with the minimum euclidean distance is being tracked; and its class label too. Algorithm 1 illustrates the idea.

Algorithm 1 Nearest neighbor approach in essence

```
for all a ∈ TEST do
  min ← ∞
  for all b ∈ TRAIN do
    if distance(a, b) < min then
      min ← distance(a, b)
      cls ← classof(b)
    end if
  end for
  classof(a) ← cls
end for
```

Although the euclidean distance would be defined as:

$$\text{distance}(a,b) = \sqrt{\sum_{i=1}^{D}(a_i - b_i)^2}$$

In [1] it is simplified to:

$$\text{distance}(a,b)^2 = \sum_{i=1}^{D}(a_i - b_i)^2$$

That is, the square of the euclidean distance. This is done to avoid the cost of computing the square root. Applying the change to algorithm 1 leads to algorithm 2.

Algorithm 2 Starting point nearest neighbor search

```
for all a ∈ TEST do
  min ← ∞
  for all b ∈ TRAIN do
    dist ← 0
    for i = 0 to D do
      dist ← dist + (a_i − b_i)^2
    end for
    if dist < min then
      min ← dist
      cls ← classof(b)
    end if
  end for
  classof(a) ← cls
end for
```

1.2 Related work

The Nearest Neighbor (NN) classification procedure is a popular technique in pattern recognition, speech recognition, multitarget tracking, medical diagnosis tools, etc. The NN algorithm has some strong consistency results. As the amount of data approaches infinity, the algorithm is guaranteed to yield an error rate no worse than twice the Bayes error rate, i.e. the minimum achievable error rate given the distribution of the data [2].

A major concern in its implementation is the immense computational load required in practical problem environments. Other important issues are the amount of storage required and the data access time.

In this document, we address these issues by using techniques widely used in linear algebra codes. We show that a simple code can be very efficient on commodity processors and can sometimes outperform complex codes which can be more difficult to implement efficiently. Comparison of the nearest neighbor with other methods on different application areas can be found elsewhere [3, 4, 5]. To find disquisitions about appropriate distance measures the reader is referred to [5, 6, 7, 8].

There is also some previous experience trying to port the NN generalization algorithm, the k-NN, to the GPU [9]. Unfortunately, it led to confusing and unclear results. Other GPU ports include a Support Vector Machine implementation [10].

Apart from being a classification tool, the nearest neighbor search can be understood as a problem by itself, with its different approaches to solve it. Aside from the naive, or brute force, algorithm, there are other possibilities to find the nearest neighbor in a set of data. For example, by using space partitioning structures such as the kd-tree [11]. By performing Locality Sensitive Hashing, a high dimensional space may be reduced after mapping similar buckets [12].

Chapter 2

Optimizing the nearest neighbor search

Starting from the base algorithm shown in section 1.1, this chapter iteratively enhances it by exploiting different hardware aspects: memory usage, through locality, and the different levels of parallelism available.

2.1 Execution environment

All the code derived from the algorithms presented in this chapter has been developed on a dual processor Intel Xeon E5520 "Gainestown", running at 2262 MHz; the first generation of the *Nehalem* Intel microarchitecture. This section describes the characteristics of such architecture and provides additional details related to the way programs are executed.

2.1.1 Hardware platform

The Nehalem microarchitecture [13] is the first Intel processor design that features an on-chip DDR3 memory controller, one per processor or chip. This makes Nehalem a hybrid between SMP[1] and NUMA[2]: each processor contains several cores, four in the case of the E5220, which share main memory in a SMP fashion, but memory is distributed in different memory modules among the number of processors in the system.

To solve "remote" memory accesses, i.e. requesting data from a module connected to a different processor, the QPI[3] bus interconnects all processors directly, point to point, just like a crossbar.

Another novelty is the presence of a third cache memory level. Traditionally, and since cache memory started to be bundled into the processor

[1] Symmetric Multi-Processing
[2] Non Uniform Memory Access
[3] Quick Path Interconnect

chip, Intel designs had two cache levels: L1 and L2. Nehalem processors include a single shared L3 cache and independent L2 and L1 caches on each core. In the E5220 the L2 caches are 256 kilobytes each and the L3 cache is 8 megabytes per processor.

Each core is capable of executing six simultaneous operations: three memory operations and three computational operations. It also features SSE[4] 4.2 where the latency of these vector operations has been reduced to a single cycle; prior SSE implementations had a two cycle latency. There is also the possibility of running two simultaneous threads on a single core, with *Hyper-Threading*, but the functionality was disabled for this work.

Instruction loops are detected with the aid of the *Loop Stream Detector*; a piece of circuitry that avoids fetching and decoding the same instructions for each iteration. Moreover, the core contains hardware prefetch logic that does anticipatory and asynchronous memory transfers dynamically at run time, without requiring changes in the program code.

Finally, the testing machine was equipped with eight gigabytes of main memory, four gigabytes for each processor.

2.1.2 Analytical model

Execution time would be the obvious performance metric to be used, but due to the variety in training and test database sizes there is a need for a normalized value. In this chapter, the same metric as in [1] will be used: the *Normalized Cycle*. NC is computed by:

$$NC = \frac{\text{CPU\_time\_in\_cycles}}{N \cdot M \cdot D}$$

Just as in [1], the NC is modeled with the following expression:

$$NC = NC_{cpu} + NC_{mem}$$

where NC_{cpu} is the component obtained when no misses occur in the memory hierarchy (caches, TLBs, page faults) and the NC_{mem} represents the penalty cycles due to the misses in the memory system. Misses produced by instruction fetches are not considered since a separate instruction cache exists and the programs evaluated are sufficiently small so that no instruction misses occur.

2.1.3 Selected databases

Benchmarking was carried out using 27 different training and test database pairs. However, to simplify the figures and tables, only some of them are used to show the results in this chapter.

[4]Streaming SIMD Extension, where SIMD stands for Single Instruction Multiple Data

Databases have three important attributes: the number of features or dimensions (D), the number of training elements (M) and the number of test elements (N). If the values of D, M and N can be small or large, there are eight possible types of database. This document mainly focuses on the magnitude of D, with the following terminology: when D is small, the database is said to be *narrow*, on the opposite case the database is *wide*. Table 2.1 provides the details of the selected databases used to measure algorithm performances.

Label	D	M	N	**Floating point**
`synth_s` (Synthetic small)	122	268	2148	no
`synth_l` (Synthetic large)	122	18760	150360	no
`aa9a` (Narrow)	122	32560	16281	no
`w8a` (Wide)	300	49749	14951	no
`ijcnn1` (Narrowest)	22	49990	91701	yes
`mnist1` (Widest)	780	21000	49000	yes

Table 2.1: Characteristics of the selected databases

There are two synthetic databases in table 2.1, `synth_s` and `synth_l` to illustrate the difference between small problems that fit in cache and large problems which do not fit in cache. This is developed further in section 2.2.1.

Details of all the databases used during development can be found in appendix A.

2.2 Memory optimizations

The memory access pattern in algorithm 2 is very predictable and with a high spatial locality, as the memory is traversed in a contiguous manner. Of course, matrices are stored in row major order which is the natural disposition for algorithm 2: all elements in the same row are next to each other, and the last element of a row precedes the first element of the next row.

Figure 2.1 illustrates a complete iteration of the outermost loop in algorithm 2. A complete row of the TEST matrix is compared against the whole TRAIN matrix. This generates a lot of memory requests, which is specially harmful for large matrices that will not fit in any of the cache memories; consequently generating more cache misses. Such misses will have an impact on NC_{mem} and, consequently, on NC.

2.2.1 Stabilizing memory access

In order to approximate NC_{mem} to zero, cache misses must be reduced. Algorithm 2 clearly stresses the memory bus when matrices are large enough.

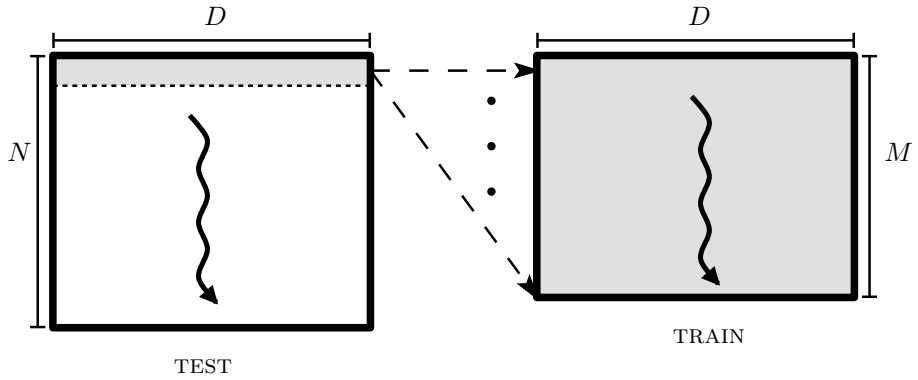


Figure 2.1: Memory access pattern for algorithm 2

Reducing main memory access helps obtaining uniform results for any database size.

The spatial locality of algorithm 2 is already good as the memory accesses are mostly contiguous: matrices are stored in row major order[5] and traversed the same way, following an uniform ascending memory address pattern. For large databases, the matrices are unlikely to fit completely in any cache level; thus resulting in an almost inexistent temporal locality between iterations in the outermost loop.

Reusing training elements has proved to be an efficient way to reduce processor stalls due to cache misses [1] by taking advantage of the outer level of the cache memory in typical two level configurations. Adding a second level of blocking is not motivated by the presence of a third cache level but by two other reasons: the difference in access latency between L2 and L3, L3 is about five times slower than L2, and the fact that L3 is shared whereas L2 is private. Naturally, the block to be hold in L2 should come from the matrix that has more potential temporal locality: TRAIN.

Algorithm 3 shows a possible solution. It includes two new notation elements:

1. The mindist(a) extended metadata that works analogously to classof(a).
2. The partition TEST_BLOCKS (and respectively TRAIN_BLOCKS), that obeys the identity:

$$\text{TEST} = \bigcup_{T \in \text{TEST\_BLOCKS}} T$$

Now the memory access pattern changes slightly. Matrices TRAIN and TEST are virtually partitioned in blocks with sizes smaller than the total

[5] All elements in the same row are next to each other, and the last element of a row precedes the first element of the next row

Algorithm 3 Two level block reuse

```
for all tsb ∈ TEST_BLOCKS do
  for all trb ∈ TRAIN_BLOCKS do
    for all a ∈ tsb do
      min ← mindist(a)
      for all b ∈ trb do
        dist ← 0
        for i ← 0 to D do
          dist ← dist + (a_i − b_i)^2
        end for
        if dist < min then
          min ← dist
          cls ← classof(b)
        end if
      end for
      mindist(a) ← min
      classof(a) ← cls
    end for
  end for
end for
```

capacities of the L2 and L3 caches respectively. This is illustrated in figure 2.2. The two outer loops control the bounds and the number of executions of the loops traversing TEST and TRAIN respectively. Because the inner loops only work in a single block, not all distance comparisons are performed. The test database metadata needs to be extended with the minimum distance associated to the currently assigned class label, as a closer element may exist in a different training database block.

Block algorithm results

When aiming for the best possible results, the size of the blocks must be adjusted properly. Approximating the size of the blocks to any of the cache level sizes would probably cause almost the same amount of conflicts as when blocks are not not used at all; and indeed it happens because there are other data to hold in cache apart from the databases content. The block size should be large enough to provide improvement and small enough to leave some cache memory for other data.

Experimenting led to block sizes of 2 megabytes for L3 blocks and 128 kilobytes for L2 blocks. L3 block size is proportionally smaller than L2 intentionally because, first, L3 cache is shared among all four cores of the chip while a private L2 is present on each one, and second, L3 is a superset of all four L2 caches. By not using all L3 available memory, the chances of

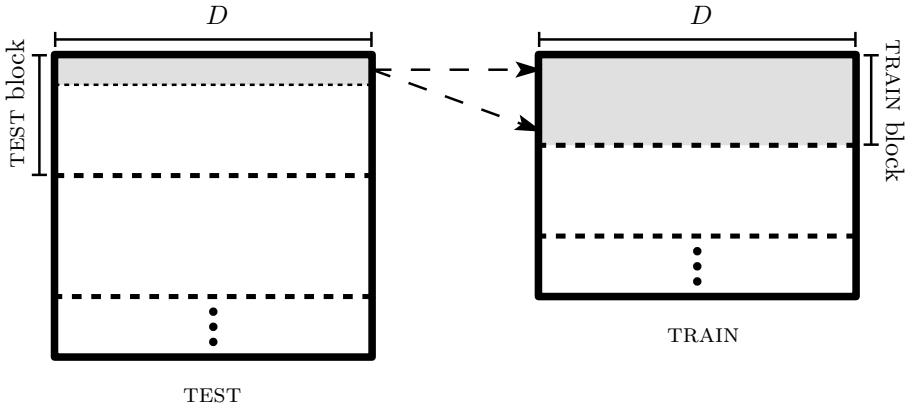


Figure 2.2: Memory access pattern for algorithm 3

cache conflicts with other thread running in other cores of the same processor are very low.

The configuration for one level of blocking consists of partitioning only the TRAIN matrix into L3 blocks. Adding the second blocking level means partitioning TEST into L3 blocks, and TRAIN into L2 blocks.

To see how algorithm 3 provides a stable rate of cache misses for any database size, the artificially crafted databases, listed in table 2.1, will serve for the purpose. The matrix sizes for the synthetic small database are 127.72 kilobytes for TRAIN and almost 1 megabyte for TEST; this way TRAIN fits completely in L2 and TEST fits in L3. The large database is exactly seventy times larger than the small one, resulting in 8.73 megabytes and 69.98 megabytes respectively.

As database `synth_s` should generate very few cache misses, then

$$NC_{mem}(\text{Small}) \approx 0$$

which means that

$$NC(\text{Small}) \approx NC_{cpu}(\text{Small})$$

Database `synth_l` is proportionally larger than `synth_s`, so NC_{cpu} values should be equal. The NC_{mem} component in the large database can be estimated by

$$NC_{mem}(\text{Large}) \approx NC(\text{Large}) - NC(\text{Small})$$

Table 2.2 contains the NC value of the executions with different levels of blocking, using single precision floating point arithmetic.

Performance for the small database sees practically no change, however, the large database experiences a total gain of a 3.94% with the two level blocking approach. Furthermore, the performance matches the small

Algorithm	NC(Small)	NC(Large)
No blocking	3.03	3.14
One level blocking	3.03	3.04
Two level blocking	3.03	3.03

Table 2.2: Synthetic database results

database results. Replacing the results presented in table 2.2, for the two level blocking algorithm, the NC_{mem} estimation leads to $NC_{mem}(\text{Large}) \approx 0$. In other words, algorithm 3 will perform equally for all database sizes.

In general, blocking truly matters when the TRAIN matrix does not fit in cache, as it is completely traversed several times. The second blocking level does not provide much improvement, but still, it brings some control over L2 and L3 altogether.

For the sake of completeness, figure 2.3 compares the performance at different blocking levels and type sizes. The gains of the blocking strategies are not as impressive as they could be. Even in the most favorable case the gain is almost 10% altogether, of which only 1% is provided by the second blocking level.

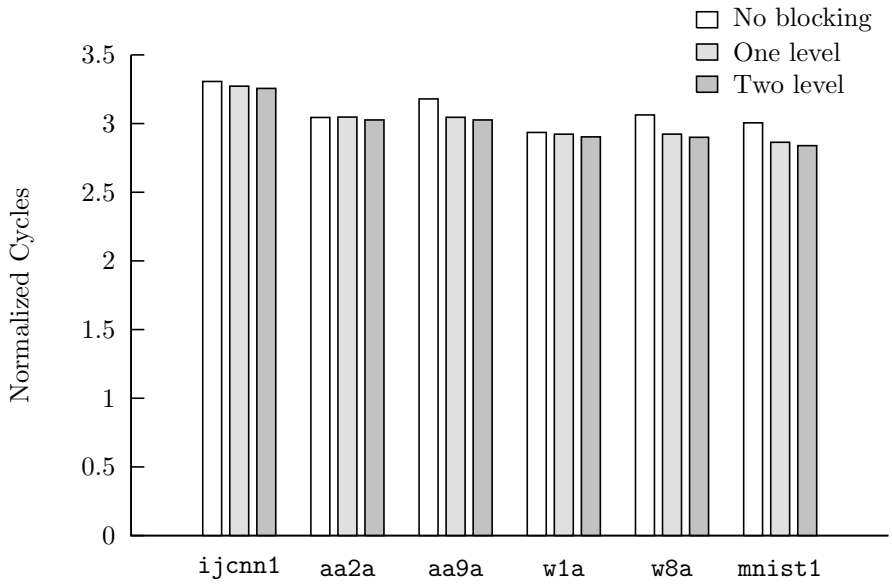


Figure 2.3: Blocking algorithm gains, from narrowest to widest

It should be noted, also from figure 2.3 that wide databases perform better than narrow ones. This is not a coincidence as it is a consistent behavior along all the results. Recalling the end of section 2.1.1, the Loop Stream Detector is a component of the core which is able to save the task of fetching and decoding the loop's body for later reuse. The innermost loop

in algorithms 2 and 3 "streams" over whole matrix rows, without branching code and accessing memory sequentially; not to forget its compact size. Seems like an ideal case for the Loop Stream Detector. Therefore, the more innermost iterations, i.e. higher value of D, the more amortized the LSD is.

Nevertheless, the LSD does not explain the little gains by applying blocking to very large databases. In this case, all bets are on hardware prefetch together with a clever cache replacement policy. The Nehalem microarchitecture also introduced changes in cache memory policies. It could be that the cache is very smart and detects streamed memory access or that LSD is triggering it. Stream data does not required to be cached, as it is usually accessed only once and in a predictable manner. Reaching such a level of sophistication in a cache controller would not be a surprise by current standards.

In the following sections, the algorithms used to describe optimizations will omit blocking, however, the performance measurements will have blocking applied.

2.2.2 Reducing memory footprint

Choosing a data type for the matrices (or data sets) affects performance in two ways: the speed of the corresponding arithmetic unit, integer vs floating point and the size of the type and the total amount of memory transferred.

Figure 2.4 shows an example of how memory usage grows with the size of the data type used. Considering that the sizes of a short, an int, a float and a double are two, four, four and 8 bytes respectively, the figure simply reflects this growth in size.

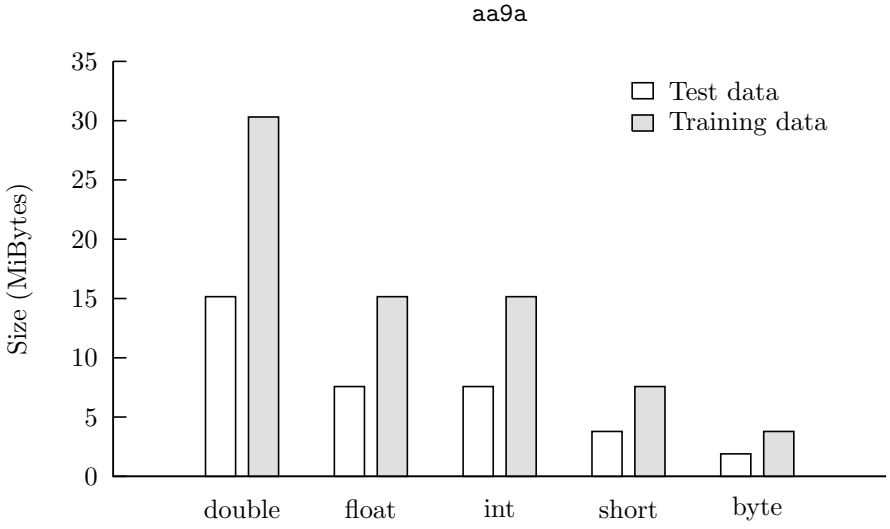


Figure 2.4: Memory usage growth for increasing data type sizes

In previous high-end architecture generations, floating point unit performance compensated the memory weight of its data types, both for single and double precision [1]. Nowadays, integer performance has improved significantly; enough to compare data types again.

On the downside, though, not all data types may be available for a concrete database. It could either contain real numbers, in which case all integer options have to be dropped, or have properties that might overflow or saturate smaller integer types while calculating distances between elements; thus invalidating the algorithm.

Type size results

Now that memory size is not an issue thanks to the blocking technique presented in section 2.2.1, figure 2.5 illustrates how the different types perform. The `ijcnn1` and `mnist1` databases contain real numbers so they cannot be loaded into integer data types.

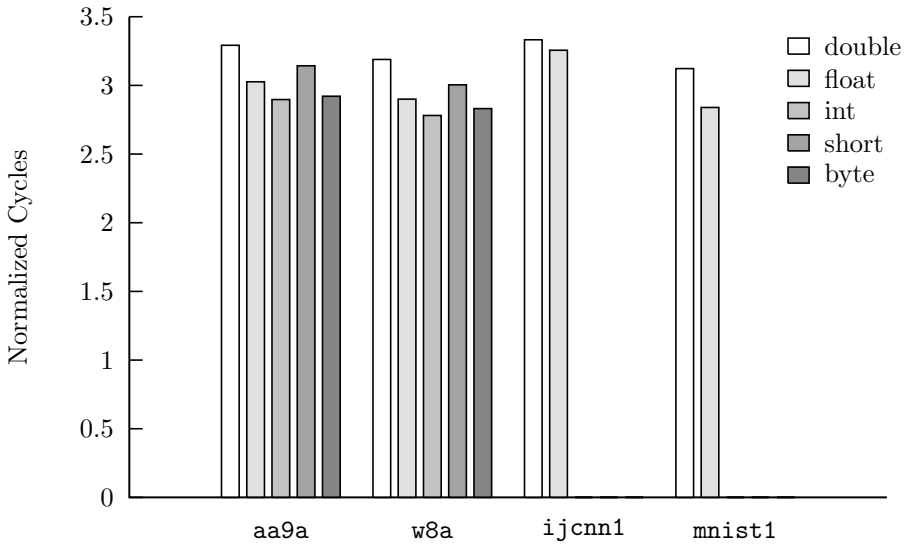


Figure 2.5: Performance of algorithm 2 for all data types

The speedups and slowdowns relative to float are shown in table 2.3. Speedups will be expressed in percentage (%) or in orders of magnitude (x); positive value means improvement whereas a negative value indicates worsening.

The first observation is that word-sized integers perform better than single precision floating points, around 4.5% better. This result contrasts with [1]. However, this behavior can be attributed to differences in operation latencies between the integer and floating point arithmetic units; Intel's processors integer arithmetic units have had one cycle latency for some generations. Floating point calculations, although having one cycle

Database	double	int	short	byte
aa9a	−8.07%	4.48%	−3.71%	3.62%
w8a	−9.06%	4.29%	−3.47%	2.43%
ijcnn1	−2.28%	–	–	–
mnist1	−9.06%	–	–	–

Table 2.3: Performance of algorithm 3 relative to float

throughput, need more cycles to complete an operation, and in the presence of data dependencies, the optimal throughput cannot be achieved.

Even more intriguing is the pathology seen in shorts and bytes. While bytes execute faster than floats ($\approx 3\%$), they are still behind ints. The case for shorts is even worse, with a 3.5% slowdown. The explanation is simple: byte load has more overhead than a word size load because it translates into more micro-operations, e.g. truncation of higher bits. In the long run, this little overhead becomes visible.

In conclusion, smaller data types only reduce the memory usage, but the execution pipeline performs the same conversion operations. Except for shorts, about which one may think that it is a data type in process of deprecation.

2.3 Parallelization

Parallelism implies performing multiple operations at the same time, and it comes with different granularities: from executing various operations in different functional units of the same pipeline, to run simultaneous tasks on multiple processors. Current hardware trends push software towards enabling and exploiting all kinds of parallelism, this section describes three complementary approaches to obtain more performance out of the processor.

This section presents parallelization at three different granularities, one for each loop of algorithm 2.

2.3.1 Instruction level parallelism

When the delays produced by memory accessed are reduced, other type of stalls may appear during the execution of the algorithm: pipeline data dependency hazards. The *dist* variable contains the sum accumulator for the distance between elements a and b. Its accumulator nature introduces a data dependency between successive iterations of the innermost loop.

A processor core of the Nehalem microarchitecture, as its contemporaries, is equipped with several integer and floating point arithmetic units; enabling the possibility of having various operation executing simultaneously. This is also known as instruction level parallelism: the ability to

issue multiple operations in a single clock cycle. Nevertheless, data dependencies impede taking advantage of instruction level parallelism by forcing the processor to be idle until computed data is available.

The solution in this case is to compare an element from TEST with multiple TRAIN elements in a single iteration; formally referred as loop unrolling. Algorithm 4 shows an example where two iterations have been unrolled. After unrolling, the resulting instructions are interleaved in a way that distances away operations with data dependencies. Treatment for residual iterations has been omitted for simplicity.

Algorithm 4 Middle loop two iteration unroll

```
for all a ∈ TEST do
  min ← ∞
  for all (b, c) ∈ TRAIN do
    dist ← 0
    dist′ ← 0
    for i ← 0 to D do
      dist ← dist + (a_i − b_i)^2
      dist′ ← dist′ + (a_i − c_i)^2
    end for
    if dist < min then
      min ← dist
      cls ← classof(b)
    end if
    if dist′ < min then
      min ← dist′
      cls ← classof(c)
    end if
  end for
  classof(a) ← cls
end for
```

The loop of algorithm 4 has a new notation in it. The meaning is simple: at each iteration, obtain two consecutive elements from TRAIN instead of one; then, in the next iteration, the following two elements should be taken. In the end, TRAIN should have been traversed completely with half the iterations as in algorithm 2. If the unrolling degree was four, then the middle loop should advance accordingly.

Loop unrolling should solve the poor floating point throughput problem found in the results of section 2.2.2.

Loop unrolling results

The unrolling technique explained in the previous section has been integrated with compiler technology to provide some automatic way to perform such optimization with a more degree of detail, as the compiler has better knowledge about its own instruction scheduling. In the *GNU C Compiler* (GCC), automatic loop unrolling may be enabled through the `-funroll-loops` command line switch. Results are shown in figure 2.6, comparing manual unrolling of two (algorithm 4) and four iterations against automatic unrolling and no unrolling at all.

Automatic unrolling brings particularly noticeable gains for integer arithmetic. The compiler does a very good job dealing with shorts and bytes, with gains of 10.28% and 12.77% respectively, better than manually unrolling four iterations. Certainly, manual optimization of code dealing with small integer types does not seem possible without resorting to descend directly into the ISA[6].

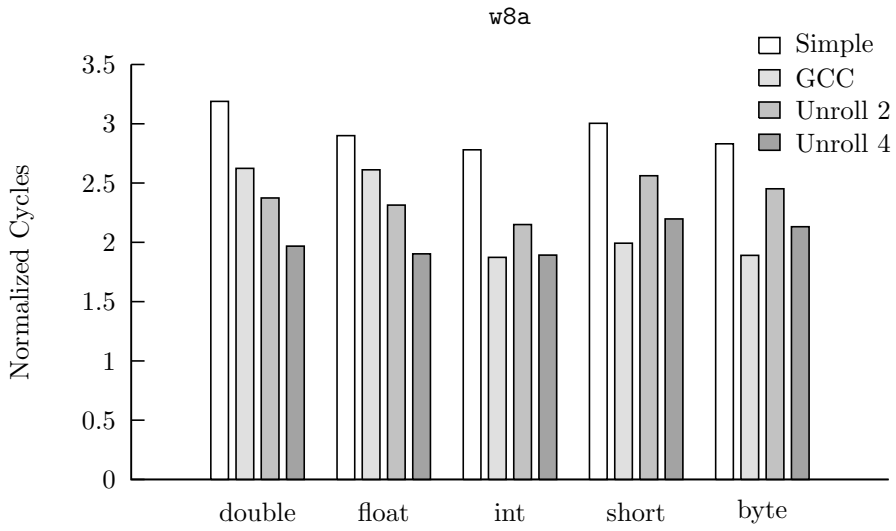


Figure 2.6: Results of loop unrolling with blocking

These results confirm the previous presence of pipeline stalls due to data dependencies. Moreover, the performance of the floating point arithmetic units is vastly improved by compensating their higher latency with better throughput; thus becoming almost as competent as integer arithmetic units, even when computing as double precision.

[6]Instruction Set Architecture

2.3.2 Exploiting SIMD extensions

SSE is the name of Intel SIMD extensions. From the instruction set architecture point of view, it consists of a bunch of new machine instructions that operate on a dedicated set of special registers. Such registers are 128 bit wide and, when loaded with data, can be treated as 2 doubles, 4 float or integers, 8 shorts or 16 bytes. Many actual processors include some extension circuitry specially designed to execute multiple operations by issuing a single instruction.

In general, vector instructions were introduced for multimedia and stream processing where a high degree of fine grained parallelism is easily achieved. For example, increasing the brightness of a picture, where each pixel can be treated independently from the rest.

SSE works best at gathering contiguous data in a stream fashion. The TRAIN matrix is traversed this way, so it seems reasonable that the distance calculation can be "vectorized" as defined in algorithm 5.

Algorithm 5 Inner loop vectorization with four element vectors

```
for all a ∈ TEST do
  min ← ∞
  for all b ∈ TRAIN do
    dist⃗ ← (0, 0, 0, 0)
    for i ← 0 to D/4 do
      dist⃗ ← dist⃗ + (a⃗_i − b⃗_i)^2
    end for
    dist ← dist_x + dist_y + dist_z + dist_t {Vector reduction}
    if dist < min then
      min ← dist
      cls ← classof(b)
    end if
  end for
  classof(a) ← cls
end for
```

With this optimization, the innermost loop needs fewer iterations to complete. Depending on the data type used, the number of parallel operations goes from two to sixteen. For example, in figure 2.7, each component of $\overrightarrow{a_i}$ is compared against its corresponding $\overrightarrow{b_i}$ component, all at the same time. As the accumulator is also a vector, the sum of its components will result in the square of the euclidean distance between a and b.

It is worth mentioning that algorithm 5 does not reflect some restrictions imposed by the use of SSE. For example, when loading a vector register, the starting memory address should be aligned to 16 bytes to avoid the performance penalty incurred otherwise. This forces to insert artificial features

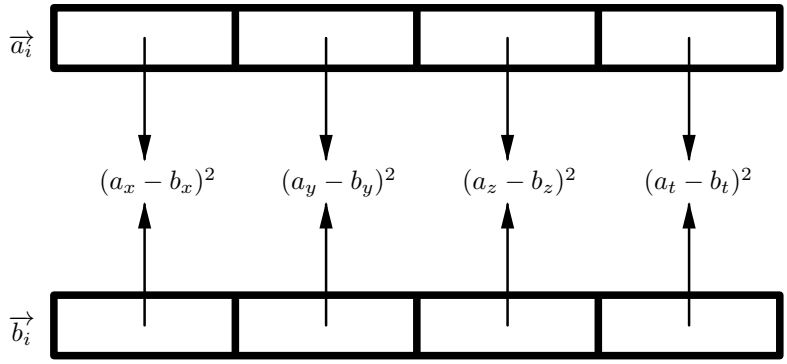


Figure 2.7: Example of vectorized computation

for padding purposes, incurring in memory increases around 5%. On the other hand, the SSE instruction set is not uniform for all the types, so the number of instructions needed to implement integer and float versions vary slightly.

Despite the limitations, vectorizing the innermost loop it is an important optimization step in various aspects. Firstly, the used data type may change the performance of the algorithm more severely, owing to the amount of possible simultaneous computations. Secondly, for some data types, special operations such as multiply and add may be available, which would reduce the amount of instructions. Finally, as data is processed in larger chunks, previously commented optimizations may become more significant.

Vectorization results

In theory, the expected speedups should be of an order of magnitude according to the amount of elements per register vector of a particular data type. In practice, only some results approach the expectations. As it can be seen from figure 2.8, the integer vectorization improvement is almost the half ($2.65x$) of what it should be ($\approx 4x$). Ironically, floating point data type vectorization generates better results: $1.95x$ for double precision and $3.80x$ for single precision, where ideal speedups would be $2x$ and $4x$ respectively. Note that scalar versions in 2.8 are the unrolled versions from figure 2.6.

The lack of concrete architectural information about the execution units of the pipeline, as well as the capabilities of all of them, hardens the task of finding a suitable explanation for this behavior. It could be that the sequencing of vector operations, even with unrolled loops, does not allow the hardware to run optimally. A more plausible reason, from the opposite point of view, could be that the hardware reorders scalar operations so well, specially with integers, that the IPC[7] count is greater than one; i.e.,

[7]Instructions Per Cycle.

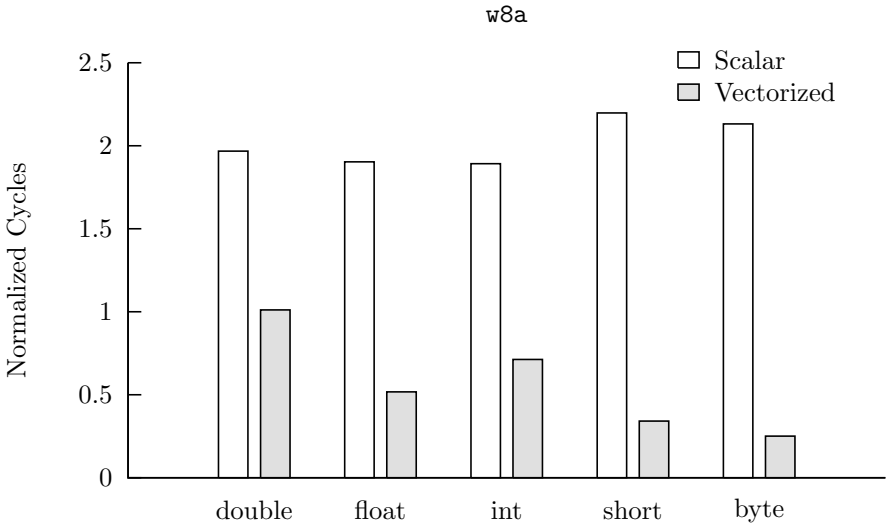


Figure 2.8: Vectorization improvements, with unrolling and blocking

Optimization	Scalar speedup	Vector speedup
Blocking (`ijcnn1`, narrow)	1.55%	3.46%
Unrolling (`ijcnn1`, narrow)	57.02%	46.77%
Blocking (`mnist1`, wide)	5.85%	32.06%
Unrolling (`mnist1`, wide)	50.59%	58.11%

Table 2.4: Differences in optimization results for database extreme widths

operations are computed in parallel in different execution units.

Table 2.4 shows the improvements obtained with the application of blocking and unrolling to algorithm 2 for both scalar and vectorized versions. Blocking and unrolling have been applied separately here so that we can measure their contribution individually. As it has been occurring along this chapter, the innermost loop is the place where all the algorithm's computational power is consumed, so, again, optimizations are more notorious when more time is spent in it. Moreover, vectorization reduces considerably the number of iterations of the innermost loop, increasing the percentage of loop overhead for narrow databases.

2.3.3 Going multicore

This is the simplest optimization yet one of the most effective. By using OpenMP, the algorithm does not have to be altered at all. It is only necessary to place the correct directive on the outermost loop of algorithm 2 (or the middle loop in algorithm 3) making all variables thread private except for the databases, which should be shared among threads in order to avoid stressing cache memories.

Finding the nearest neighbor of an element from the test database is a completely independent operation. Consequently, the amount of work is equally distributed among the cores, leading to an expected linear scalability. In other words, doubling the number of threads should double the overall performance of the algorithm.

OpenMP results

The scalability of most optimization combinations is, in general, stable and almost linear. Figure 2.9 shows the scalability graph of all optimizations applied for various database types and dimensions. Contrary to what could be seen in previous results, narrow databases scale particularly well independently from the applied optimizations. While the initial performance is worse than with wide databases, cache conflicts between different cores are not as hard due to a lower prefetching performance; thus easing scalability.

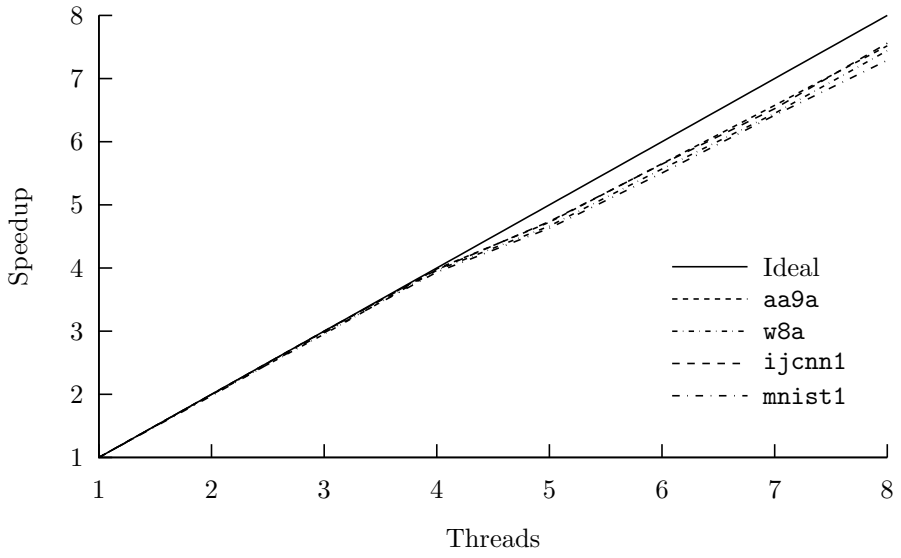


Figure 2.9: Scalability of accumulated optimizations

All tests show the particular shape seen in figure 2.9: a small drop in scalability from four to eight threads. This is a consequence of the Nehalenm design and its QPI bus. Nehalem machines with more than one processor socket do not follow a completely SMP[8] architecture; instead, they can be categorized as a mixture of SMP and NUMA[9]. Multiple cores in a single chip are, in fact, SMP and share their memory whereas each chip has a dedicated memory module.

Upon initialization, the nearest neighbor process loads data from disk to

[8]Symmetric Multi Processing
[9]Non-Uniform Memory Access

memory. At this stage, there is only one thread running and the operating system allocates all the necessary memory space in its corresponding memory module. When the four cores of the chip are used, additional threads are scheduled on the second processor; however, the required data is available only in the memory module of the initial processor so cache line requests and responses need to go over the QPI bus, causing a slight overhead.

Scalability only flattens for non-blocking versions, as shown in figure 2.10, where cache memory conflicts between different cores degrades the performance as the concurrency grows. Ironically, narrow databases are less affected because they generate less potential cache misses from the innermost loop. Once blocking is applied, scalability becomes the same as in figure 2.9.

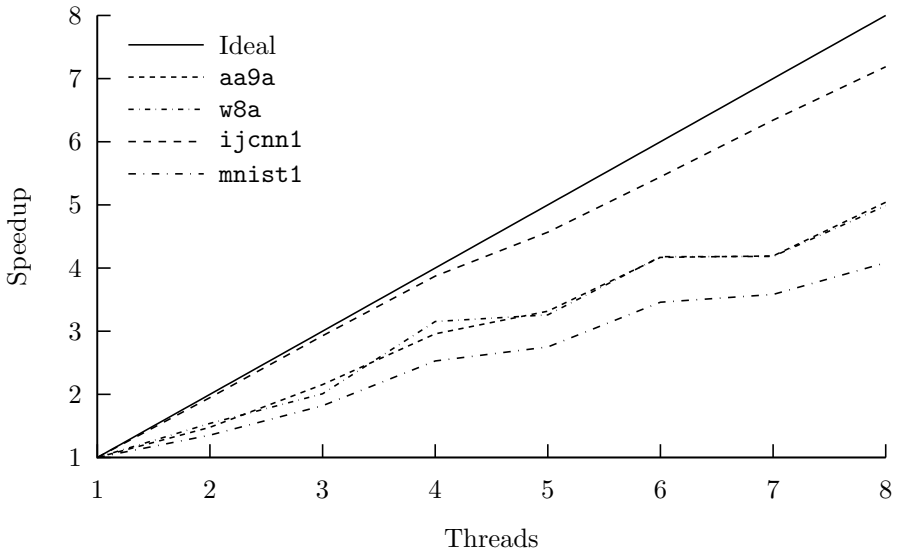


Figure 2.10: Scalability of accumulated optimizations except blocking

2.4 Optimization summary

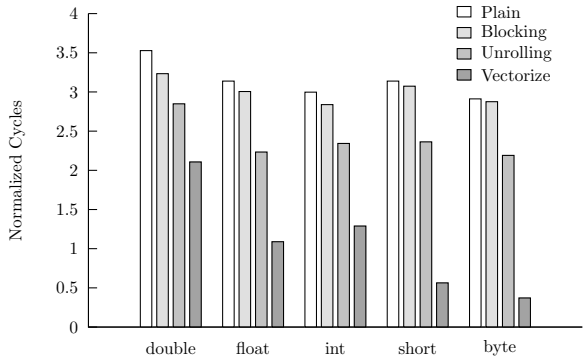
In summary, this section presents some global review of the previously presented optimizations. Firstly, table 2.5 shows the average speedup from algorithm 2 to a combination of algorithms 3, 4 and 5, executed with a single or multiple threads. The scalability is almost ideal given that the right column is, approximately, the result of multiplying the values on the left column by 7.5.

Figure 2.11 examines the effect of each optimization applied on its own over algorithm 2, figure 2.11(a), or accumulated on top of the previous optimization, figure 2.11(b). The NC value is the average of the four databases: `ijcnn1`, `aa9a`, `w8a` and `mnist1`. Notice how unrolling benefits from blocking, specially with large data types, and vectorization benefits from both;

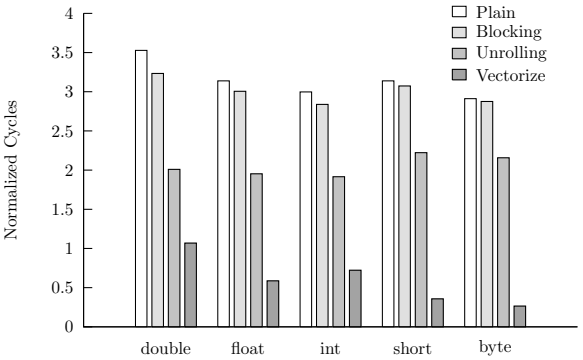
Data type	Single thread	Eight threads
byte	11.04x	82.74x
short	8.80x	66.42x
int	4.15x	31.52x
float	5.49x	40.85x
double	3.33x	24.35x

Table 2.5: Average speedups for different data types

although it is the strongest standalone optimization.



(a) Independently applied



(b) Accumulated

Figure 2.11: Average NCs per optimization and data type

Chapter 3

Porting to GPGPU

GPGPU is the acronym for General Purpose programming on the Graphics Processing Unit. In essence, it is about using taking advantage of the great computing power available in current GPU devices.

This practice is becoming increasingly popular among the scientific community thanks to the favorable price per GigaFLOP ratio. By assembling multiple devices together, a single computer can easily become a home supercomputer in terms of peak performance. Nonetheless, the GPGPU architecture carries some inherent complexities the programmer must overcome.

GPU vendors try to lessen these difficulties by providing new programming languages, toolkits and libraries. As the time of this writing, the most popular is CUDA, the NVIDIA GPGPU toolkit. Other brands also provided their own tools. Fortunately, a new standard framework called OpenCL[1] appeared to unify research effort and reduce market fragmentation.

3.1 OpenCL programming overview

OpenCL has been designed to develop on heterogeneous platforms, where different architectures coexist and cooperate. In an OpenCL application, there must be at least two entities involved: a host and one or more devices. OpenCL provides a programming language, based on C99, to develop code, in the form of functions, that will be executed on the devices; these functions are called *kernels*. On the host side, OpenCL offers a complete API to control the execution of the kernels in the devices. The device executes its kernels asynchronously from the host to allow overlapped execution, although the host has the option to block its execution until the device finishes.

A device is structured as shown in figure 3.1. It essentially contains execution units and memory. The execution units are called work items and are organized in equally sized work groups. Each work item could be understood as a processor core or a thread, in any case it is a single execution

[1] Open Computing Language

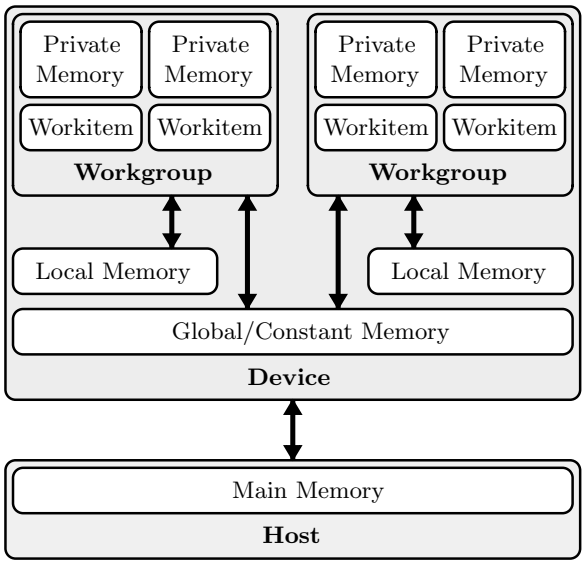


Figure 3.1: OpenCL logical structure

unit that runs in parallel with other work items. Complementary, there are four memory types or address spaces: constant, global, local and private. Constant and global memories are shared among all work items, local memory is shared only among the work items in the same work group, and private memory is independent to each work item. The difference between constant and global memory is that constant memory cannot be written by the device, only by the host. The presence and size of a register file is implementation dependent.

There is yet another memory type: texture memory; which has a global scope and it is also read-only like constant memory. It has the additional benefit of being cached, but also has some restrictions that makes it unsuitable for the nearest neighbor search.

Global and constant memory allocation and data transfers to and from the device are exclusively controlled from the host. Local and private memory is only allocatable inside a kernel and do not support dynamic allocation, they must be declared statically within the kernel source code.

Albeit the OpenCL host API and the kernel programming language are device independent, the proper tunning to obtain the maximum efficiency is still device dependent. In particular, the limit in the number of work items available or memory sizes depends on the specific hardware used.

3.2 Hardware platform

The hardware used to develop and test the GPU port was a NVIDIA GeForce GTX 295. It consists of two graphics cards bundled as a single PCIe device. Each graphics card has 240 CUDA cores, 480 overall, running at 1242 MHz. The bundle also has a total dedicated memory of 1792 megabytes split in 896 megabytes for each graphics card.

As already mentioned, each device has its own peculiarities that drive the way kernel code is developed. The NVIDIA OpenCL implementation inherits all its available optimization tricks from CUDA, with minor syntax changes. The rest of this section is dedicated to highlight the most transcendental optimization considerations.

In NVIDIA GPUs consecutive work items are grouped in so-called thread *warps*. A warp consists of 32 threads that execute every instruction synchronously. This means that no synchronization is required to ensure those 32 threads have all finished some operation. A warp can also be subdivided into two half-warps of 16 threads each.

Memory scopes in NVIDIA GPUs have different memory access latencies: the fastest memory access is to the register file. Therefore, it is generally recommended to minimize global memory access in favor of local or private access. Nevertheless, when accessing global memory, if all threads in a warp or a half-warp read or write to consecutive 32-bit words in memory[2] all memory latencies are hidden under a single memory operation. In CUDA terms, this is called memory *coalescing*. In addition, coalesced memory accesses should be aligned to 16 words, that is, 64 bytes.

Finally, local memory is scattered in 16 separate memory banks. These banks are organized so that successive 32-bit words are assigned to successive banks. Each bank can only serve one request at a time, so the ideal access to local memory of all threads in a warp should be similar to that of global memory. In the presence of bank conflicts, i.e. two threads reading or writing to the same bank, accesses are serialized, thus affecting the whole half-warp execution. There is one exception to this, when all threads in a half-warp access the same memory bank the request becomes a broadcast and no serialization is carried out.

3.3 Nearest neighbor in OpenCL

On a first approach, the idea was to merge together vectorization and OpenMP parallelization into the work group. In other words, define a two dimensional work group of 16 by 32 elements where each of the 32 rows would find the nearest neighbor of a different element from TEST. Then,

[2]For example, thread 0 reads 4 bytes at address 32, thread 1 reads 4 bytes at address 36, and so on.

calculate distance in a vectorized fashion, as in in section 2.3.2, using 16 threads, a half-warp, to access global memory with coalescing. Due to the use of multiple threads to calculate a single distance, a final reduction was needed.

Unfortunately, that first approach did not fully exploit the potential of the device. The speedup was not stable among different databases, the maximum speedup obtained with that implementation was approximately $16x$, easily beaten by the CPU using multiple cores.

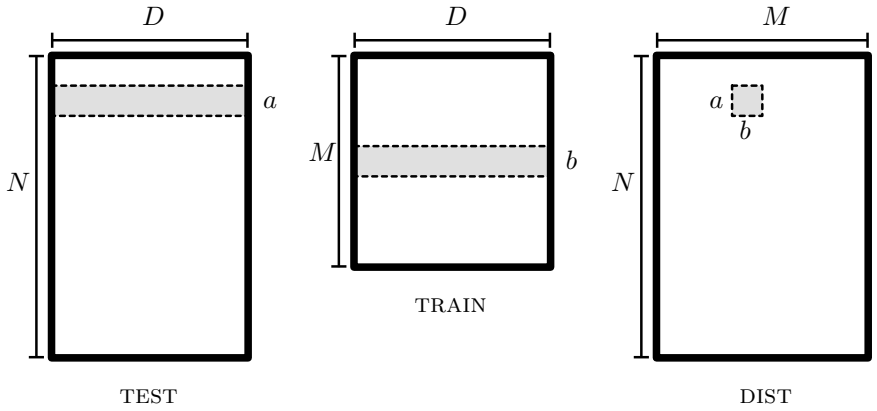


Figure 3.2: Relation between the database matrices and the distances matrix

In a change of perspective, it was found that calculating the distance of all elements in TEST with all elements in TRAIN traverses the matrices similarly to matrix multiplication where the second matrix is transposed[3]. This step also requires introducing a new matrix: the distances matrix or DIST. Each position (a, b) in the DIST matrix represents the square of the euclidean distance between element a from matrix TEST and element b from matrix TRAIN. Figure 3.2 illustrates the concept.

The memory access patterns were modified to imitate those found in the fastest GEMM implementation for CUDA [14]. This forced kernel to be separated in two parts: distance calculation and reduction. The distance calculation is the GEMM adaptation which generates the DIST matrix. Reduction is in charge of obtaining the minimum distance for each row in the DIST matrix, and the minimum distance's column location; with this information the classes are finally assigned.

Following [14], the memory layout of the matrices had to be switched from row major order to column major order, distancing features by a constant stride of N and M in the TEST and TRAIN matrices respectively.

The size of the work group was also changed from 512 to 64 (16×4), where each thread calculates the distances between a single TEST element and sixteen TRAIN elements and then stores them in DIST. Figure 3.3 shows

[3]Usually known as `sgemmnt` in linear algebra programming.

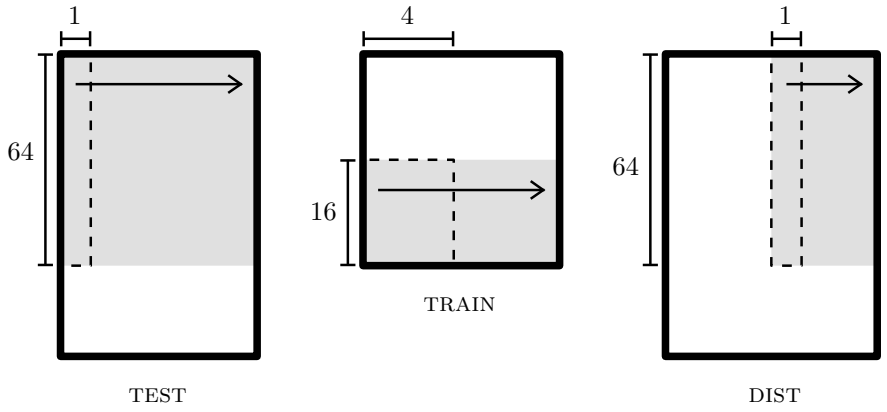


Figure 3.3: Work group layouts

the starting positions of the threads within a work group traversing trajectory. Each thread iterates over a complete TEST row, calculating its distance to the sixteen rows, fetched in blocks, from TRAIN. When finished, the computed distances are written to DIST.

By following this pattern, global memory is accessed with coalescing, there are no bank conflicts in local memory and, also important, the computations performed by each thread are completely independent, therefore eliminating the need of partial reductions. In some cases, it might be necessary to insert padding in local memory structures to avoid local memory bank conflicts; in our particular case, this trick was not necessary.

In practice, the DIST matrix can be very large. To save some memory its width (M) is divided by 16. This only affects the end of the kernel, where DIST is written. Instead of writing the sixteen computed distances, only one is kept: the minimum distance. And because the DIST column does not correspond with the TRAIN row, indexes need to be saved apart. This means that the total memory savings for DIST are:

$$\frac{N \cdot M}{16} \cdot 2 = \frac{N \cdot M}{8}$$

3.3.1 Multiple GPUs

In addition, there was a chance to spread the computation on two GPU devices (see section 3.2). Unlike using OpenMP, with OpenCL the memory had to be assigned manually to each device. As a result, the TEST and DIST matrices were split in half, sending one half to each GPU. The TRAIN matrix, though, had to be replicated in both. Figure 3.4 illustrates the situation.

The actual implementation scales automatically to the number of devices available, as reported by OpenCL.

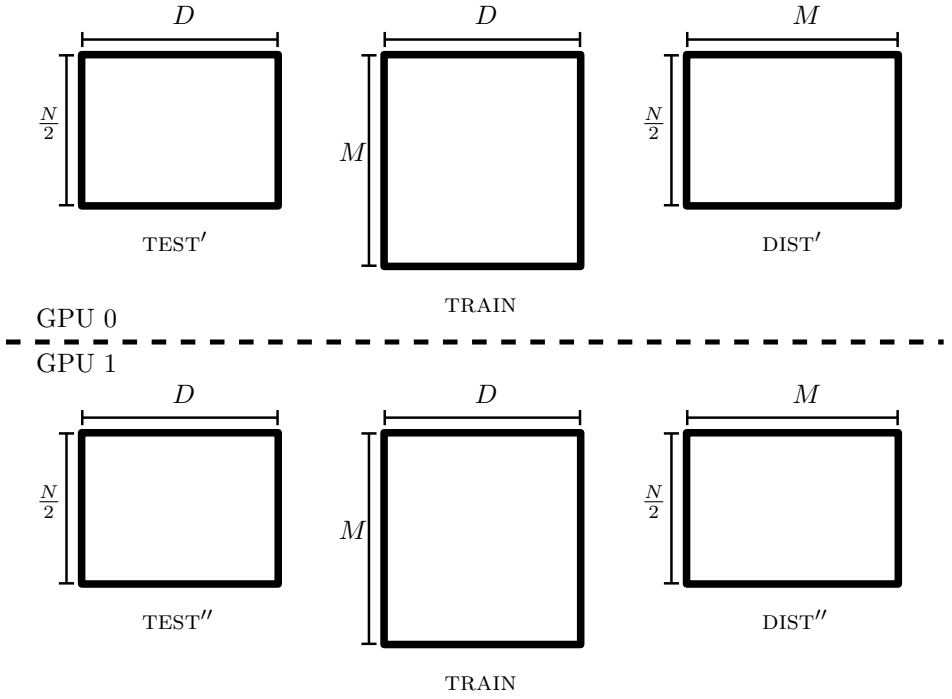


Figure 3.4: Data separation to compute with two GPU devices

3.3.2 Limitations

In order to reduce the complexity of the implementation, some assumptions were made, namely:

- Only single precision floating point is used at the moment. OpenCL theoretically supports more data types, but studying the influence of data type size on the GPU performance was not a priority for this work.

- The values for D, N and M must be multiples of 4, 64 and 16 respectively. This situation is handled by the host at database load time.

- All matrices must fit in the device's global memory. If multiple devices are used, the corresponding fractions, as represented in figure 3.4, must fit in the memory of each GPU device.

In fact, because of the last restriction, the `ijcnn1` database, the narrowest, could not be executed.

3.4 Performance results

GPU performance is also measured in NC using the same formula presented in section 2.1.2. The time used to calculate the cycles only measures kernel execution, data transfers between host and devices are excluded.

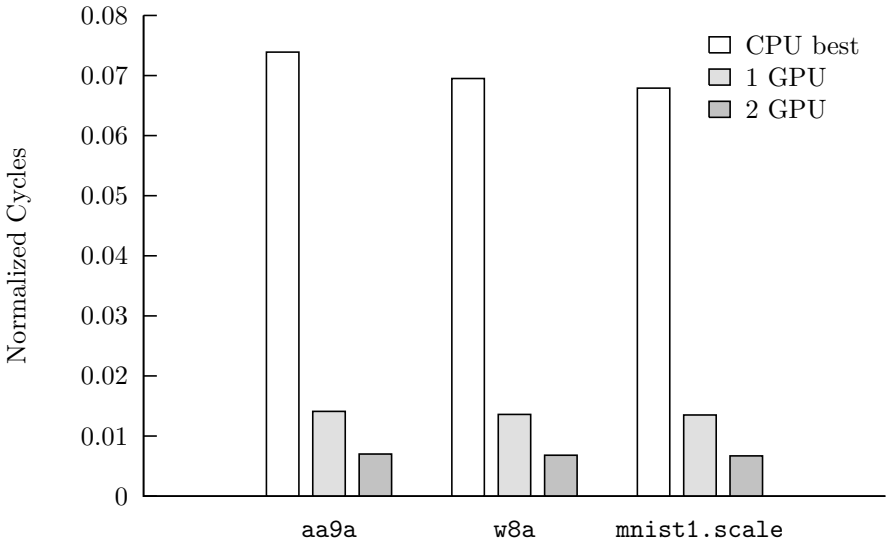


Figure 3.5: Performance comparison between CPU and GPU

Figure 3.5 shows a comparison between the best CPU implementation, with eight threads, and the OpenCL implementations using one and two devices. Apart from being quite an improvement, it should be noted that, although not visible in figure 3.5, the GPU version is also slower for narrow databases (D is small). Like it happened with the CPU version, the value of D determines the amount of iterations in the only loop each thread executes; therefore, the fewer iterations the more notable the loop overhead and thread scheduling is.

In a nutshell, table 3.1 contains the average speedups of the GPU versions over the best and worst CPU version using one and eight threads. The "worst" CPU version corresponds with algorithm 2 whereas the "best" CPU version is the combination of all optimizations described in chapter 2.

GPU	**CPU worst**		**CPU best**	
	1 thread	8 threads	1 thread	8 threads
1 device	224.45x	29.71x	38.03x	5.13x
2 devices	451.08x	59.77x	76.43x	10.30x

Table 3.1: Average speedups on the GPU

The linear scalability of the GPU can be appreciated from table 3.1, as happens with the CPU OpenMP version. Nevertheless, it can also be

appreciated that the most optimized parallel CPU version is not too far away from the GPU implementation.

Chapter 4

Other classification methods

This chapter briefly compares the nearest neighbor search against other popular classification methods. Not only performance is measured, but classification or prediction accuracy is also contrasted.

4.1 k nearest neighbors

A generalization of the nearest neighbor search is the k nearest neighbor algorithm (k-NN). An element is classified according to the class of its majority of k closest elements from a training set, using the same distance calculation methods as in the nearest neighbor.

The k-nearest neighbor is guaranteed to approach the Bayes error rate, for some value of k, where k increases as a function of the number of data points). Various improvements to k-nearest neighbor methods are possible by using proximity graphs [15].

4.2 Support Vector Machines

Support Vector Machines [16] (SVMs) are a set of related supervised learning methods used for classification and regression. In simple words, given a set of training examples, each marked as belonging to one of two categories or classes, an SVM training algorithm builds a model that predicts whether a new example falls into one class or the other. SVM can also be adapted to solve multiclass classification problems [17].

Intuitively, an SVM model is a representation of the examples as points in space, mapped so that the examples of the separate categories are divided by a clear gap that is as wide as possible. New examples are then mapped into that same space and predicted to belong to a category based on which side of the gap they fall on.

More formally, a support vector machine constructs a hyperplane or set of hyperplanes in a high or infinite dimensional space, which can be used

for classification, regression or other tasks. Intuitively, a good separation is achieved by the hyperplane that has the largest distance to the nearest training datapoints of any class (so-called functional margin), since in general the larger the margin the lower the generalization error of the classifier.

Initially, SVM were designed to create linear prediction models, that is, models that separate the hyperspace linearly. However, data may not always be linearly separable; in such cases, SVM can be adapted to solve classifier problems with the use of non-linear kernels [18].

In this chapter, two SVM software packages are benchmarked: LibLINEAR [19, 20, 21], a linear SVM classifier that does not use kernels, and LibSVM [22], a more general-purpose SVM package for classification and regression.

When using the LibSVM and LibLINEAR package, the classification process for a particular database involves four steps:

1. Scaling the data of the databases to fit in the continuous range $[0, 1)$. This is not necessary for LibLINEAR.
2. Test and cross-validate different kernel parameters in order to find the optimum values for the particular problem. This step is optional.
3. Generate a SVM model from the training set.
4. Classify the testing set with the previously generated model.

4.3 Classification accuracy

The accuracy in the classification is the percentage of correctly predicted classes. As the databases contain the correct classes for the testing set, it is possible to calculate this value. Table 4.1 contains the values for the databases, selected in section 2.1.3.

Database	1-NN	3-NN	13-NN	23-NN	LibSVM	LibLINEAR
`ijcnn1`	97.39%	97.09%	95.63%	94.71%	97.82%	91.79%
`aa9a`	79.51%	81.73%	83.69%	84.09%	85.03%	84.96%
`w8a`	97.93%	98.74%	98.36%	94.48%	99.18%	90.54%
`mnist1`	95.71%	95.93%	95.24%	98.15%	97.70%	90.07%

Table 4.1: Accuracy results

From these results, it is difficult to conclude if k-NN, with $k > 1$, is really beneficial in terms of quality in classification. LibSVM is the only classifier that provides both accuracy stability and the best results in terms of quality.

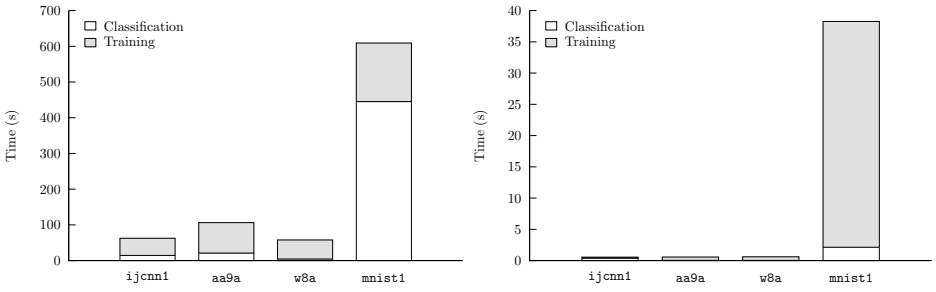
The SVM implementations show higher accuracy, specially LibSVM. It should be noted that the cross-validation step was carried out for LibSVM

but not for LibLINEAR. In other words, LibLINEAR was executed with the default solving method (dual) and default parameters, whereas in the LibSVM case the utilized kernel was also the default (radial basis function), but a previous cross-validation process led to the parameter values that give the best accuracy.

4.4 Performance comparison

The k-NN implementation was not optimized so throughly as in the $k = 1$ case. It only features a block algorithm. Comparing k-NN against NN results in an approximate 3% overhead for k-NN.

When measuring the performances of LibSVM, the cross-validation phase has not been counted. It it had, the times would be much higher as looking for the best kernel parameter is a very time consuming task. This is one of the arguments that favor the usage of k-NN as a classification algorithm over SVM [23]. LibLINEAR test did not go over the cross-validation step.

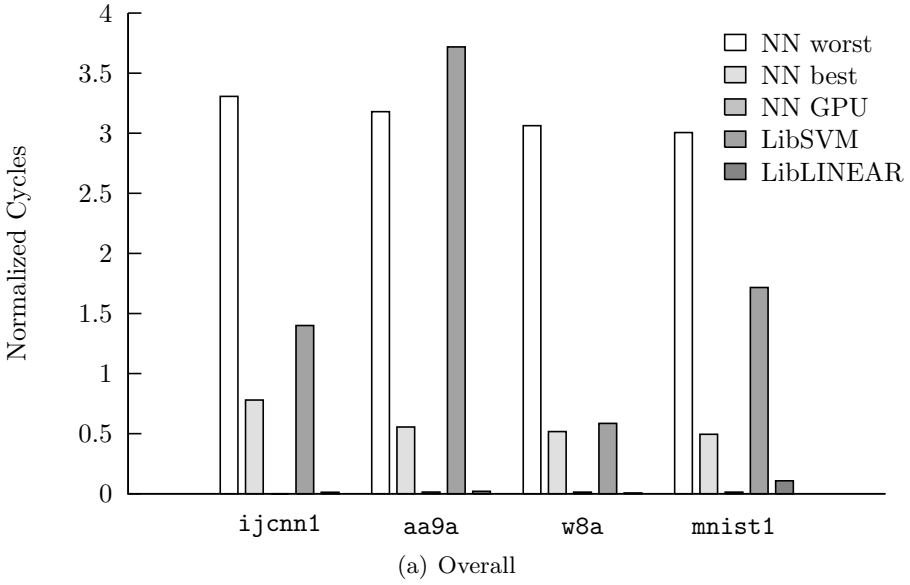


(a) LibSVM times (b) LibLINEAR times

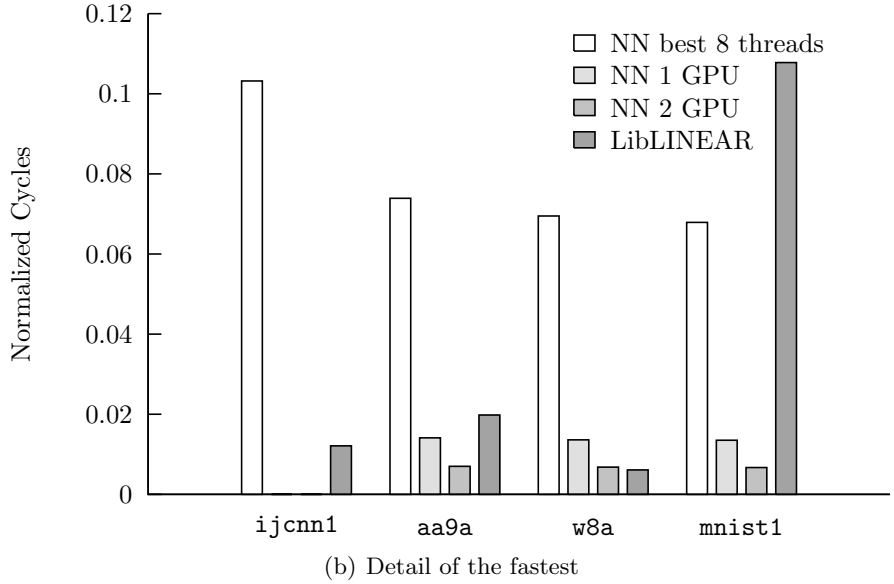
Figure 4.1: SVM implementations comparison

Figure 4.1 show the time spent in model generation, i.e. training phase, over the time spent in classifying. The difference in time between LibSVM, figure 4.1(a), and LibLINEAR, 4.1(b), is notorious. In fact, LibLINEAR is very efficient on large databases. This is in part thanks to the extremely small and condensed model it creates from the training set; it also benefits from not having to solve complex kernels. The drawback is that it only predicts accurately on data sets that are linearly separable.

In figure 4.2 some NN implementations are contrasted with LibSVM and LibLINEAR. Some implementations, such as LibLINEAR and the GPU NN, in figure 4.2(a) are almost invisible; figure 4.2(b) contains a detail of the fastest classification implementations mentioned so far. Notice how LibLINEAR rivals with the GPU ports. This is indeed interesting as the LibLINEAR code was running in the CPU using a single thread.



(a) Overall



(b) Detail of the fastest

Figure 4.2: Comparison between different classification methods

Chapter 5

Conclusions

We have shown how memory access can be influential in algorithm performance and stability. Having some consciousness and knowledge of the memory hierarchy can reduce processor stalls waiting for data.

Also, we have taken advantage of the different parallelism granularities offered by current architectures. Despite the growing availability of multiple cores, there is still room for sequential improvement before resorting to a parallel programming model. When taking advantage of SIMD processor extensions, smaller data types yield improvements of about an order of magnitude by giving the possibility of executing more operations at the same cost.

In addition, we have seen the huge potential that GPGPU computing can bring, at the cost of a less flexible programming environment. With the GPU port, we learned that a proper memory access pattern is a fundamental change to get the most out of the hardware. We conceived three basic guidelines to follow when looking for a better way use memory on NVIDIA graphics cards:

1. Use coalescing global memory access as much as possible.
2. Avoid memory bank conflicts in local memory.
3. Minimize thread inter-dependencies, e.g. reductions.

Simple algorithms and implementations are generally easier to analyze and manipulate than those more sophisticated. When it comes to adapting them to new and less mature architectures such as the GPU, simplicity is an advantage.

Apart from the hardware performance aspect, the actual numerical results were not clearly heading towards a unique conclusion. The trade offs between quality of results and time involved are very sensible to the nature of data. In some situations, a simple brute force approach might be good enough or even better than a more sophisticated solution. However, cleverer

designs like SVM could compensate the necessary initial time investment in the long term. In no case a single solution fits all.

Nevertheless, these facts cannot hide the obvious fact that the LibLINEAR classifier is algorithmically superior to NN. After some simple experiments, we concluded that the only way to be competitive with LibLINEAR was to reduce the TRAIN matrix to something as small as the model created by LibLINEAR, while still preserving an acceptable classification accuracy.

5.1 Future work

Some ideas could be further developed. For example, the described blocking implementation does not cover extremely wide database, e.g. thousands of dimensions, which is a common case in some usage scenarios. The change would require some thought and, specially, more intermediate memory for unfinished data calculations. The k-NN implementation could be optimized similarly to the NN.

On the GPU side, its limitations could be overcome by subdividing data and overlapping data transfers between host and device, with computations. Even the GPU plus CPU combination hold great potential, after checking if floating point arithmetic is fully compatible. Studying the influence of data type size on the GPU performance is also interesting. A more precise analytical model could be designed and tested for the GPU.

Experiments mentioned at the end of previous section also constitute an investigation path to follow. Specially since condensing the TRAIN matrix introduces a new phase, similar to the training phase in SVM, which is also subject to optimizations.

The implementations presented in this work could also be benchmarked against alternative nearest neighbor approaches. Converting the tool, NN as a classification mechanism, to a problem itself, i.e. finding the nearest neighbor.

Finally, for databases that contain only discrete data, distances other than the euclidean could be tested for classification accuracy and performance.

Appendix A

Database properties

The following table contains the attributes for all the databases used in this work.

Label	D	M	N	**Floats**	**Classes**
aa1a	122	1605	30955	no	2
aa2a	122	2265	30295	no	2
aa3a	122	3185	29375	no	2
aa4a	122	4781	27779	no	2
aa5a	122	6414	26146	no	2
aa6a	122	11220	21340	no	2
aa7a	122	16100	16460	no	2
aa8a	122	22695	9865	no	2
aa9a	122	32560	16281	no	2
acoustic	50	78823	19705	yes	3
combined	100	78823	19705	yes	3
dna	180	2000	1186	no	3
ijcnn1	22	49990	91701	yes	2
letter	16	15000	5000	yes	26
mnist1	780	21000	49000	yes	10
protein	357	14895	6621	yes	3
satimage	36	4435	2000	yes	6
seismic	50	78823	19705	yes	3
shuttle	9	43500	14500	yes	7
splice	60	1000	2175	no	2
svmguide1	4	3089	4000	yes	2
svmguide3	22	1243	41	yes	2
usps	256	7291	2007	yes	10
vowel	10	528	462	yes	11
w1a	300	2477	47272	no	2
w7a	300	24692	25057	no	2
w8a	300	49749	14951	no	2

Appendix B

NN Executions results

The following table contains the NC measurements for a selection of optimization combinations with some databases. Each row lists the executions from one to eight threads. Optimization combinations are encoded as follows:

B from 0 to 2 to describe the blocking level.

U as 0, 2 or 4 to describe the unrolling degree.

S denotes scalar.

V denotes vectorized.

Combination	1T	2T	3T	4T	5T	6T	7T	8T
`aa2a`								
U0 S B0 byte	2.9240	1.4704	0.9804	0.7373	0.6175	0.5147	0.4420	0.3860
U0 S B2 byte	2.9207	1.4604	0.9751	0.7313	0.6168	0.5170	0.4418	0.3856
U0 S B0 double	3.3365	1.6683	1.1168	0.8400	0.7074	0.5910	0.5046	0.4397
U0 S B2 double	3.2952	1.6494	1.1061	0.8318	0.6961	0.5798	0.5001	0.4356
U0 S B0 float	3.0443	1.5307	1.0162	0.7651	0.6425	0.5385	0.4604	0.4016
U0 S B2 float	3.0265	1.5137	1.0149	0.7584	0.6399	0.5333	0.4600	0.4002
U0 S B0 int	2.9164	1.4567	0.9728	0.7295	0.6152	0.5153	0.4409	0.3845
U0 S B2 int	2.8995	1.4572	0.9674	0.7261	0.6165	0.5133	0.4389	0.3831
U0 S B0 short	3.1538	1.5769	1.0525	0.7951	0.6694	0.5581	0.5088	0.4161
U0 S B2 short	3.1427	1.5716	1.0479	0.7904	0.6673	0.5533	0.5003	0.4151
U0 V B0 byte	0.3678	0.1846	0.1235	0.0931	0.0784	0.0652	0.0560	0.0486
U0 V B2 byte	0.3642	0.1826	0.1216	0.0962	0.0769	0.0645	0.0556	0.0481
U0 V B0 double	1.6677	0.8337	0.5582	0.4185	0.3529	0.2931	0.2539	0.2200
U0 V B2 double	1.6114	0.8074	0.5406	0.4059	0.3408	0.2843	0.2448	0.2133
U0 V B0 float	0.8664	0.4332	0.2893	0.2179	0.1840	0.1534	0.1318	0.1146
U0 V B2 float	0.8425	0.4213	0.2824	0.2119	0.1782	0.1484	0.1281	0.1114
U0 V B0 int	1.0893	0.5452	0.3633	0.2726	0.2298	0.1926	0.1652	0.1437
U0 V B2 int	1.0664	0.5322	0.3555	0.2680	0.2251	0.1888	0.1619	0.1409
U0 V B0 short	0.5007	0.2498	0.1685	0.1256	0.1054	0.0878	0.0760	0.0659
U0 V B2 short	0.4874	0.2860	0.1630	0.1224	0.1028	0.0862	0.0740	0.0643
U4 S B0 byte	2.1787	1.0887	0.7264	0.5475	0.4604	0.3834	0.3293	0.2874
U4 S B2 byte	2.1828	1.0917	0.7336	0.5512	0.4607	0.3843	0.3304	0.2880
U4 S B0 double	2.0883	1.0412	0.6941	0.5244	0.4393	0.3659	0.3157	0.2757
U4 S B2 double	2.0076	1.0042	0.6688	0.5045	0.4253	0.3542	0.3046	0.2657

Combination	1T	2T	3T	4T	5T	6T	7T	8T
U4 S B0 float	1.9924	0.9959	0.6642	0.4987	0.4220	0.3526	0.3017	0.2630
U4 S B2 float	1.9523	0.9753	0.6505	0.4913	0.4122	0.3434	0.2952	0.2580
U4 S B0 int	1.9676	0.9838	0.6598	0.4927	0.4156	0.3484	0.2978	0.2598
U4 S B2 int	1.9412	0.9701	0.6525	0.4862	0.4118	0.3440	0.2941	0.2566
U4 S B0 short	2.2622	1.1311	0.7584	0.5662	0.4775	0.4002	0.3417	0.2984
U4 S B2 short	2.2474	1.1238	0.7507	0.5651	0.4745	0.3957	0.3401	0.2969
U4 V B0 byte	0.2805	0.1405	0.0940	0.0704	0.0598	0.0496	0.0426	0.0371
U4 V B2 byte	0.2777	0.1395	0.0929	0.0702	0.0589	0.0491	0.0423	0.0368
U4 V B0 double	1.1193	0.5656	0.3750	0.2826	0.2418	0.2011	0.1846	0.1615
U4 V B2 double	1.0273	0.5139	0.3433	0.2580	0.2190	0.1809	0.1558	0.1365
U4 V B0 float	0.5987	0.2994	0.2017	0.1507	0.1279	0.1066	0.0967	0.0846
U4 V B2 float	0.5553	0.2802	0.1861	0.1399	0.1176	0.0992	0.0849	0.0734
U4 V B0 int	0.7762	0.3856	0.2594	0.1944	0.1636	0.1372	0.1188	0.1043
U4 V B2 int	0.7331	0.3660	0.2457	0.1847	0.1555	0.1296	0.1110	0.0967
U4 V B0 short	0.3838	0.1918	0.1281	0.0964	0.0812	0.0675	0.0585	0.0512
U4 V B2 short	0.3722	0.1872	0.1252	0.0936	0.0789	0.0661	0.0565	0.0493
aa9a								
U0 S B0 byte	2.9300	1.4734	0.9821	0.7335	0.6175	0.5186	0.4432	0.3867
U0 S B2 byte	2.9207	1.4606	0.9746	0.7368	0.6168	0.5140	0.4418	0.3856
U0 S B0 double	3.6062	1.8362	1.2585	0.9160	0.8519	0.6956	0.6558	0.5629
U0 S B2 double	3.2921	1.6463	1.1041	0.8283	0.7005	0.5837	0.4996	0.4362
U0 S B0 float	3.1797	1.6019	1.0658	0.8122	0.6802	0.5669	0.4852	0.4233
U0 S B2 float	3.0265	1.5135	1.0136	0.7585	0.6394	0.5330	0.4582	0.4001
U0 S B0 int	3.0575	1.5390	1.0335	0.7789	0.6487	0.5451	0.4661	0.4058
U0 S B2 int	2.8966	1.4486	0.9660	0.7286	0.6156	0.5102	0.4380	0.3829
U0 S B0 short	3.1895	1.6054	1.0677	0.7987	0.6708	0.5612	0.5113	0.4199
U0 S B2 short	3.1430	1.5717	1.0494	0.7903	0.6685	0.5533	0.4896	0.4150
U0 V B0 byte	0.3755	0.1864	0.1240	0.0939	0.0787	0.0654	0.0563	0.0492
U0 V B2 byte	0.3630	0.1822	0.1211	0.0913	0.0767	0.0640	0.0553	0.0479
U0 V B0 double	2.1047	1.4529	1.1736	0.8123	0.7383	0.6111	0.6037	0.5081
U0 V B2 double	1.6079	0.8044	0.5398	0.4037	0.3406	0.2859	0.2453	0.2140
U0 V B0 float	1.0680	0.7117	0.5186	0.4025	0.3664	0.2966	0.2954	0.2449
U0 V B2 float	0.8399	0.4203	0.2804	0.2107	0.1778	0.1482	0.1285	0.1113
U0 V B0 int	1.2835	0.7203	0.5589	0.4212	0.3840	0.3068	0.3034	0.2526
U0 V B2 int	1.0625	0.5312	0.3548	0.2663	0.2263	0.1884	0.1616	0.1406
U0 V B0 short	0.5594	0.3068	0.2125	0.1607	0.1404	0.1148	0.1071	0.0873
U0 V B2 short	0.4863	0.2429	0.1622	0.1217	0.1029	0.0855	0.0738	0.0642
U4 S B0 byte	2.1781	1.0892	0.7282	0.5454	0.4599	0.3833	0.3295	0.2875
U4 S B2 byte	2.1836	1.0918	0.7277	0.5469	0.4611	0.3866	0.3305	0.2883
U4 S B0 double	3.3036	1.9560	1.3622	0.9617	0.8739	0.6506	0.6646	0.5782
U4 S B2 double	2.0049	1.0044	0.6729	0.5063	0.4280	0.3554	0.3047	0.2670
U4 S B0 float	2.4417	1.3339	1.0016	0.7653	0.5389	0.5286	0.4501	0.3991
U4 S B2 float	1.9477	0.9749	0.6509	0.4887	0.4145	0.3453	0.2958	0.2579
U4 S B0 int	2.4072	1.3281	0.8997	0.7781	0.5302	0.5406	0.4670	0.4106
U4 S B2 int	1.9393	0.9701	0.6497	0.4879	0.4128	0.3426	0.2940	0.2564
U4 S B0 short	2.3381	1.1701	0.7792	0.5856	0.4940	0.4101	0.3507	0.3061
U4 S B2 short	2.2474	1.1245	0.7524	0.5651	0.4748	0.3958	0.3406	0.2968
U4 V B0 byte	0.2847	0.1431	0.0961	0.0721	0.0603	0.0501	0.0434	0.0378
U4 V B2 byte	0.2782	0.1392	0.0928	0.0698	0.0588	0.0490	0.0423	0.0374
U4 V B0 double	2.3810	1.5225	1.1219	0.7966	0.7290	0.5718	0.5810	0.5032
U4 V B2 double	1.0237	0.5129	0.3440	0.2587	0.2181	0.1829	0.1567	0.1372
U4 V B0 float	1.1965	0.8094	0.5554	0.4044	0.3607	0.2870	0.2855	0.2373
U4 V B2 float	0.5556	0.2781	0.1856	0.1394	0.1176	0.0983	0.0845	0.0739
U4 V B0 int	1.4129	0.8837	0.5986	0.4298	0.3822	0.3067	0.2998	0.2432
U4 V B2 int	0.7315	0.3681	0.2441	0.1850	0.1548	0.1300	0.1112	0.0970
U4 V B0 short	0.4640	0.2972	0.1907	0.1566	0.1293	0.1090	0.0975	0.0828
U4 V B2 short	0.3723	0.1862	0.1248	0.0951	0.0787	0.0658	0.0575	0.0492
acoustic								
U0 S B0 double	3.5128	1.7763	1.2802	0.9357	0.7951	0.7010	0.6496	0.5563
U0 S B2 double	3.1911	1.5940	1.0631	0.8018	0.6735	0.5630	0.4836	0.4226
U0 S B0 float	3.2592	1.6416	1.0955	0.8265	0.6961	0.5813	0.4958	0.4334

Combination	1T	2T	3T	4T	5T	6T	7T	8T
U0 S B2 float	3.1012	1.5509	1.0353	0.7764	0.6586	0.5458	0.4695	0.4106
U0 V B0 double	2.1509	1.4502	1.0683	0.8357	0.7412	0.6149	0.6039	0.5082
U0 V B2 double	1.6778	0.8396	0.5599	0.4237	0.3551	0.2960	0.2585	0.2280
U0 V B0 float	1.1575	0.7111	0.5379	0.4147	0.3699	0.3019	0.2986	0.2502
U0 V B2 float	0.9453	0.4726	0.3152	0.2382	0.2013	0.1665	0.1455	0.1273
U4 S B0 double	3.1747	1.8688	1.2040	0.8639	0.8224	0.6756	0.6564	0.5351
U4 S B2 double	2.0080	1.0051	0.6736	0.5063	0.4274	0.3551	0.3055	0.2681
U4 S B0 float	2.2323	1.1465	0.8030	0.5987	0.4968	0.4028	0.3619	0.3100
U4 S B2 float	1.9584	0.9748	0.6501	0.4903	0.4116	0.3451	0.2958	0.2575
U4 V B0 double	2.3684	1.6115	1.0833	0.8340	0.7419	0.6050	0.5885	0.4938
U4 V B2 double	1.0939	0.5483	0.3678	0.2759	0.2319	0.1941	0.1688	0.1498
U4 V B0 float	1.0097	0.7126	0.5490	0.4075	0.3343	0.3019	0.2957	0.2443
U4 V B2 float	0.6505	0.3238	0.2159	0.1632	0.1368	0.1155	0.0990	0.0860
`combined`								
U0 S B0 double	3.6333	1.8663	1.2767	1.0018	0.8622	0.7070	0.6632	0.5739
U0 S B2 double	3.3076	1.6551	1.1100	0.8296	0.7046	0.5832	0.5024	0.4382
U0 S B0 float	3.2546	1.6348	1.0820	0.8249	0.6954	0.5755	0.4919	0.4297
U0 S B2 float	3.0726	1.5362	1.0246	0.7693	0.6490	0.5424	0.4649	0.4059
U0 V B0 double	2.1089	1.4793	1.1806	0.8513	0.7361	0.6223	0.6062	0.5176
U0 V B2 double	1.6179	0.8091	0.5419	0.4062	0.3428	0.2859	0.2495	0.2212
U0 V B0 float	1.0905	0.7185	0.5344	0.4176	0.3798	0.3082	0.3030	0.2551
U0 V B2 float	0.8580	0.4315	0.2875	0.2151	0.1815	0.1513	0.1326	0.1167
U4 S B0 double	3.3301	2.0756	1.3979	0.9993	0.8976	0.6596	0.6759	0.5335
U4 S B2 double	2.0199	1.0108	0.6783	0.5094	0.4286	0.3582	0.3082	0.2685
U4 S B0 float	2.4895	1.3582	0.9042	0.7174	0.5554	0.5141	0.4250	0.3897
U4 S B2 float	1.9618	0.9853	0.6609	0.4955	0.4173	0.3491	0.2986	0.2606
U4 V B0 double	2.3911	1.7309	1.1070	0.8604	0.7560	0.6166	0.5963	0.5023
U4 V B2 double	1.0359	0.5189	0.3462	0.2608	0.2203	0.1837	0.1605	0.1426
U4 V B0 float	1.1581	0.8005	0.4802	0.4282	0.3792	0.3026	0.2950	0.2484
U4 V B2 float	0.5636	0.2823	0.1883	0.1421	0.1194	0.1006	0.0871	0.0763
`ijcnn1`								
U0 S B0 double	3.5632	1.8426	1.2239	0.9154	0.7755	0.6431	0.5608	0.4871
U0 S B2 double	3.3319	1.6665	1.1126	0.8348	0.7036	0.5866	0.5059	0.4402
U0 S B0 float	3.3064	1.6522	1.1007	0.8223	0.6932	0.5780	0.4968	0.4343
U0 S B2 float	3.2558	1.6375	1.0892	0.8151	0.6875	0.5760	0.4928	0.4310
U0 V B0 double	2.1405	1.2335	0.8342	0.6616	0.6803	0.4715	0.4365	0.3611
U0 V B2 double	1.8440	0.9218	0.6182	0.4633	0.3893	0.3245	0.2813	0.2463
U0 V B0 float	1.1847	0.5923	0.3964	0.2953	0.2492	0.2082	0.1795	0.1576
U0 V B2 float	1.1451	0.5761	0.3840	0.2870	0.2419	0.2027	0.1745	0.1517
U4 S B0 double	2.5334	1.4120	0.9280	0.7269	0.6306	0.5192	0.4850	0.3985
U4 S B2 double	2.1154	1.0577	0.7059	0.5296	0.4464	0.3720	0.3217	0.2796
U4 S B0 float	2.1001	1.0515	0.7004	0.5294	0.4428	0.3691	0.3170	0.2765
U4 S B2 float	2.0735	1.0422	0.6938	0.5190	0.4379	0.3649	0.3148	0.2738
U4 V B0 double	1.7913	1.1152	0.7275	0.6029	0.5270	0.4472	0.4279	0.3562
U4 V B2 double	1.2466	0.6227	0.4155	0.3135	0.2632	0.2205	0.1901	0.1651
U4 V B0 float	0.8258	0.4239	0.2820	0.2133	0.1808	0.1517	0.1302	0.1149
U4 V B2 float	0.7802	0.3904	0.2600	0.1964	0.1647	0.1382	0.1196	0.1032
`mnist1`								
U0 S B0 double	3.4507	1.7557	1.2590	0.8984	0.8627	0.7147	0.6553	0.5678
U0 S B2 double	3.1221	1.5623	1.0429	0.7840	0.6707	0.5576	0.4793	0.4189
U0 S B0 float	3.0054	1.5212	1.0087	0.7700	0.6482	0.5275	0.4529	0.3957
U0 S B2 float	2.8393	1.4278	0.9470	0.7142	0.6024	0.5001	0.4313	0.3754
U0 V B0 double	2.0714	1.5159	1.1557	0.8526	0.7654	0.6291	0.6133	0.5215
U0 V B2 double	1.5884	0.7979	0.5385	0.4053	0.3477	0.2890	0.2492	0.2183
U0 V B0 float	1.0334	0.7427	0.5306	0.4293	0.3804	0.3110	0.3025	0.2575
U0 V B2 float	0.7825	0.3924	0.2633	0.1979	0.1681	0.1400	0.1213	0.1054
U4 S B0 double	2.2696	1.4295	1.1266	0.8498	0.7435	0.5979	0.5670	0.4822
U4 S B2 double	1.9505	0.9826	0.6569	0.4927	0.4227	0.3519	0.3009	0.2658
U4 S B0 float	2.0923	1.0904	0.7157	0.6002	0.4822	0.4178	0.3463	0.3118
U4 S B2 float	1.8855	0.9477	0.6294	0.4740	0.4006	0.3332	0.2865	0.2501
U4 V B0 double	1.5976	1.3565	1.0165	0.7581	0.6894	0.5485	0.5252	0.4501

Combination	1T	2T	3T	4T	5T	6T	7T	8T
U4 V B2 double	0.9927	0.5007	0.3368	0.2551	0.2182	0.1838	0.1584	0.1405
U4 V B0 float	0.9449	0.6983	0.5192	0.3737	0.3435	0.2732	0.2638	0.2314
U4 V B2 float	0.4949	0.2504	0.1664	0.1258	0.1069	0.0899	0.0771	0.0679
protein								
U0 S B0 double	3.4778	1.7687	1.2703	0.8936	0.8551	0.7039	0.6530	0.5606
U0 S B2 double	3.1562	1.5798	1.0611	0.7959	0.6742	0.5603	0.4812	0.4213
U0 S B0 float	3.0697	1.5425	1.0232	0.7750	0.6547	0.5410	0.4627	0.4040
U0 S B2 float	2.8882	1.4524	0.9653	0.7235	0.6152	0.5095	0.4378	0.3827
U0 V B0 double	2.1104	1.4744	1.0343	0.9024	0.7582	0.6173	0.6034	0.5111
U0 V B2 double	1.6202	0.8127	0.5431	0.4113	0.3471	0.2922	0.2516	0.2207
U0 V B0 float	1.0611	0.7200	0.5269	0.4074	0.3721	0.2999	0.3008	0.2513
U0 V B2 float	0.8189	0.4104	0.2742	0.2057	0.1743	0.1455	0.1259	0.1095
U4 S B0 double	3.0062	1.8078	1.3726	0.9574	0.7696	0.6677	0.6111	0.5497
U4 S B2 double	1.9622	0.9834	0.6609	0.4957	0.4198	0.3533	0.3024	0.2634
U4 S B0 float	2.2699	1.2056	0.8197	0.7328	0.4866	0.5033	0.4373	0.3812
U4 S B2 float	1.8982	0.9504	0.6380	0.4785	0.4020	0.3357	0.2884	0.2521
U4 V B0 double	2.1546	1.4162	1.0739	0.7144	0.7010	0.5433	0.5522	0.4773
U4 V B2 double	0.9985	0.5021	0.3369	0.2542	0.2156	0.1817	0.1563	0.1372
U4 V B0 float	1.1879	0.6784	0.5545	0.3624	0.3486	0.2735	0.2839	0.2361
U4 V B2 float	0.5121	0.2586	0.1720	0.1295	0.1097	0.0918	0.0789	0.0691
satimage								
U0 S B0 double	3.2776	1.6378	1.0928	0.8201	0.6915	0.5780	0.4973	0.4327
U0 S B2 double	3.2455	1.6232	1.0840	0.8583	0.6870	0.5740	0.4936	0.4306
U0 S B0 float	3.1727	1.5955	1.0615	0.7972	0.6699	0.5623	0.4823	0.4190
U0 S B2 float	3.1595	1.5876	1.0566	0.7927	0.6681	0.5586	0.4785	0.4184
U0 V B0 double	1.7999	0.8963	0.6008	0.4755	0.3787	0.3164	0.2723	0.2390
U0 V B2 double	1.7395	0.8723	0.5850	0.4612	0.3714	0.3140	0.2664	0.2353
U0 V B0 float	1.0516	0.5258	0.3511	0.2772	0.2253	0.1853	0.1596	0.1389
U0 V B2 float	1.0335	0.5200	0.3461	0.2625	0.2194	0.1840	0.1572	0.1375
U4 S B0 double	2.1110	1.0564	0.7082	0.5309	0.4467	0.3735	0.3201	0.2796
U4 S B2 double	2.0504	1.0272	0.6843	0.5443	0.4324	0.3631	0.3116	0.2717
U4 S B0 float	2.0320	1.0095	0.6775	0.5352	0.4262	0.3559	0.3064	0.2665
U4 S B2 float	1.9872	0.9959	0.6669	0.4989	0.4203	0.3526	0.3018	0.2633
U4 V B0 double	1.2337	0.6168	0.4138	0.3305	0.2813	0.2221	0.1986	0.1737
U4 V B2 double	1.1565	0.5776	0.3867	0.3069	0.2492	0.2045	0.1781	0.1544
U4 V B0 float	0.7329	0.3665	0.2458	0.1952	0.1592	0.1335	0.1138	0.0995
U4 V B2 float	0.6932	0.3461	0.2327	0.1838	0.1488	0.1236	0.1051	0.0930
shuttle								
U0 S B0 double	3.7409	1.8748	1.2501	0.9450	0.7911	0.6632	0.5667	0.4945
U0 S B2 double	3.6841	1.8408	1.2277	0.9255	0.7775	0.6480	0.5583	0.4860
U0 S B0 float	3.6418	1.8311	1.2166	0.9126	0.7693	0.6442	0.5516	0.4809
U0 S B2 float	3.6243	1.8118	1.2096	0.9078	0.7652	0.6410	0.5483	0.4786
U0 V B0 double	2.2500	1.1240	0.7525	0.5662	0.4742	0.3954	0.3444	0.3008
U0 V B2 double	2.1747	1.0937	0.7290	0.5480	0.4589	0.3827	0.3326	0.2913
U0 V B0 float	1.4831	0.7413	0.4988	0.3741	0.3129	0.2608	0.2251	0.1956
U0 V B2 float	1.4609	0.7308	0.4899	0.3658	0.3084	0.2584	0.2223	0.1928
U4 S B0 double	2.4415	1.2219	0.8162	0.6102	0.5148	0.4289	0.3704	0.3229
U4 S B2 double	2.3704	1.1915	0.7938	0.5934	0.5002	0.4196	0.3604	0.3137
U4 S B0 float	2.3496	1.1680	0.7785	0.5891	0.4938	0.4110	0.3553	0.3084
U4 S B2 float	2.3221	1.1614	0.7743	0.5837	0.4903	0.4108	0.3528	0.3066
U4 V B0 double	1.6257	0.8106	0.5447	0.4068	0.3443	0.2870	0.2530	0.2202
U4 V B2 double	1.5377	0.7728	0.5146	0.4058	0.3243	0.2720	0.2351	0.2056
U4 V B0 float	1.0817	0.5406	0.3628	0.2718	0.2292	0.1911	0.1650	0.1427
U4 V B2 float	1.0632	0.5319	0.3569	0.2674	0.2247	0.1872	0.1642	0.1410
splice								
U0 S B0 byte	2.8899	1.4507	0.9640	0.7629	0.6100	0.5090	0.4381	0.3815
U0 S B2 byte	2.8890	1.4516	0.9640	0.7243	0.6100	0.5090	0.4364	1.0696
U0 S B0 double	3.2196	1.6112	1.0796	0.8493	0.6794	0.5667	0.4886	0.4249
U0 S B2 double	3.1808	1.5924	1.1194	0.8017	0.6728	0.5645	0.4835	0.4208
U0 S B0 float	3.0901	1.5500	1.0318	0.7762	0.6530	0.5486	0.4701	0.4083
U0 S B2 float	3.0887	1.5543	1.0308	0.8015	0.6553	0.5469	0.4692	0.4080

Combination	1T	2T	3T	4T	5T	6T	7T	8T
U0 S B0 int	2.7392	1.3696	0.9150	0.6921	0.5813	0.4822	1.0270	1.2207
U0 S B2 int	2.7348	1.3770	0.9143	0.7222	0.5778	0.4818	0.4147	0.3613
U0 S B0 short	3.6626	1.8309	1.2605	0.9619	0.7768	0.6484	0.5553	0.4831
U0 S B2 short	3.6624	1.8411	1.2197	0.9173	0.7723	0.6450	0.5552	0.4833
U0 V B0 byte	0.4562	0.2282	0.1523	0.1236	0.0962	1.6010	0.0772	0.0677
U0 V B2 byte	0.4547	0.2315	0.1604	0.1140	0.0962	0.9129	0.8125	0.0675
U0 V B0 double	1.7314	0.8623	0.5761	0.4595	0.3638	0.3061	0.2619	0.2281
U0 V B2 double	1.6724	0.8384	0.5595	0.4478	0.3586	0.2964	0.2541	0.2219
U0 V B0 float	0.9429	0.4743	0.3173	0.2495	0.2004	0.1665	0.1440	0.1279
U0 V B2 float	0.9361	0.4727	0.3138	0.2508	0.1979	0.1660	0.1419	0.1272
U0 V B0 int	1.1593	0.5790	0.3885	0.3065	0.2475	0.3560	0.1755	0.1533
U0 V B2 int	1.1510	0.5821	0.3840	0.3043	0.2455	0.2047	0.1746	0.1526
U0 V B0 short	0.5517	0.2763	0.1949	0.1458	0.1167	0.0984	0.0835	0.0731
U0 V B2 short	0.5524	0.2776	0.1945	0.1457	0.1182	0.0972	0.0843	1.6247
U4 S B0 byte	2.1495	1.0748	0.7229	0.5711	0.4536	0.5049	0.3264	0.2840
U4 S B2 byte	2.1491	1.0755	0.7185	0.5673	0.4557	0.3811	0.3264	0.2839
U4 S B0 double	2.0982	1.0501	0.7044	0.5269	0.4424	0.3743	0.3178	0.2787
U4 S B2 double	2.0034	1.0028	0.6749	0.5289	0.4244	0.3555	0.3031	0.2660
U4 S B0 float	1.9564	0.9784	0.6539	0.4907	0.4144	0.3444	0.2969	0.6190
U4 S B2 float	1.9433	0.9716	0.6511	0.5125	0.4111	0.3421	0.2950	0.2567
U4 S B0 int	1.9671	0.9785	0.6579	0.4906	0.4143	0.3493	0.2968	0.2584
U4 S B2 int	1.9469	0.9767	0.6565	0.5228	0.4111	0.3455	0.2952	0.2571
U4 S B0 short	2.2407	1.1201	0.7496	0.5936	0.4724	0.3979	0.3378	0.2959
U4 S B2 short	2.2402	1.1204	0.7538	0.5616	0.4790	0.3942	0.3397	1.2368
U4 V B0 byte	0.3459	0.1732	0.1154	0.0915	0.0731	0.0610	0.0524	0.0466
U4 V B2 byte	0.3461	0.1760	0.1155	0.0924	0.0745	0.0611	0.0645	1.6246
U4 V B0 double	1.1936	0.5978	0.4010	0.3205	0.2573	0.2121	0.1943	0.1702
U4 V B2 double	1.0837	0.5485	0.3669	0.2919	0.2308	0.1935	0.1651	0.1436
U4 V B0 float	0.6468	0.3245	0.2166	0.1705	0.1369	0.1148	0.1058	0.0954
U4 V B2 float	0.6398	0.3237	0.2146	0.1731	0.1376	0.1141	0.0976	0.0883
U4 V B0 int	0.8126	0.4006	0.2737	0.2143	0.1697	0.1449	0.1228	0.7136
U4 V B2 int	0.7956	0.3971	0.2784	0.2092	0.1689	0.1425	0.1202	0.1047
U4 V B0 short	0.4287	0.2148	0.1529	0.1132	0.0907	0.0756	0.0649	0.0697
U4 V B2 short	0.4283	0.2202	0.1518	0.1096	0.0906	0.0891	0.0649	0.9944
svmguide1								
U0 S B0 double	4.3137	2.1601	1.5384	1.1488	0.9194	0.7632	0.6532	0.5708
U0 S B2 double	4.3132	2.1594	1.4681	2.6689	0.9101	0.7609	0.6546	0.5703
U0 S B0 float	4.3186	2.1753	1.5400	1.1392	0.9198	0.7609	0.6531	0.5702
U0 S B2 float	4.3144	2.1593	1.5325	1.1575	0.9164	0.7635	0.6547	0.5703
U0 V B0 double	3.0256	1.5186	1.0153	0.8075	0.6497	0.5351	0.4622	0.4008
U0 V B2 double	3.0287	1.5391	1.0272	0.8173	0.6418	0.5411	0.4617	1.9294
U0 V B0 float	2.6708	1.3393	0.8966	0.7062	0.5628	1.4229	0.4066	0.3538
U0 V B2 float	2.6712	1.3408	0.9567	0.7120	0.5703	0.4732	0.4066	0.3539
U4 S B0 double	2.9879	1.4991	1.0539	0.7904	0.6322	0.5310	0.4542	0.8764
U4 S B2 double	2.9900	1.5008	1.0050	0.8012	0.6389	0.5354	0.4543	0.3961
U4 S B0 float	2.8293	1.4219	0.9497	0.7477	0.6005	0.5014	0.4287	0.3747
U4 S B2 float	2.8346	1.4189	0.9477	0.7207	0.6011	0.5035	0.4288	1.8885
U4 V B0 double	2.4867	1.2517	0.8780	0.6569	0.5247	0.4389	0.3778	0.3366
U4 V B2 double	2.4845	1.2458	0.8786	0.6676	0.5257	0.4385	0.3774	1.7619
U4 V B0 float	2.2638	1.1376	0.7576	0.5990	0.4803	0.4112	0.3452	0.3001
U4 V B2 float	2.2643	1.1492	0.7573	0.6067	0.4791	0.3996	0.3433	0.9136
svmguide3								
U0 S B0 double	3.3955	1.8016	1.1702	0.9805	0.8474	0.6597	0.5447	0.5629
U0 S B2 double	3.3813	1.7875	1.1782	0.9664	0.8070	0.6718	0.5467	0.5810
U0 S B0 float	3.2906	1.6927	1.1298	0.9381	0.7767	0.6032	0.5185	0.5205
U0 S B2 float	3.2906	1.6866	1.1298	0.9402	0.8171	0.6073	0.5185	0.5185
U0 V B0 double	1.9025	1.1601	0.8595	0.5367	0.4439	0.3531	0.3793	0.3107
U0 V B2 double	1.8622	1.1601	0.7646	0.7021	0.4519	0.4378	0.3107	0.3147
U0 V B0 float	1.1614	0.6010	0.4938	0.3884	0.3236	0.2256	55.1929	0.2256
U0 V B2 float	1.1633	0.5992	0.4531	0.3884	0.3162	0.2145	0.2219	0.2256
U4 S B0 double	2.1486	1.1056	0.7424	0.6819	0.5044	0.4136	0.3450	0.3430

Combination	1T	2T	3T	4T	5T	6T	7T	8T
U4 S B2 double	2.1305	1.1056	0.8635	0.6416	0.5064	0.4398	0.3813	0.3813
U4 S B0 float	2.0861	1.0733	0.8252	0.5952	43.7153	0.4015	0.3309	0.3329
U4 S B2 float	2.0881	1.0773	0.7202	0.6517	0.4903	80.0789	0.3329	82.1024
U4 V B0 double	1.2892	0.7465	0.5609	0.4459	0.3067	0.2482	0.2441	0.2522
U4 V B2 double	1.2690	0.6638	0.4983	0.3531	0.3067	0.2925	0.2502	0.2542
U4 V B0 float	0.8026	0.5012	0.3699	0.2312	0.1905	51.7013	0.1535	0.1683
U4 V B2 float	0.8008	0.4161	0.3310	0.2293	0.2145	0.1516	0.1757	0.1701
vowel								
U0 S B0 double	3.7147	1.8639	1.3140	1.0126	0.8401	0.6639	0.5694	0.5221
U0 S B2 double	3.7138	1.8639	1.3140	1.0117	0.7993	0.6639	0.5944	0.5146
U0 S B0 float	3.6526	1.8397	1.2296	0.9764	0.7975	10.2827	0.5592	46.3294
U0 S B2 float	3.6582	1.8397	1.2991	0.9764	0.7761	0.6537	0.5592	0.4942
U0 V B0 double	2.2682	1.3872	0.9801	0.7242	0.5703	0.4071	0.3542	0.3718
U0 V B2 double	2.2682	1.1461	0.8559	0.7409	0.5916	0.4071	0.3496	0.3709
U0 V B0 float	1.5254	0.7697	0.5170	0.4343	0.3307	0.2751	0.2357	0.2079
U0 V B2 float	1.5262	0.7697	0.5162	0.3871	0.3246	0.2720	0.2365	0.2079
U4 S B0 double	2.3878	1.3909	0.9505	0.6370	0.5221	0.4228	0.3681	0.3700
U4 S B2 double	2.3794	1.1981	0.9913	0.6352	0.5388	0.4266	0.3663	0.3746
U4 S B0 float	2.3136	1.1647	0.8216	0.6166	44.4303	0.4590	0.3616	0.3171
U4 S B2 float	2.3108	1.1647	0.7780	0.6408	0.5119	0.4154	0.3561	28.6839
U4 V B0 double	1.6116	0.8142	0.7001	0.5573	0.3839	45.0256	0.2513	0.2698
U4 V B2 double	1.6116	1.0441	0.5870	0.5462	0.3561	0.3672	0.3181	26.6949
U4 V B0 float	1.0795	0.5463	0.3848	0.2898	0.2303	0.1978	0.1685	0.1484
U4 V B2 float	1.0811	0.5486	0.3848	0.2898	0.2365	0.1978	0.1669	0.1476
w1a								
U0 S B0 byte	2.8484	1.4242	0.9457	0.7103	0.5989	0.4991	0.4292	0.3743
U0 S B2 byte	2.8335	1.4168	0.9438	0.7090	0.5998	0.5008	0.4275	0.3738
U0 S B0 double	3.2283	1.6421	1.0931	0.8235	0.6846	0.5768	0.4980	0.4283
U0 S B2 double	3.1853	1.5951	1.0628	0.8044	0.6799	0.5632	0.4843	0.4248
U0 S B0 float	2.9354	1.4719	0.9785	0.7342	0.6182	0.5151	0.4426	0.3864
U0 S B2 float	2.9032	1.4529	0.9698	0.7272	0.6138	0.5121	0.4403	0.3840
U0 S B0 int	2.8148	1.4074	0.9384	0.7068	0.5966	0.4972	0.4251	0.3709
U0 S B2 int	2.7836	1.3917	0.9335	0.6998	0.5913	0.4909	0.4218	0.3683
U0 S B0 short	3.0153	1.5092	1.0062	0.7547	0.6365	0.5304	0.4803	0.3978
U0 S B2 short	3.0041	1.5039	1.0028	0.7527	0.6344	0.5294	0.5356	0.3994
U0 V B0 byte	0.3225	0.1606	0.1071	0.0804	0.0680	0.0565	0.0489	0.0423
U0 V B2 byte	0.3145	0.1571	0.1049	0.0792	0.0664	0.0556	0.0477	0.0415
U0 V B0 double	1.7502	0.8899	0.5914	0.4592	0.3808	0.3212	0.2839	0.2314
U0 V B2 double	1.6333	0.8195	0.5463	0.4131	0.3522	0.2910	0.2502	0.2189
U0 V B0 float	0.8665	0.4315	0.2877	0.2162	0.1822	0.1519	0.1314	0.1141
U0 V B2 float	0.8324	0.4169	0.2802	0.2089	0.1761	0.1470	0.1270	0.1105
U0 V B0 int	1.1088	0.5548	0.3718	0.2789	0.2353	0.1962	0.1680	0.1464
U0 V B2 int	1.0827	0.5441	0.3636	0.2722	0.2306	0.1915	0.1645	0.1436
U0 V B0 short	0.4658	0.2315	0.1667	0.1159	0.0976	0.0818	0.0705	0.0610
U0 V B2 short	0.4485	0.2244	0.1503	0.1125	0.0948	0.0796	0.0682	0.0595
U4 S B0 byte	2.1262	1.0628	0.7097	0.5324	0.4486	0.3738	0.3217	0.2804
U4 S B2 byte	2.1320	1.0650	0.7114	0.5337	0.4500	0.3769	0.3225	0.2818
U4 S B0 double	2.2351	1.1042	0.8240	0.5573	0.4973	0.4121	0.3444	0.2917
U4 S B2 double	1.9663	0.9890	0.6595	0.4972	0.4176	0.3494	0.3011	0.2635
U4 S B0 float	1.9414	0.9707	0.6509	0.4881	0.4129	0.3418	0.2940	0.2565
U4 S B2 float	1.9054	0.9537	0.6383	0.4789	0.4030	0.3362	0.2888	0.2522
U4 S B0 int	1.9271	0.9636	0.6450	0.4841	0.4086	0.3407	0.2913	0.2541
U4 S B2 int	1.8927	0.9481	0.6352	0.4766	0.4007	0.3360	0.2875	0.2507
U4 S B0 short	2.2126	1.1072	0.7386	0.5541	0.4668	0.3891	0.3346	0.2918
U4 S B2 short	2.1978	1.1049	0.7332	0.5509	0.4643	0.3871	0.3330	0.2903
U4 V B0 byte	0.2571	0.1285	0.0863	0.0647	0.0544	0.0456	0.0391	0.0341
U4 V B2 byte	0.2508	0.1261	0.0836	0.0630	0.0530	0.0444	0.0379	0.0331
U4 V B0 double	1.2981	0.7185	0.4786	0.3611	0.2859	0.2614	0.2263	0.1954
U4 V B2 double	1.0113	0.5099	0.3398	0.2568	0.2157	0.1812	0.1554	0.1381
U4 V B0 float	0.5639	0.2823	0.1900	0.1423	0.1217	0.1017	0.0925	0.0808
U4 V B2 float	0.5182	0.2611	0.1740	0.1309	0.1106	0.0918	0.0790	0.0692

Combination	1T	2T	3T	4T	5T	6T	7T	8T
U4 V B0 int	0.7458	0.3728	0.2501	0.1885	0.1591	0.1318	0.1150	0.1002
U4 V B2 int	0.7130	0.3569	0.2391	0.1796	0.1511	0.1262	0.1083	0.0946
U4 V B0 short	0.3613	0.1806	0.1210	0.0914	0.0767	0.0639	0.0560	0.0487
U4 V B2 short	0.3421	0.1710	0.1142	0.0858	0.0722	0.0609	0.0517	0.0453
w8a								
U0 S B0 byte	2.8931	1.4458	0.9575	0.7218	0.6080	0.5058	0.4340	0.3789
U0 S B2 byte	2.8310	1.4234	0.9451	0.7092	0.5978	0.5008	0.4285	0.3738
U0 S B0 double	3.4925	1.7704	1.2979	0.9049	0.8546	0.7179	0.6555	0.5715
U0 S B2 double	3.1887	1.5955	1.0687	0.8047	0.6751	0.5646	0.4851	0.4284
U0 S B0 float	3.0629	1.5490	1.0248	0.7813	0.6599	0.5427	0.4661	0.4054
U0 S B2 float	2.9000	1.4593	0.9681	0.7318	0.6135	0.5119	0.4402	0.3840
U0 S B0 int	2.9380	1.5148	0.9905	0.7543	0.6387	0.5258	0.4484	0.3921
U0 S B2 int	2.7807	1.3915	0.9283	0.6971	0.5882	0.4909	0.4219	0.3684
U0 S B0 short	3.0883	1.5514	1.0319	0.7752	0.6576	0.5442	0.4669	0.4073
U0 S B2 short	3.0040	1.5026	1.0016	0.7527	0.6345	0.5328	0.5491	0.3968
U0 V B0 byte	0.3659	0.1948	0.1416	0.1070	0.0969	0.0785	0.0777	0.0645
U0 V B2 byte	0.3135	0.1570	0.1047	0.0786	0.0668	0.0556	0.0477	0.0415
U0 V B0 double	2.1104	1.5121	1.1944	0.8724	0.7613	0.6286	0.6139	0.5213
U0 V B2 double	1.6341	0.8185	0.5498	0.4121	0.3495	0.2914	0.2511	0.2205
U0 V B0 float	1.0699	0.7428	0.5415	0.4382	0.3698	0.3124	0.3051	0.2592
U0 V B2 float	0.8313	0.4163	0.2779	0.2099	0.1768	0.1477	0.1278	0.1111
U0 V B0 int	1.2937	0.7177	0.6054	0.4428	0.3991	0.3243	0.3164	0.2682
U0 V B2 int	1.0825	0.5417	0.3636	0.2728	0.2298	0.1929	0.1652	0.1440
U0 V B0 short	0.5676	0.3653	0.2969	0.2130	0.1932	0.1560	0.1543	0.1299
U0 V B2 short	0.4478	0.2242	0.1496	0.1124	0.0948	0.0791	0.0682	0.0595
U4 S B0 byte	2.2031	1.1085	0.7342	0.5490	0.4658	0.3864	0.3300	0.2898
U4 S B2 byte	2.1314	1.0660	0.7109	0.5354	0.4530	0.3748	0.3223	0.2813
U4 S B0 double	3.2865	1.8032	1.3094	1.1557	0.7338	0.7996	0.5980	0.6099
U4 S B2 double	1.9679	0.9856	0.6617	0.4988	0.4184	0.3511	0.3017	0.2654
U4 S B0 float	2.2996	1.2882	0.8870	0.7340	0.5250	0.5165	0.4141	0.3867
U4 S B2 float	1.9030	0.9526	0.6394	0.4798	0.4031	0.3364	0.2892	0.2527
U4 S B0 int	2.2794	1.2149	0.8489	0.7454	0.4959	0.5147	0.4288	0.3884
U4 S B2 int	1.8919	0.9470	0.6323	0.4752	0.4031	0.3365	0.2875	0.2510
U4 S B0 short	2.3879	1.2265	0.8128	0.6021	0.5169	0.4272	0.3658	0.3178
U4 S B2 short	2.1975	1.0990	0.7362	0.5525	0.4668	0.3871	0.3331	0.2903
U4 V B0 byte	0.3430	0.2053	0.1487	0.1077	0.0959	0.0772	0.0743	0.0618
U4 V B2 byte	0.2505	0.1253	0.0839	0.0632	0.0533	0.0444	0.0381	0.0332
U4 V B0 double	2.3501	1.4117	1.1575	0.7813	0.7258	0.5556	0.5759	0.4873
U4 V B2 double	1.0112	0.5075	0.3394	0.2609	0.2168	0.1822	0.1574	0.1398
U4 V B0 float	1.2274	0.7968	0.6111	0.3893	0.3766	0.2936	0.2934	0.2461
U4 V B2 float	0.5172	0.2596	0.1751	0.1305	0.1109	0.0929	0.0802	0.0695
U4 V B0 int	1.4563	0.8742	0.6868	0.5379	0.3685	0.3814	0.3122	0.2433
U4 V B2 int	0.7125	0.3584	0.2387	0.1795	0.1515	0.1266	0.1091	0.0953
U4 V B0 short	0.6834	0.4539	0.3038	0.2148	0.1926	0.1518	0.1493	0.1208
U4 V B2 short	0.3415	0.1710	0.1143	0.0859	0.0724	0.0604	0.0521	0.0454

Appendix C

Project resources

All the development and testing material developed for this work can be found on line at:

`http://code.ac.upc.edu/projects/nnvect`

In particular, result files and databases are accessible through this location:

`http://code.ac.upc.edu/projects/nnvect/wiki/ExternalFiles`

Bibliography

[1] José R. Herrero and Juan J. Navarro. Exploiting computer resources for fast nearest neighbor classification. *Pattern Analysis & Applications*, 10:265–275, 2007.

[2] T. M. Cover and P. E. Hart. Nearest neighbor pattern classification. *IEEE Transactions on Information Theory*, IT-13(1):21–7, January 1967.

[3] D. Michie, D. J. Spiegelhalter, and C. C. Taylor, editors. *Machine Learning, Neural and Statistical Classification*. Ellis Horwood, 1994.

[4] G. Shakhnarovich, P. Indyk, and T. Darrell, editors. *Nearest Neighbor Methods in Learning and Vision. Theory & Practice*. MIT Press, March 2006.

[5] Kevin Beyer, Jonathan Goldstein, Raghu Ramakrishnan, and Uri Shaft. When is "nearest neighbour" meaningful? In Catriel Beeri and Peter Buneman, editors, *Proc. 7th Int. Conf. Data Theory, ICDT*, number 1540 in Lecture Notes in Computer Science, LNCS, pages 217–235. Springer-Verlag, 10–12 January 1999.

[6] S. Mahamud and M. Hebert. Minimum risk distance measure for object recognition. In *International Conference on Computer Vision*, pages 242–248, 2003.

[7] E.Pekalska, R.P.W. Duin, S.Gunter, and H.Bunke. On not making dissimilarities euclidean. In *Joint IAPR International Workshops on Statistical and Structural Pattern Recognition*, pages 1143–1151, 2004.

[8] Vassilis Athitsos, Jonathan Alon, and Stan Sclaroff. Efficient nearest neighbor classification using a cascade of approximate similarity measures. In *IEEE Computer Vision and Pattern Recognition (CVPR)*, pages I: 486–493. IEEE Computer Society, 2005.

[9] V. Garcia, E. Debreuve, and M. Barlaud. Fast k nearest neighbor search using GPU. In *Computer Vision on GPU*, pages 1–6, 2008.

[10] Bryan Catanzaro, Narayanan Sundaram, and Kurt Keutzer. Fast support vector machine training and classification on graphics processors. In *ICML '08: Proceedings of the 25th international conference on Machine learning*, pages 104–111, New York, NY, USA, 2008. ACM.

[11] Indyk. Nearest-neighbor searching in high dimensions. In Jacob E. Goodman and Joseph O'Rourke, editors, *Handbook of Discrete and Computational Geometry*, volume 2. CRC Press, 2004.

[12] Piotr Indyk and Rajeev Motwani. Approximate nearest neighbors: Towards removing the curse of dimensionality. In *Proceedings of the 30th Annual ACM Symposium on Theory of Computing, STOC'98 (Dallas, Texas, May 23-26, 1998)*, pages 604–613, New York, 1998. ACM Press.

[13] Kevin J. Barker, Kei Davis, Adolfy Hoisie, Darren J. Kerbyson, Michael Lang 0003, Scott Pakin, and José Carlos Sancho. A performance evaluation of the nehalem quad-core processor for scientific computing. *Parallel Processing Letters*, 18(4):453–469, 2008.

[14] Vasily Volkov and James W. Demmel. Benchmarking GPUs to tune dense linear algebra. In *SC '08: Proceedings of the 2008 ACM/IEEE conference on Supercomputing*, pages 1–11, Piscataway, NJ, USA, 2008. IEEE Press.

[15] Godfried T. Toussaint. Geometric proximity graphs for improving nearest neighbor methods in instance-based learning and data mining. *Int. J. Comput. Geometry Appl*, 15(2):101–150, 2005.

[16] Nello Cristianini and John Shawe-Taylor. *An Introduction to Support Vector Machines (and Other Kernel-Based Learning Methods).* Cambridge University Press, Cambridge, United Kingdom, 2000.

[17] Koby Crammer and Yoram Singer. On the algorithmic implementation of multiclass kernel-based vector machines. *Journal of Machine Learning Research*, 2:265–292, 2001.

[18] John Shawe-Taylor and Nello Cristianini. *Kernel methods for pattern analysis.* Cambridge University Press, 2004.

[19] Rong-En Fan, Kai-Wei Chang, Cho-Jui Hsieh, Xiang-Rui Wang, and Chih-Jen Lin. Liblinear: A library for large linear classification. *J. Mach. Learn. Res.*, 9:1871–1874, 2008.

[20] Cho-Jui Hsieh, Kai-Wei Chang, Chih-Jen Lin, S. Sathiya Keerthi, and S. Sundararajan. A dual coordinate descent method for large-scale linear SVM. In William W. Cohen, Andrew McCallum, and Sam T. Roweis, editors, *Machine Learning, Proceedings of the Twenty-Fifth*

International Conference (ICML 2008), Helsinki, Finland, June 5-9, 2008, volume 307 of *ACM International Conference Proceeding Series*, pages 408–415. ACM, 2008.

[21] S. Sathiya Keerthi, S. Sundararajan, Kai-Wei Chang, Cho-Jui Hsieh, and Chih-Jen Lin. A sequential dual method for large scale multi-class linear svms. In Ying Li, Bing Liu, and Sunita Sarawagi, editors, *Proceedings of the 14th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining, Las Vegas, Nevada, USA, August 24-27, 2008*, pages 408–416. ACM, 2008.

[22] Chih chung Chang and Chih jen Lin. LIBSVM: a library for support vector machines (version 2.31), September 07 2001.

[23] Oren Boiman, Eli Shechtman, and Michal Irani. In defense of nearest-neighbor based image classification. In *CVPR*. IEEE Computer Society, 2008.