A Hardware Accelerator for k-th Nearest Neighbor Thinning

Tobias Schumacher, Robert Meiche, Paul Kaufmann, Enno Lübbers, Christian Plessl and Marco Platzner
Paderborn Center for Parallel Computing, University of Paderborn
Fuerstenallee 11, 33102 Paderborn, Germany
Email: tobe@uni-paderborn.de

***Abstract*—This paper presents an accelerator for k-th nearest neighbor thinning, a run time intensive algorithmic kernel used in recent multi-objective optimizers. We discuss the thinning algorithm and the accelerator architecture with its modules and operation, and evaluate the accelerator with respect to two different application scenarios. The first is an embedded computing scenario where the accelerator core is part of a configurable system-on-chip implemented on a modern platform FPGA. We show the resource requirements for different instances of the accelerator and report on the raw speedups achieved, which are up to 358x. The second scenario is in high performance computing where we map the accelerator core to a cutting-edge reconfigurable computer, the XD1000 system, and achieve overall speedups of up to 6.6x compared to a software reference.**

***Index Terms*—k-th nearest neighbor thinning, reconfigurable accelerator, FPGA, SPEA2.**

I. Introduction

k-th-nearest-neighbor (KNN) methods are omnipresent in many areas of science and engineering. In statistics and data analysis, for example, KNN techniques play an important role for the non-parametric estimation of density functions from data samples [1]. Given a set of n data samples, where each sample i is a d-dimensional vector, an Euclidean distance metric σ_i is computed for any pair of samples. For each data sample, the resulting n distance values are sorted in ascending order, i.e., $\sigma_i^1 \leq \sigma_i^2, \ldots, \leq \sigma_i^n$. A KNN density estimate $\hat{f}(i)$ can then be formulated by setting:

$$\hat{f}(i) \propto \frac{1}{\sigma_i^k} \tag{1}$$

The parameter k is typically chosen as $k \approx \sqrt{n}$ [2]. Thus, the local density around each data sample i is estimated by the reciprocal of the distance to the k-th nearest neighbor. In other words, a low density means that a d-dimensional sphere with data sample i at its origin has to be rather large in order to contain k data samples.

The KNN approach is also widely applied for solving classifications problems, for example in machine learning, data mining and stochastic optimization [3]. There, a KNN classifier requires a labelled training data set consisting of d-dimensional feature vectors and their class labels. In order to classify a new feature vector, the k nearest training vectors are determined according to some distance metric. A 1-NN classifier will then assign the class of the closest training vector to the new vector. For $k > 1$, typically a voting scheme is applied and the new vector receives the class of the majority votes. There exists a great variety of KNN classifiers [4], differing in methods to break ties, in the efficiency of the used query data structures, and in the techniques applied to reduce the set of training data samples.

The reduction of the size of data samples is often desired to reduce both the classification time and the memory required to store the data set. Many reduction techniques fall into the category of condensing or thinning approaches [5], that aim at properly selecting a subset of training vectors from the original data set. Well-known thinning approaches are the condensed nearest neighbor algorithm, the reduced nearest neighbor algorithm, Baram's method, and proximity graph based thinning (see, e.g., [4]).

Interestingly, KNN-based thinning algorithms also find use in recent multi-objective evolutionary optimizers. Such optimizers are basically population-based, stochastic search algorithms inspired by principles from evolution theory. They try to solve a problem by keeping a set (population) of candidate solutions (individuals) in parallel and improving the quality (fitness) of the individuals over a number of iterations (generations). To form a new generation, genetically-inspired operators such as crossover and mutation are applied to the individuals. A fitness-based selection process steers the population towards better candidates. When optimizing for several, typically conflicting, objectives, the goal is to find reasonable trade-offs between the different objectives. The definition of a reasonable trade-off can be based on the concept of Pareto dominance. A solution vector i dominates another solution vector j when i is superior or equal to j in all objectives, and superior in at least one objective. A non-dominated solution is denoted as a Pareto point. As evolutionary algorithms are population-based methods, they are well-suited to approximate the set of Pareto points (Pareto front) in a single optimization run.

In particular, one of the most popular multi-objective optimizers, SPEA2 [6], relies on KNN methods for determining the fitness of solutions (using a KNN density estimate similar to Equation 1) and also for thinning out the approximated Pareto fronts. The latter becomes necessary when the algorithm generates more non-dominated solutions than can be stored in the fixed-size archive, which is rather likely for higher-dimensional problems. Depending on the actual problem under optimization, this KNN thinning technique

Published in: *Proc. Int. Conf. on Engineering of Reconfigurable Systems and Algorithms (ERSA)*

easily takes the vast majority of the optimizer's run time.

The acceleration of KNN-based thinning has not yet been studied in related work. However, related methods have been successfully accelerated with FPGAs. For example, Yeh et al. present a KNN classifier [7] that operates in the wavelet domain and uses partial distance search to accelerate the classification process. The resulting architecture is integrated as a core to the Altera NIOS CPU softcore. In [8] Chikhi et al. present an FPGA accelerated KNN classifier for content-based image retrieval that achieves a speedup of 45x over a software implementation. Also the related k-means clustering method has been successfully accelerated in reconfigurable hardware. For example, Saegusa and Maruyama have presented an architecture [9] that can perform k-means clustering on video data in realtime.

The contribution of the work presented in this paper is the development of a modular and portable hardware accelerator for KNN-based thinning. While the accelerator core aims at speeding up multi-objective evolutionary optimizers such as SPEA2, the main modules of the core are exactly the same as needed for other KNN-based methods, widening the applicability of this work.

The paper is organized as follows: Section II formally introduces the KNN-based thinning method that is accelerated in this work. Section III presents the parametrized architecture of the KNN core. In Section IV we evaluate the raw performance and the application-level performance of the KNN accelerator for two scenarios. Finally, we present conclusions and outline future work in Section V.

II. k-TH NEAREST NEIGHBOR THINNING

In this section, we detail the KNN-based thinning algorithm for reducing a set of vectors in a multi-dimensional feature space to a smaller set of representative vectors. The procedure, shown in Algorithm 1, is called with a set $\mathcal{P}$ of different d-dimensional vectors $p_i = (p_{i1}, p_{i2}, \ldots, p_{id})$ and N, the targeted cardinality of $\mathcal{P}$, and successively eliminates vectors with the shortest Euclidean distance to their neighbors until $\mathcal{P}$ has been reduced to N elements. In each iteration, the algorithm first constructs a distance matrix $\underline{\sigma} = (\sigma_{il})$ from the pair-wise Euclidean distances σ_{il} between all vectors. Sorting $\underline{\sigma}$ row-wise in ascending order defines sorted distance vectors $\sigma_{\mathbf{i}}{}'$ of length m. While initially equal to $|\mathcal{P}|$, the number of vectors m is reduced by one in each iteration. Now, the algorithm iterates over all columns of $\underline{\sigma}'$. Starting with column $l = 3$, the rows σ'_i with minimum distance values σ'_{il} among all distances in column l are selected and assigned to set $\mathcal{M}$. If the minimum is unique, the respective row $\sigma'_i \in \underline{\sigma}'$ as well as the corresponding vector p_i are deleted which reduces the set of vectors $\mathcal{P}$. If the minimum is not unique, the next column of $\underline{\sigma}'$ is considered which corresponds to checking the distances to the next closest neighbors. If no unique minimal distance is found for all columns of $\underline{\sigma}'$, an arbitrary row having a minimal distance value in the last column is deleted. Obviously, the first two columns never need to be considered since each vector has a distance of 0 to itself (first column) and the distances between pairs of vectors are symmetric (second column).

Algorithm 1 KNN thinning algorithm

```
1:  procedure KNN_THINNING(P, N)
2:      while |P| > N do
3:          compute/update σ_il ← sqrt(Σ_{j=1}^{d} (p_ij − p_lj)^2)
4:          ∀ rows of σ : σ_i' ← sort(σ_i)
5:          for l ← 3, ..., m do
6:              M ← {σ'_i | ∀σ'_jl : σ'_il ≤ σ'_jl}
7:              if |M| == 1 then
8:                  break
9:              end if
10:         end for
11:         delete arbitrary row σ_i ∈ M from σ'
12:         P ← P \ {p_i}
13:     end while
14: end procedure
```

The operation of the KNN-based thinning algorithm is illustrated in the example shown in Figure 1. The initial population of six 2-dimensional vectors as well as the three vectors that are discarded by three iterations of the thinning algorithm are shown in Figure 1(a). The distance matrix $\underline{\sigma}$ for the first iteration is presented in Figure 1(b), and the row-wise sorted distance matrix $\underline{\sigma}'$ in Figure 1(c). Note that the matrices actually show the square distances which is sufficient for this algorithm. A unique minimum is found in the third column which leads to the deletion of row b from the matrix and vector b from the population, respectively. In the second iteration, the distance matrix is updated and re-sorted which results in the matrix of Figure 1(d). Again, a unique minimum is identified in the third column and, consequently, row e and vector e are deleted. Finally, the third iteration leads to the deletion of vector c.

III. ACCELERATOR ARCHITECTURE

This section presents the architecture of the KNN-based thinning accelerator, including the main datapath modules and the system interface.

A. Accelerator Modules

The accelerator is composed of four main datapath modules that together implement the different tasks of the thinning procedure shown in Algorithm 1. These modules are:

- the *vector table*
- the *distance calculator*
- the *distance sorter*
- the *row selector*

Figure 2 displays the block diagram for the accelerator's datapath. A controller module is responsible for handling the commands from the bus interface and for sequencing the execution of the datapath modules. For simplicity, control lines have been omitted in Figure 2. The datapath modules perform following the tasks:

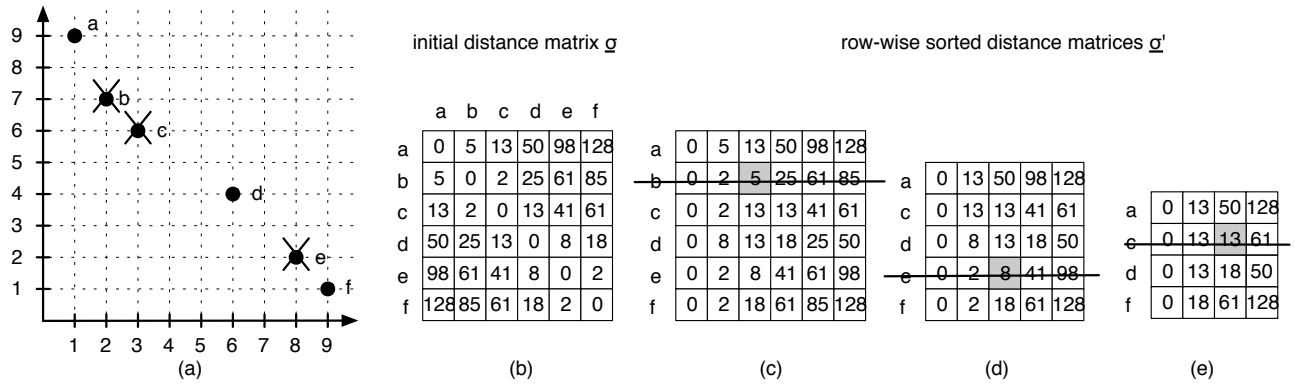

Figure 1. Example for the KNN-based thinning algorithm in two dimensions: thinning 6 vectors to 3 vectors

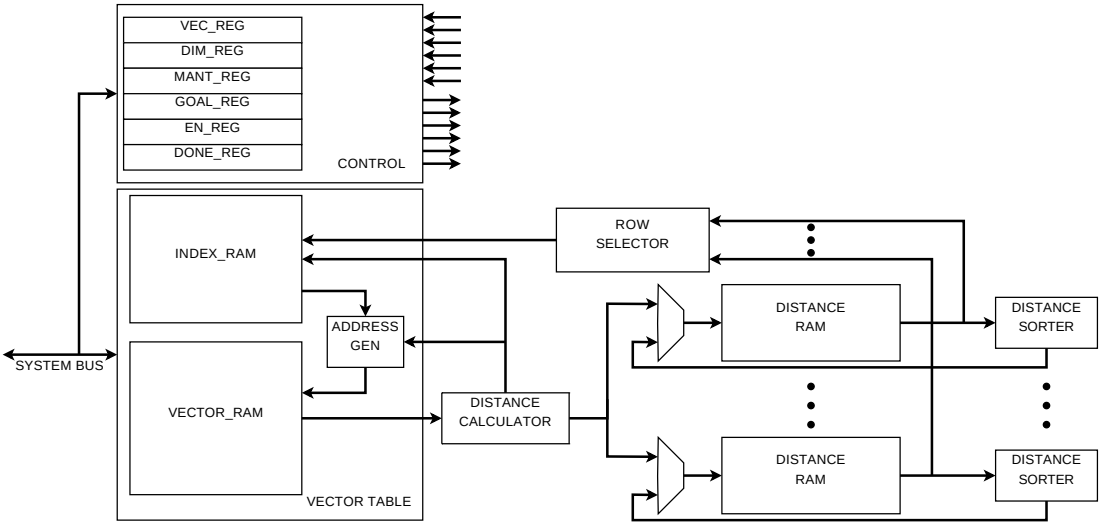

Figure 2. Datapath of the KNN-based thinning accelerator

a) Vector table: The vector table module stores and manages the data vectors during the thinning process. The module mainly consists of two RAM blocks: the vector RAM that stores the input vectors and the index RAM which stores the indices of active vectors located in the vector RAM. Initially, the vectors are written into the vector RAM and the index RAM is initialized with entries running from 0 to $|\mathcal{P}|-1$. The current number of active vectors is stored in the register `VEC_REG` within the controller module, and is also initialized to $|\mathcal{P}| - 1$.

During the thinning process, vectors are successively being removed from the matrix $\underline{\sigma}'$. This step is implemented in the vector table module by marking the vector to be deleted as inactive. If vector i is to be deleted, the index RAM is compacted by copying the last active element of the index RAM (pointed to by `VEC_REG`) to position i of the index RAM. Hence, at any time the contents of the first `VEC_REG` entries in the index RAM point to the remaining active vectors in the vector RAM.

The data in the vector RAM is organized vector-by-vector, each vector in ascending dimensions. The address of a specific data element p_{im} in the vector RAM is thus easily determined by the address generator as address$(p_{im}) = \text{INDEX_RAM}[i] \cdot d + m$, where d is the number of dimensions.

b) Distance calculator: This module computes $\underline{\sigma}$, the pair-wise distances between all active vectors. The vector table's address generator is used to fetch the components for each pair of vectors from the vector RAM. The resulting distances are written to the distance RAM blocks. Although the thinning algorithm uses the Euclidean distance metric, we omit the square root operation in our accelerator. This has no impact on the result as the algorithm applies only comparison operations to the distance values. The distance calculator module exploits the pair-wise symmetry of the distance values and writes a distance result σ_{ij} not only to the i-th row in $\underline{\sigma}$ but concurrently also the j-th row as σ_{ji}. As soon as all distances for one vector are calculated, the distance sorter module associated to the respective distance RAM is started by the controller.

c) Distance sorter: The distance sorter modules sort the distance values in the distance RAM blocks. To speed up the sorting step the architecture employs a dedicated sorter

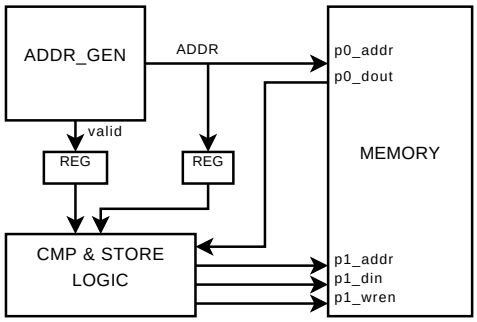

Figure 3. Architecture of the distance sorter

for each distance RAM block, i.e., all rows σ_i are sorted in parallel. Our current implementation relies on Bubblesort for sorting.

Algorithm 2 Pseudocode executed by the sorter module

```
1:  procedure SORT_VECTOR(n)
2:      for i ← 0, ..., n − 2 do
3:          cur_max := getData(i)
4:          for j ← i + 1, ..., n − 1 do
5:              cur_val := getData(j)
6:              if cur_val < cur max then
7:                  storeData(cur_val, j − 1)
8:              else
9:                  storeData(cur_max, j − 1)
10:                 cur_max := cur_val
11:             end if
12:         end for
13:         storeData(cur_max, n − 1)
14:     end for
15: end procedure
```

Figure 3 shows a block diagram of the sorter module, the functionality of the module is outlined in Algorithm 2. The address generator roughly implements both of the for-loops, e. g., it generates i and j in the algorithm and provides these values as address to the first port of the memory module. The registered address is also forwarded to the compare module. The compare module stores the first value read from the memory into the cur_max register. Subsequent values read are compared to this registered value, the smaller value is stored to the registered address and the bigger one is again registered as cur_max. As the last step of each iteration, the value of the cur_max register is stored to the memory.

Using a faster sorter architecture, e. g., a bitonic sorter [10], could speedup the sorting of a single row. However, a more sophisticated sorter requires also significantly more hardware resources for larger distance vectors.

d) Row selector: As soon as all distance RAMs have been sorted, the row selector module is activated by the controller. The row selector sequentially searches all distance RAMs for a column with a unique minimal element according to Algorithm 1. When such an element is found, the vector corresponding to the row with the minimal element is scheduled for deletion. In case no column contains a unique minimal element, the row selector schedules the first row for deletion that contains one of the minimal elements of the last column. The index of the selected row (vector) is passed to the vector table module where the index RAM is updated.

B. Interface and Operation

The accelerator core exposes an easy-to-use interface consisting of six control registers and two read and writable memories, the vector RAM and the index RAM. These blocks are mapped into the address space of the system bus.

To prepare the operation of the core, all input vectors are loaded to the vector RAM and the index RAM is initialized with values from 0 to $|\mathcal{P}| - 1$. After the parameters of the KNN-based thinning procedure such as the number of vectors ($|\mathcal{P}|$), the dimension (d) and the thinning goal (N) have been written to the control registers of the core, the execution is enabled by triggering an enable register. The core announces the end of the operation by setting a flag in the done register, that can be polled or used for triggering an interrupt on the host CPU. After the $|\mathcal{P}| - N$ elimination steps, the host CPU can read the indices of the vectors that have not been eliminated from the index RAM.

The memory mapped core interface makes porting the accelerator to different architectures a rather straight-forward task. Originally, the accelerator core was targeted to be attached to the OPB bus of a PPC or Microblaze CPU core in Xilinx Virtex-II Pro FPGA technology. Later, we have ported the core to an Altera Stratix-II FPGA in an XtremeData XD1000 system [11]. Adapting the core to use the HyperTransport communication interface provided by XtremeData's reference design was a task of a few hours.

We have implemented the KNN accelerator core in VHDL. The design is parametrized in the number of vectors and dimensions through VHDL generics. These parameters, however, are maximum values. That is, a core synthesized for a certain number of vectors can also cope with less vectors. In contrast to an accelerator core designed to be able to handle any number of vectors and dimensions, our core benefits from the higher level of customization and will thus deliver higher performance. The restriction to a maximum number of vectors and dimension is not considered a limitation, as for a given reconfigurable computer one can easily pre-synthesize several accelerator instances differing in the number of vectors and dimensions that fully utilize the available FPGA resources. Since the problem size is known for applications using a KNN-based thinning accelerator, the appropriate accelerator core can be configured before the applications starts.

IV. Evaluation

In this section we evaluate our KNN-based thinning accelerator with two sets of experiments. In the first set of experiments, we measure the raw performance of the accelerator for an embedded computing scenario, where the core is part of a configurable system-on-chip architecture

#vectors #dimensions	64 2/4	128 2/4	256 2/4	capacity (2VP70)
registers	6%	13%	28%	66176
slices	30%	57%	115%	33088
BRAMs	19%	39%	78%	328
multipliers	1%	1%	1%	328
speed [MHz]	79	78	74	–

Table I
HARDWARE RESOURCE REQUIREMENTS FOR DIFFERENT KNN-BASED THINNING ACCELERATORS ON VIRTEX-II PRO

implemented in a Virtex-II Pro device. The second set of experiments applies the KNN-based thinning accelerator to a high-performance computing scenario where the core is mapped to an XtremeData XD1000 system for speeding up the Pareto front thinning process of the SPEA2 evolutionary multi-objective optimizer [6].

A. Raw Speedup

In this experiment we target an AlphaData ADM-XP board providing a Virtex-II Pro 2VP70-5 FPGA. The required hardware resources for the accelerator depend on the maximum number of vectors and the dimension. We can fit accelerator cores for up to 128 vectors with 4 dimensions to this FPGA. The limiting factor for implementing cores with more vectors is the availability of logic resources. For reference, we have also added synthesis results for a core instance with 256 vectors. Table I shows the resource requirements and the speed as reported by Xilinx ISE 9.2i for six instances of the core. The resources account for the accelerator core only and exclude the OPB bus interface. The figures for 64 and 128 vectors are post place-and-route results, results for 256 vectors are post synthesis results reported by Xilinx XST. An interesting observation is that the resource requirements hardly depend on the number of dimensions. In fact, the increase in resources for a core supporting 4 instead of 2 dimensions is less then 0.1%. Thus we have summarized the results from the 2 and 4-dimensional versions in a single column in Table I. While the logic resource requirements for cores with higher dimensions are slightly increased, the BRAM requirements remain the same. This behavior is determined by the core's architecture, where only the vector RAM increases linearly with higher dimensional vectors while the distance and index RAMs are completely independent of the dimension. In contrary, the logic and BRAM requirements grow linearly with the number of vectors that are supported because a distance RAM and a sorter module is instantiated for every vector.

We have performed benchmarks for determining the raw speedup of the accelerator core compared to the PPC processor. To that end, the KNN-based thinning core has been attached to the PowerPC CPU core of the Virtex-II Pro as an OPB slave peripheral running at 40 MHz. The used software reference implementation of the KNN-based thinning procedure is based on SPEA2's implementation. The procedure has been rewritten to operate with fixed-point arithmetic only, and to avoid dynamic memory allocation. These changes are realistic for embedded systems, where typically no floating-point units are available and dynamic memory allocation is avoided if possible. It has to be noted that in each iteration of the thinning algorithm, the accelerator sorts the distance matrix $\underline{\sigma}$, while in the software implementation re-sorting is avoided by compacting the corresponding rows of the matrix. Consequently, the asymptotic run time complexity of the accelerator is higher than that of the software version. The software reference has been executed on the PPC core of the Virtex-II Pro running at 240 MHz.

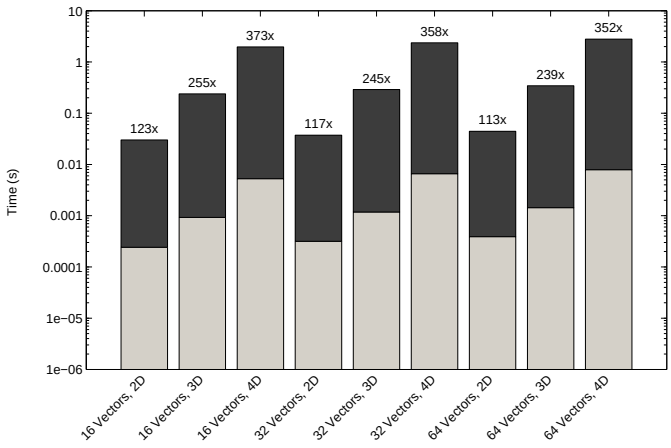

Figure 4. Performance comparison between the hardware and software implementation of the KNN-based thinning accelerator for different numbers of vectors and dimensions

We have evaluated 9 different accelerator architectures that are characterized by the number of vectors $|\mathcal{P}| = \{16, 32, 64\}$ and the number of dimensions $d = \{2, 3, 4\}$. For each architecture we have ran 500 experiments in which $|\mathcal{P}|$ randomly generated input vectors with d dimensions have been thinned to $|\mathcal{P}|/4$ vectors. The random input vectors have been generated by the PPC core using the `rand()` function of Xilinx runtime library. Figure 4 shows the results of this comparison. For each architecture the lower bar shows the mean of the execution times in hardware, and the upper bar shows the mean of the execution times in software. On top of each bar, the speedup of the hardware execution over the software execution is annotated. Figure 4 shows that the accelerator achieves significant speedups of at least two orders of magnitude over the embedded software implementation. We also observe that for a given number of vectors the speedup increases approximately linearly with the number of dimensions.

B. Accelerating the SPEA2 evolutionary algorithm

To study the performance of the KNN-based thinning accelerator in a high performance computing scenario, we have integrated the accelerator core into the SPEA2 multi-objective evolutionary optimizer [6]. As explained in Section I, SPEA2 preserves a fixed-size archive of non-dominated solutions. When the number of non-dominated solutions created in an iteration of the algorithm exceeds the size of the archive,

resource	usage				capacity
	complete		KNN core		(EP2S180)
ALUTs	26846	19%	24513	17%	143520
Registers	17694	12%	9535	7%	143520
DSPs	8	1%	8	1%	768
Mem bits	1549728	17%	1083136	12%	9383040
M512s	38	4%	0	0%	930
M4Ks	289	38%	266	35%	768
M-RAMs	1	11%	0	0%	9

Table II
HARDWARE RESOURCE REQUIREMENTS FOR A 128 VECTORS 4D KNN-BASED THINNING ACCELERATOR ON STRATIX-II FOR THE COMPLETE DESIGN INCLUDING THE HYPERTRANSPORT INTERFACE AND FOR THE KNN CORE ALONE

KNN-based thinning is used for reducing the set of solutions. Depending on the problem that is optimized with SPEA2, the KNN-based thinning process can consume a significant share of the run time of SPEA2.

For this set of experiments, we have used a multi-dimensional Knapsack problem [12], which is a synthetic application commonly used for comparing evolutionary algorithms. We have taken up on the PISA framework [13] for integrating the Knapsack application with the SPEA2 optimizer, as the source code for this application is available from the PISA website [14].

As target architecture, we have used an XtremeData XD1000 system [11], which is a high-performance reconfigurable workstation. The XD1000 system is based on a dual-processor capable AMD Opteron mainboard, where one CPU socket is populated with an FPGA module. The FPGA module implements the HyperTransport protocol which allows for high-bandwidth and low latency communication with the 2.2 GHz Opteron CPU in the second CPU socket. The FPGA module mounts a high-density Altera Stratix-II EP2S180 FPGA, provides 4 MB local ZBT SRAM as well as an interface to the DDR-400 SDRAM installed on the mainboard. The control registers of our accelerator are made available to the Opteron CPU as memory mapped registers. Also the vector and index RAMs are directly mapped into the address space of the CPU. The KNN-based thinning core is running on the Altera FPGA at 100 MHz. Table II shows the post place-and-route resource usage as reported by Altera Quartus Version 7.2 for a core supporting a maximum of 128 vectors and four dimensions as well as for the complete design including the HyperTransport core and clock management logic.

We compare the execution times of a pure software implementation for solving the multi-dimensional Knapsack problem with SPEA2 with a hardware accelerated version of the same algorithm. The software version uses the KNN-based thinning procedure from an unmodified SPEA2 implementation, which is reasonably optimized and exploits efficient data structures and algorithms. For the hardware accelerated version, we have replaced the KNN-based thinning routine in SPEA2 with the call to our accelerator core. The code has been compiled with GCC 4.1.2., optimization level -O3. The execution times have been measured with the Unix time command.

#vectors	128		128		128	
#dimensions	2		3		4	
	SW	HW	SW	HW	SW	HW
time [s]	15.0	4.9	32.8	5.0	20.3	5
speedup		3.1x		6.6x		4.1x

Table III
RESULTS FROM SOLVING THE MULTI-DIMENSIONAL KNAPSACK PROBLEM USING SPEA2 ON THE XD1000

For computing the resulting speedups, we compare the execution time of the software-only implementation running on the Opteron CPU of the XD1000 system to the execution time of the hardware accelerated version, where the host code runs on the Opteron CPU and the accelerator runs on the HyperTransport-attached FPGA board. We take a conservative approach. For the pure software version, we are considering the user time consumed by the process only. For the hardware accelerated version, we are adding user and system time to also account for the data transfers between CPU and FPGA module.

For each instance of the architecture, we have executed 10 runs with different random seeds. In each run, SPEA2 optimizes the Knapsack problem over 5000 iterations with an archive size of 16 solutions. In each iteration, 128 individuals are generated, i. e., in the worst case, each iteration creates 128 non-dominated solutions that must be thinned down to 16 solutions. Since we use a multi-dimensional Knapsack problem, the number of dimensions can be easily scaled by increasing the number of knapsacks.

Table III summarizes the results from the benchmarks. The measured overall speedups range from 3.1x to 6.6x. In contrast to the case-study in Sec. IV-A which showed the raw-speedup of the accelerator architecture itself, the speedups reported in Table III are application-level speedups for solving the Knapsack optimization problem with the SPEA2 optimizer.

There main causes for the difference in raw-speedup and application-level speedup are:

- The gap between the clock speed of the CPU and the clock speed of the FPGA accelerator is increased from 6x in the embedded computing scenario to 22x in the HPC scenario.
- The CPU and memory architecture of the XD1000 system is significantly more advanced as the architecture of the embedded PPC core.

Still, the results show, that the KNN accelerator can significantly reduce the execution time for optimization solved with the SPEA2 evolutionary stochastic optimizer. The amount of acceleration that can be obtained is however problem dependent.

V. CONCLUSIONS AND FUTURE WORK

In this paper, we have presented an accelerator core for a KNN-based thinning algorithm. We have detailed the algorithm and the core's architecture, and reported on the result of two sets of experiments. The first set of experiments

has addressed an embedded computing scenario, where the accelerator core is part of a larger system-on-chip mapped to a modern platform FPGA. For this scenario, we have demonstrated that our accelerator can achieve very high raw speedups which, in part, has to be attributed to the relatively slow CPU. However, such an embedded system setup is realistic in scenarios where multi-objective optimizers such as SPEA2 have to be run on resource-limited systems. This is in line with recent research that has promoted system-on-chip versions of multi-objective optimizers to create self-optimizing autonomous systems [15][16]. Such approaches will strongly benefit from the accelerator presented in this paper.

The second set of experiments has targeted a high performance computing scenario, where the accelerator core has been mapped to a cutting-edge reconfigurable workstation. We have shown that our core is portable and has the potential to significantly accelerate SPEA2, even when compared to a software reference running on a state-of-the-art CPU. Although the accelerator core on the XD1000 system is clocked more than twice as fast as on the embedded system, the resulting speedups are lower which is mainly due to the much faster clock speed of the CPU. Multi-objective evolutionary algorithms are rather general optimization tools that are applied to many hard problems, especially in science and engineering, for example in automated design-space exploration [17]. Since such optimization problems often show very long run times, an accelerator core delivering a speedup of one order of magnitude with a standard workstation setup such as the XD1000 system is significant.

As future work, we will study how to improve the architecture such that the distance RAMs need to be sorted only once. This improvement will enable our architecture to match the asymptotic complexity of the software implementation. Also, we are planning to port the KNN-based thinning accelerator to our recently developed IMORC core interconnection infrastructure [18]. This will allow us to collect execution statistics of the core that can be used for performance tuning.

VI. Acknowledgements

This work was supported by the German Research Foundation under project number PL 471/1-2 within the priority program *Organic Computing* and by the Altera-AMD-Sun-XtremeData university program.

References

[1] D. Loftsgarden and C. Quesenberry, "A Nonparametric Estimate of a Multivariate Density Function," *The Annals of Mathematical Statistics*, vol. 31, pp. 1049–1051, 1965.

[2] B. W. Silverman, *Density Estimation for Statistics and Data Analysis*. Chapman & Hall/CRC, April 1986.

[3] T. Cover and P. Hart, "Nearest Neighbor Pattern Classification," *IEEE Transactions on Information Theory*, vol. 13, pp. 21–27, 1967.

[4] F. Bajramovic, F. Mattern, N. Butko, and J. Denzler, "A Comparison of Nearest Neighbor Search Algorithms for Generic Object Recognition," in *Proceedings of Advanced Concepts for Intelligent Vision Systems (ACIVS)*. Springer, 2006, pp. 1186–1197.

[5] J. Sanchez, J. Sotoca, and F. Pla, "Efficient Nearest Neighbor Classification with Data Reduction and Fast Search Algorithms," in *Proceedings of the IEEE International Conference on Systems, Man and Cybernetis*. IEEE, 204, pp. 4757–4762.

[6] E. Zitzler, M. Laumanns, and L. Thiele, "SPEA2: Improving the strength pareto evolutionary algorithm for multiobjective optimization," in *Evolutionary Methods for Design, Optimisation and Control with Application to Industrial Problems (EUROGEN 2001)*. International Center for Numerical Methods in Engineering (CIMNE), 2002, pp. 95–100.

[7] Y.-J. Yeh, H.-Y. Li, W.-J. Hwang, and C.-Y. Fang, "FPGA implementation of kNN classifier based on wavelet transform and partial distance search," in *Proc. Scandinavian Conf. on Image Analysis (SCIA)*, ser. LNCS, no. 4522. Springer-Verlag, 2007, pp. 512–521.

[8] R. Chikhi, S. Derrien, A. Noumsi, and P. Quinton, "Combining flash memory and FPGAs to efficiently implement a massively parallel algorithm for content-based image retrival," in *Proc. Int. Workshop on Reconfigurable Computing: Architectures, Tools and Applications*, ser. LNCS, no. 4419, 2007, pp. 247–258.

[9] T. Saegusa and T. Maruyama, "An FPGA implementation of k-means clustering for color images based on Kd-tree," *Proc. Int. Conf. on Field Programmable Logic and Applications (FPL)*, pp. 1–6, August 2006.

[10] K. Claessen, M. Sheeran, and S. Singh, "The design and verification of a sorter core," in *Correct Hardware Design and Verification Methods*, ser. Lecture Notes in Computer Science, no. 2411. Springer-Verlag, 2001, pp. 355–368.

[11] *XD1000 Development System*, XtremeData, Inc., Schaumburg, IL, USA, 2008. [Online]. Available: www.xtremedatainc.com

[12] E. Zitzler and L. Thiele, "Multiobjective evolutionary algorithms: A comparative case study and the strength pareto approach," *IEEE Trans. on Evolutionary Computation*, vol. 3, no. 4, pp. 257–271, 1999.

[13] S. Bleuler, M. Laumanns, L. Thiele, and E. Zitzler, "PISA — a platform and programming language independent interface for search algorithms," in *Evolutionary Multi-Criterion Optimization (EMO)*, ser. LNCS. Berlin: Springer-Verlag, 2003, pp. 494–508.

[14] [Online]. Available: http://www.tik.ee.ethz.ch/sop/pisa/

[15] P. Kaufmann and M. Platzner, "Toward Self-adaptive Embedded Systems: Multiobjective Hardware Evolution," in *Proceedings of the 20th International Conference on Architecture of Computing Systems (ARCS)*, ser. LNCS, vol. 4415. Springer, 2007.

[16] ——, "Multi-objective Intrinsic Hardware Evolution," in *Proceedings of the MAPLD Internional Conference*, 2006.

[17] M. Palesi and T. Givargis, "Multi-objective design space exploration using genetic algorithms," in *Proc. Int. Symp. on Hardware/Software Codesign (CODES)*. New York, NY, USA: ACM Press, 2002, pp. 67–72.

[18] T. Schumacher, C. Plessl, and M. Platzner, "IMORC: an infrastructure for performance monitoring and optimization of reconfigurable computers," Many-core and Reconfigurable Supercomputing Conference (MRSC) (poster), April 2008.