AN ACCELERATOR FOR K-TH NEAREST NEIGHBOR THINNING BASED ON THE IMORC INFRASTRUCTURE

Tobias Schumacher, Christian Plessl and Marco Platzner

Paderborn Center for Parallel Computing
University of Paderborn, Germany
{tobe|christian.plessl|platzner}@uni-paderborn.de

ABSTRACT

The creation and optimization of FPGA accelerators comprising several compute cores and memories are challenging tasks in high performance reconfigurable computing. In this paper, we present the design of such an accelerator for the k-th nearest neighbor thinning problem on an XD1000 reconfigurable computing system. The design leverages IMORC, an architectural template and highly versatile on-chip interconnect, to achieve speedups of 74× over a 2.2GHz Opteron. Using IMORC with its asynchronous FIFOs and bitwidth conversion in the links between the cores, we are able to quickly create acclerator versions with varying degrees of core-level parallelism and memory mappings. Through the performance monitoring infrastructure of IMORC we gain insight into the data-dependent behavior of the accelerator which facilitates further performance optimizations.

1. INTRODUCTION

Accelerator technologies such as general-purpose graphic processing units, SIMD floating point arrays, and FPGAs are steadily gaining importance in the area of high performance computing. However, implementing such accelerators as well as estimating their performance benefits before an implementation are still challenging tasks, especially for FPGA accelerators. In [1], we have presented IMORC, an *Infrastructure for performance Monitoring and Optimization of Reconfigurable Computers*. IMORC provides a designer with an architectural template for creating multi-core FPGA accelerators where the cores can communicate with each other using the IMORC interconnect. IMORC features methods for profiling the communication between the computing cores and, hence, allows a designer to identify performance bottlenecks and to optimize an accelerator. Particularly for accelerator architectures that generate data-dependent loads for their cores such advanced monitoring and profiling tools are indispensable.

This paper concentrates on a real-world case-study, the implementation of an accelerator for the k-th nearest neighbor (kNN) thinning problem on an XD1000 high performance reconfigurable computing system. While previously we have developed a kNN accelerator with a monolithic architecture that achieved speedups of one order of magnitude [2] over a contemporary workstation, we show in this paper how IMORC helps us to implement and optimize a highly flexible accelerator architecture utilizing several reconfigurable compute cores that delivers speedups of up to 74×.

Section 2 gives an overview over related work in infrastructures for reconfigurable accelerators. In Section 3, we briefly summarize the elements of the IMORC architectural template and show how different network topologies are created. Section 4 presents as a case study the k-th nearest neighbor thinning problem and the IMORC-based design of cores for it. The creation of an accelerator on the XD1000, as well as resource and performance measurements are discussed in Section 5. Section 6 concludes the paper and points to further work.

2. RELATED WORK

IMORC constitutes an architectural template for composing FPGA accelerators out of single computing cores and, as a main component of the architectural template, a highly versatile on-chip interconnect based on multiple buses with slave-side arbitration. Regarding the objectives, the SIMPPL approach presented in [3, 4] is similar to IMORC. Both architectures are using FIFOs to connect single cores. While IMORC differentiates between request and data channels in the links between cores, SIMPPL further decouples the data and control paths within the cores. Further, SIMPPL's communication structure is mainly targeted at 1:1 connections between cores, while one of IMORC's most important features is the slave side arbitration for connecting multiple cores to each other. IMORC additionally includes supporting cores such as farming cores for dynamic load balancing. While SIMPPL provides the ability to externally connect load sensors for identifying bottlenecks, such performance counters are an integral part of the IMORC on-chip interconnect.

Published in: *Proc. Int. Conf. on Field Programmable Logic and Applications (FPL)*

There also exists some related work in performance prediction for FPGA accelerators. Usually, these performance prediction methods rely on estimates for the computation times of the different cores and the amount of data transferred and parameters such as clock frequencies, bandwidths, and latencies for accessing memories and the host system, to estimate performance metrics with analytic formulae. Examples are given by Steffen [5], who provides a model for determining an application's suitability for FPGA acceleration, by the reconfigurable computing amenability test (RAT), a methodology presented by Holland et al. in [6], and by Smith and Peterson in [7], who present an analytical model for the expected load imbalance in multi-node systems.

IMORC is orthogonal to these performance prediction approaches and, as an architectural template, facilitates monitoring and performance optimization. After performance modeling and prediction, IMORC greatly supports a designer in mapping the cores and their logical communications to a synthesizable FPGA accelerator. In case of a model-based performance prediction, the models can even be verified and updated using the performance data measured by the IMORC infrastructure in the real system. A similar method for gathering such information during runtime is presented by Koehler et al. [8]. However, their work requires the designer to manually define load sensors while IMORC automatically provides several such sensors for monitoring the communication infrastructure.

3. IMORC SUMMARY

IMORC is an Infrastructure for Performance Monitoring and Optimization of Reconfigurable Computers. While we have presented a detailed overview of the infrastructure in [1], here we briefly summarize the main features of IMORC. IMORC cores can have an arbitrary number of master and slave ports. The infrastructure consists of links, connecting master to slave ports via asynchronous FIFOs, enabling each core to run at full speed in a separate clock domain. Bitwidth conversion modules inserted into the IMORC links enable cores to access data at their native data width. Hence, cores operating on 32bit integers can easily be connected to, for example, 256bit wide memory. Besides 1:1 connections, IMORC supports multiple master ports accessing a single slave port through the use of slave side arbitration. Conversely, a master port can operate in a multi or broadcast mode to reach several slave ports simultaneously. Load sensors are provided within the IMORC links for gathering performance information from the cores, e.g., how often FIFOs run full or empty. Additionally, user specified load sensors are supported. Further available infrastructure cores include the IMORC-to-Register converters for setting and reading control registers of cores, farming cores for dynamic load balancing and, depending on the target platform, memory controller and host interface cores.

4. ACCELERATOR DESIGN CASE STUDY

We demonstrate the IMORC workflow and benefits on the detailed example of developing a hardware accelerator for the k-th nearest neighbor thinning problem on the XtremeData XD1000 system [9]. In this section, we present the k-th nearest neighbor thinning problem and the application-specific IMORC cores that have been derived from it.

4.1. k-th Nearest Neighbor Thinning

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. The KNN approach is also widely applied for solving classifications problems, such as in machine learning, data mining and stochastic optimization [10]. There, a KNN classifier requires a labeled 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. Often, a reduction of the size of data samples is 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, that aim at properly selecting a subset of training vectors from the original data set.

In this paper we concentrate on KNN-based thinning. Given a set of input vectors, KNN-based thinning searches for the subset of vectors that represents the distribution of the original set the most representative. The KNN thinning 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 $\sigma_{\mathrm{i}}{}'$ 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.

4.2. IMORC KNN Cores

Algorithm 1 breaks up into three major blocks: computing the distance values, sorting the distance values, and discarding a vector from the distance matrix. We have organized our KNN accelerator in exactly these three phases and developed corresponding cores which are outlined below. At this stage in the design process, we have taken two decisions affecting the accelerator architecture. First, to reduce the amount of computations we have decided to move the computation and sorting of distance values out of the loop in Algorithm 1 and compute the sorted matrix $\underline{\sigma}$ only once. This is made possible by storing with each distance value σ_{il} the indices of the two vectors for which the distance has been computed. While increasing the amount of storage needed for the distance matrix, this allows for an efficient deletion of vectors from the distance matrix. Note that deleting a vector comprises the deletion of a complete row plus one cell in each of the remaining rows (cmp. Figure 1). In essence, we trade an increased memory requirement for improved performance. Second, we foresee two memory spaces, one for the vectors and one for the distance matrix. The cores are as follows:

4.2.1. Distance calculator core

The computation of the Euclidian distances, line 3 in Algorithm 1, is wrapped into a distance calculator core. We omit the square root function as squared values are sufficient for comparing distances. The distance calculator core computes one row of the distance matrix σ_i on each invocation, which allows to exploit parallelism by instantiating several such cores. To receive job messages, the distance calculator core contains a slave port. A job message comprises the base address of the vectors in vector memory, the number of vectors and dimensions, m and d, the index i of the vector for which the distances to all vectors have to be determined, and the address in distance memory where σ_i is to be stored.

The distance calculator core starts with retrieving the vector i from vector memory by issuing a read request over a master port. The vector i is stored within the core. Then, read requests are issued to vector memory to fetch all vectors. The elements of vector i and the incoming vectors are subtracted, squared, and added up. The resulting distances together with the indices i and l are written to distance memory. The distance calculator core signals its completion via another master port.

4.2.2. Sorter core

For the sorting step, line 4 in Algorithm 1, we use a variant of bubble sort. On each invocation, the sorter core sorts one row σ_i of the distance matrix, again allowing to exploit parallelism by instantiating several sorter cores. The job messages are received via a slave port and comprise the number of vectors m and the base address of σ_i in distance memory.

Then, the core proceeds as follows: In the first iteration, n elements of σ_i, i.e., the complete row, are read sequentially from distance memory over the sorter's master port. The first element is stored as the current maximum. If the next element is smaller or equal than the current maximum, it is directly written back to distance memory. If the next element is larger than the current maximum, it becomes the new current maximum and the previous maximum is written to distance memory instead. At the end of the iteration, the current maximum is written back. We additionally remember the position l of the last element in σ_i that has been

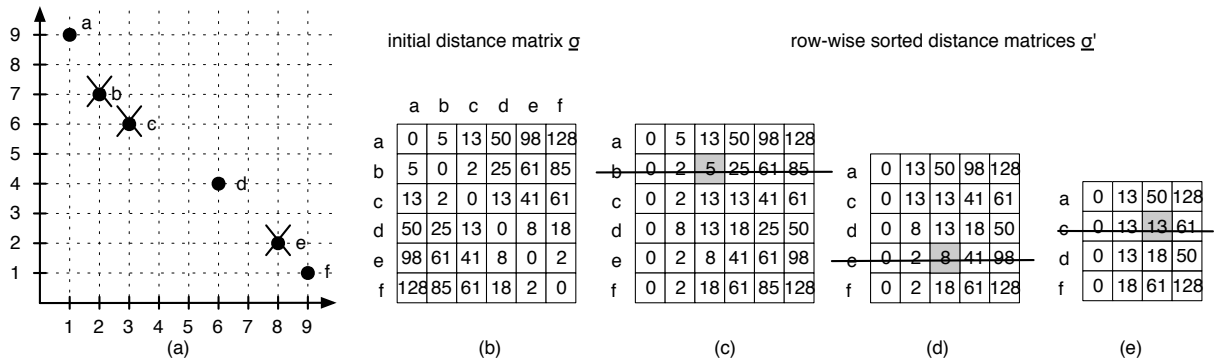

Fig. 1. Example for the KNN-based thinning algorithm in two dimensions: thinning 6 vectors to 3 vectors

moved left. Apparently, all elements right of σ_{il} are sorted after the first iteration. Thus, in the next iteration only l elements of σ_i are streamed through the sorter core, and so on, until the row is sorted.

The completion of a sorting job is signalled over the core's slave port instead of using another master port. This design decision is driven by the KNN thinning algorithm which requires us to have the complete matrix $\underline{\sigma}$ sorted before the deletion starts. Thus, the core that provides a sorter with jobs needs to be informed about the completion which is most easily done via the sorter's slave port.

4.2.3. Discarder core

The discarder core implements lines 5-12 of Algorithm 1 and is invoked with a job message specifying the total number of vectors n, the number of vectors to discard k, and the base address of the distance matrix $\underline{\sigma}$ in distance memory. The job message is received at a slave port. The discarder core operates in iterations, where in each iteration one vector is eliminated from $\underline{\sigma}$. As we have annotated each distance value with the indices of the two vectors used to compute the distance value, there is no need to recalculate and resort the distance matrix.

The core starts with reading column three of $\underline{\sigma}$ from distance memory. Similar to the sorter core, the discarder core scans the elements of a column to identify both the minimum element and whether it is unique. If the minimum is not unique, the next column is read in. Eventually, a unique minimum will be found in row σ_i; otherwise, the minimum value in the last column is treated as unique. Now, row σ_i and all distances to and from vector i have to be deleted from $\underline{\sigma}$. To this end, the discarder core reads the complete distance matrix with its $n \times n$ elements and writes back a thinned matrix with $n-1 \times n-1$ elements. Elements that have i as either source or destination vector are discarded; all other elements are written back. This process is iterated until $\underline{\sigma}$ is reduced to k vectors. Finally, the discarder core signals completion.

5. EXPERIMENTAL VALIDATION

In this section, we first briefly discuss the IMORC infrastructure for the XD1000 accelerator system. Then, we present the KNN accelerator architecture, detail experiments and report on results achieved for different accelerator configurations.

5.1. IMORC on the XtremeData XD1000 System

The XtremeData XD1000 is a dual socket Opteron system where one of the sockets is equipped with a 2.2GHz AMD Opteron processor, and the other one with a module featuring an Altera Stratix II EP2S180-3 FPGA. CPU and FPGA communicate via a HyperTransport link which provides a rather low latency communication. Physically, the FPGA is connected to the processor using a 16bit HyperTransport link with 800MT/s. While the original HyperTransport core provided by XtremeData only supports 8bit wide links, we have recently been able to integrate the 16bit core provided by the HyperTransport Center of Excellence of University of Heidelberg [11] into the IMORC framework. However, the results presented in this paper are generated using the 8bit core which does not pose a limitation as our benchmarks show that the bandwidth between host and FPGA is not the main bottleneck. The IMORC interface to the HyperTransport core provides three IMORC links, two master links that are directly mapped into two distinct address spaces of the host application and one IMORC slave interface that maps memory of the host application into the IMORC address space. Writes to the master links are generated by the host application by writing data into the appropriate address space, DMA writes from FPGA to host memory is performed by sending data to the IMORC slave link. At this side, a page mapping table is used for translating IMORC addresses into physical addresses of the host processor. This table has to be filled with physical page addresses by the host application prior to using this link.

Further, the XD1000 system comes with 4GB of main

Table 1. Resource requirements for the accelerator

	ALUTs	REGs	DSP	M512	M4k	M-Ram
EP2S180-3	143520	143520	768	930	768	9
distcalc core	1731	1642	18	-	-	-
sorter core	1117	1235	-	-	-	-
discarder core	1513	1642	-	-	-	-
CTRL core	782	325	-	-	-	-
IMORC link	98	142	-	-	4	-
bitwidth conv.	68	126	-	2	-	-
farming core	184	207	-	-	-	-
DDR CTRL	544	1536	-	-	8	-
HT core	3604	5353	-	28	2	-
complete acc.	31561	33465	108	67	386	8
IMORC	5896	4766	-	28	344	-

memory (DDR-SDRAM) for each the Opteron and the FPGA. All three interfaces, the interface to DDR SDRAM, the HyperTransport interface to host memory, and the interface to on-chip memory are completely identical allowing for transparently exchanging memory spaces.

5.2. KNN Accelerator Architecture

A complete IMORC architecture for KNN thinning comprises an arbitrary number of distance calculator and sorter cores, and one discarder core. IMORC farming cores distribute the load to the distance calculator and sorter cores, IMORC on-chip and off-chip memory controllers provide access to internal and external memory, and the IMORC HyperTransport core constitutes the host interface. Finally, a controller core is required to steer the overall accelerator.

Figure 2 shows a configuration of the KNN accelerator on the XD1000 system with three distance calculator and three sorter cores. The vectors are received from the host and stored in the vector memory which is mapped to on-chip storage elements. The distance memory is mapped to off-chip DDR-SDRAM.

Table 1 presents an overview of the resources used by the KNN accelerator, this time configured with six distance calculators and six sorter cores. The table lists the resources available on the Stratix II FPGA, the resources for the application cores, for different elements of IMORC and, finally, for the complete accelerator architecture and the IMORC infrastructure. Relative to the overall accelerator, IMORC requires some 19% of the ALUTs and 14% of the registers. Sorter and discarder cores are not particularly complex. The distance calculator core, on the other hand, performs more involved calculations but most of them are mapped to the FPGA's DSP blocks. Stratix II provides three different types of on-chip memory: M512s, M4ks and M-Rams. The small-sized M512s and the medium-sized M4ks are primarily used in the slave arbiters, a smaller number in the HyperTransport and the DDR memory controller cores. The four large M-Ram blocks are used for vector memory.

The complete accelerator design uses 10 different clock domains. Both the DDR memory controller and the HyperTransport core require three different clock domains where one of them is used by user logic. Another clock domain is assigned to the on-chip memory controller, and the remaining clock domains are used by the application cores, i.e., distance calculator, sorter, and discarder. In our current implementation, the distance calculator and sorter modules are running at 125MHz, the discarder at 150MHz, and the on-chip memory is configured for running at 200MHz. Overall, Stratix II provides up to 48 clock domains which allows to implement IMORC-based accelerators on the XD1000 with a reasonably large number of different cores.

5.3. Benchmarks

Using IMORC, we have been able to quickly synthesize four configurations of the KNN accelerator and compare their runtimes to a software reference implementation run on the 2.2GHz Opteron processor of the XtremeData XD1000 system. The accelerator configurations differ in the number of distance calculator and sorter cores, n_d and n_i, respectively, while in all cases one discarder core has been used. The first three configurations are $(n_d \times n_i) = (1 \times 1), (3 \times 3), (6 \times 6)$. Additionally, we have experimented with a (1×6) configuration as the sorter reveals a runtime quadratic in the number of distance values while the discarder's runtime depends only linearly on that number.

On the six accelerator configurations, we have run benchmarks with different problem sizes concerning the number of vectors n and the number of dimensions d. For all benchmarks, the thinning goal k was set to $n/4$, i.e., $(3/4)n$ vectors had to be discarded. As expected, for both the software reference and the hardware accelerators, the runtime increases dramatically with the number of vectors n. However, while the software runtime also significantly depends on the number of dimensions d, the accelerator runtime is only weakly dependent on d. Software runtimes range from about 11ms for a $(n, d, k) = (32, 2, 8)$ benchmark up to 23min for a $(1024, 128, 256)$ benchmark. In contrast, using the same benchmark the runtime for an accelerator in (1×1) configuration ranges from about 0.95ms up to 22.412s, and for an accelerator in (6×6) configuration from 0.55ms up to 18.840s.

Figure 3 presents the speedups for the different configurations over the software reference. The fact that the number of dimensions hardly influences the accelerator runtime can be clearly seen. The curves are grouped into three bundles, depending on the number of dimensions. For $d = 2$, the speedups for the different configurations range from $2.7\times$ to $5.8\times$. For $d = 32$, the speedups are around $20\times$ and for $d = 128$ up to $74\times$. Another effect observed in Figure 3 is that for a low number of vectors the different accelerator configurations produce very different speedups, while for larger n the speedups converge to the same value and satu-

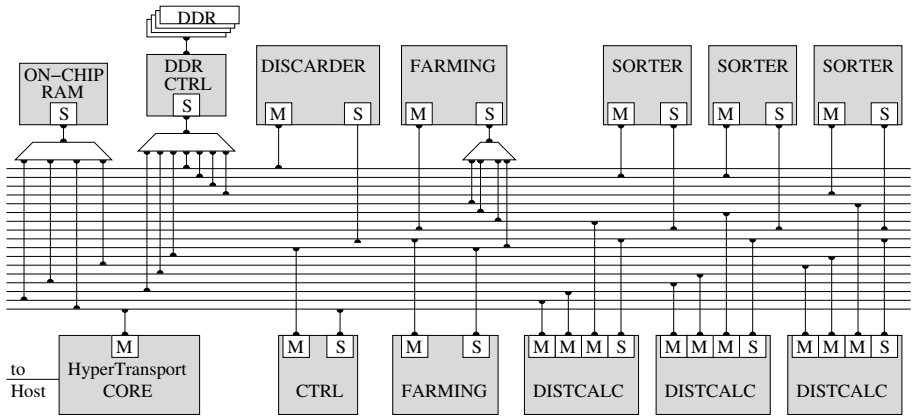

Fig. 2. IMORC KNN accelerator with three distance calculator and sorter cores mapped to the XD1000 system

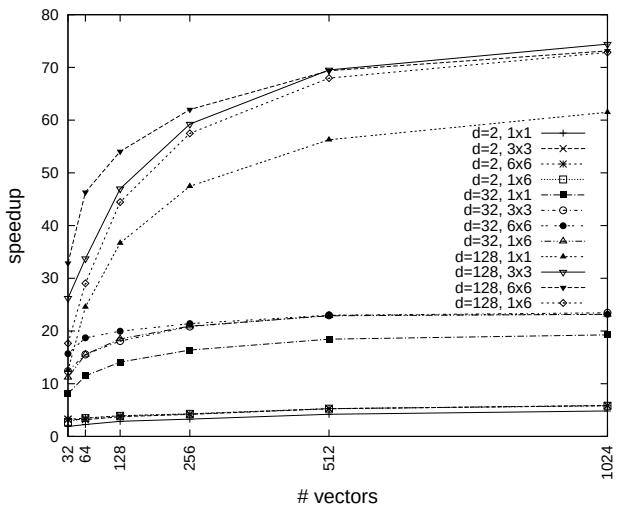

Fig. 3. Speedups for different KNN accelerator configurations and number of dimensions, depending on the number of vectors

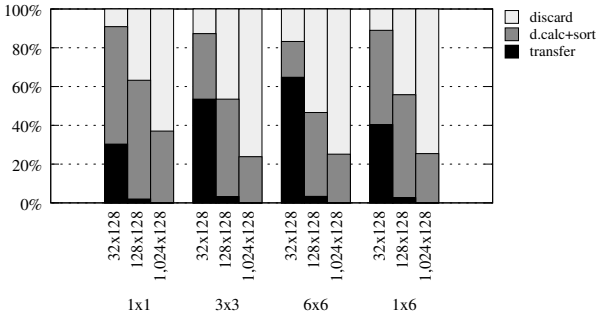

Fig. 4. Execution time breakdown into three phases transfer, distance calculation / sorting, and discarding for 32/128/1024 vectors with 128 dimensions and different accelerator configurations

rate. An exception is the accelerator configuration with only one distance calculator and sorter core which is consistently slower than configurations exploiting core-level parallelism.

To explain this behavior and determine the relative runtimes of the different cores in the accelerator, we have used the performance counter infrastructure provided by IMORC. Figure 4 visualizes the runtime breakdown for following three phases of the execution: the transfer of vectors from the host to the XD1000 system, distance calculation and sorting combined, and discarding. Distance calculation and sorting has been combined into one phase, as both cores are running in parallel most of the time. Figure 4 shows that for a low number of vectors the data transfer constitutes a substantial part of the overall runtime. For parallel accelerator configurations, (3×3) and (6×6), the data transfer is even dominating. Another substantial part of the runtime is spent for distance calculation and sorting. While the absolute transfer time is about constant for given n and d, the time for distance calculation and sorting depends on the number of cores used. Importantly, with growing n the overall execution time of the accelerator is increasingly dominated by the runtime of the discarder core. For n exceeding 512, the runtimes for configurations using parallel sorter cores are almost identical. To create improved KNN accelerators for higher numbers of d and n, it is thus of utmost importance to parallelize the discarder function.

6. CONCLUSION & FUTURE WORK

In this paper we have presented a KNN accelerator as a case study for the IMORC architectural template and on-chip interconnect. Farming cores have enabled us to parallelize runtime intensive functions with multiple hardware cores, which has resulted in an accelerator that can be scaled

with minimal changes in the design. The performance optimization process has been guided by data gathered from the IMORC performance counters, which has resulted in speedups of up to $74\times$ over a commodity processor.

The case study has underlined the importance of a performance monitoring infrastructure. Much like established profiling tools for software, IMORC provides runtime breakdowns and allows to identify bottlenecks in more complex accelerator architectures with data-dependent behavior. It must be emphasized that IMORC monitors the running accelerator in real-time instead of a simulation model. While for $n = 32$ the KNN accelerator design in VHDL can easily be simulated using a testbench, for $n \geq 128$ such a simulation takes more than five days on a modern workstation. We have measured $8.6 \cdot 10^6$ cycles on the simulating machine for one cycle of the simulated accelerator. During the design of the KNN accelerator we could identify several bottlenecks that only appeared on such large problem sizes.

In future we plan to extend our work in two main directions. First, we plan to work towards an integrated performance analysis and optimization approach that uses data gathered by the performance counters to calibrate analytical or statistical performance models. Second, we will look into automatically mapping application models to IMORC architectures. While in our case study we have manually analyzed the KNN thinning algorithm and decided which cores to implement and how to schedule them, more formal application models, e.g., process networks, might allow us to derive core topologies and schedules automatically.

7. REFERENCES

[1] T. Schumacher, C. Plessl, and M. Platzner, "IMORC: Application Mapping, Monitoring and Optimization for High-Performance Reconfigurable Computing," in *Proc. IEEE Symp. on Field-Programmable Custom Computing Machines (FCCM '09)*. IEEE, 2009.

[2] T. Schumacher, R. Meiche, P. Kaufmann, E. Lübbers, C. Plessl, and M. Platzner, "A hardware accelerator for k-th nearest neighbor thinning," in *Proc. Int. Conf. on Engineering of Reconfigurable Systems and Algorithms (ERSA)*. CSREA Press, 2008, pp. 245–251.

[3] L. Shannon and P. Chow, "Simplifying the Integration of Processing Elements in Computing Systems Using a Programmable Controller," in *Proc. IEEE Symp. on Field-Programmable Custom Computing Machines (FCCM '05)*. IEEE, 2005, pp. 63–72.

[4] ——, "SIMPPL: an adaptable SoC framework using a programmable controller IP interface to facilitate design reuse," *IEEE Trans. Very Large Scale Integr. Syst.*, vol. 15, no. 4, pp. 377–390, 2007.

[5] C. Steffen, "Parametrization of algorithms and FPGA accelerators to predict performance," in *Proc. Reconfigurable System Summer Institute (RSSI)*, 2007, pp. 17–20.

[6] B. Holland, K. Nagarajan, C. Conger, A. Jacobs, and A. D. George, "RAT: A Methodology for Predicting Performance in Application Design Migration to FPGAs," in *Proc. High-Performance Reconfigurable Computing Technologies and Applications Workshop (HPRTCA)*, 2007.

[7] M. Smith and G. Peterson, "Analytical Modeling for High Performance Reconfigurable Computers," in *Proc. Int. Symp. on Performance Evaluation of Computer and Telecommunication Systems (SPECTS02), San Diego, California*, July 2002.

[8] S. Koehler, J. Curreri, and A. D. George, "Performance analysis challenges and framework for high-performance reconfigurable computing," *Parallel Computing*, vol. 34, no. 4-5, pp. 217–230, 2008.

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

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

[11] D. Slogsnat, A. Giese, and U. Brüning, "A versatile, low latency HyperTransport core," in *FPGA '07: Proc. 2007 ACM/SIGDA 15th Int. Symp. on Field Programmable Gate Arrays*. New York, NY, USA: ACM, 2007, pp. 45–52.