A Resource-Aware Nearest-Neighbor Search Algorithm for K-Dimensional Trees

Johny Paul
and Walter Stechele
Institute for Integrated Systems
Technical University of Munich
Germany
{johny.paul,walter.stechele}@tum.de

Manfred Kroehnert
and Tamim Asfour
Institute for Anthropomatics
Karlsruhe Institute of Technology
Germany
{manfred.kroehnert,asfour}@kit.edu

Benjamin Oechslein, Christoph Erhardt,
Jens Schedel, Daniel Lohmann
and Wolfgang Schröder-Preikschat
Department of Computer Science
FAU Erlangen-Nuremberg, Germany
{oechslein,erhardt,schedel,
lohmann,wosch}@cs.fau.de

Abstract—**Kd-tree search is widely used today in computer vision – for example in object recognition to process a large set of features and identify the objects in a scene. However, the search times vary widely based on the size of the data set to be processed, the number of objects present in the frame, the size and shape of the kd-tree, etc. Constraining the search interval is extremely critical for real-time applications in order to avoid frame drops and to achieve a good response time. The inherent parallelism in the algorithm can be exploited by using massively parallel architectures like many-core processors. However, the variation in execution time is more pronounced on such hardware (*HW*) due to the presence of shared resources and dynamically varying load situations created by applications running concurrently. In this work, we propose a new resource-aware nearest-neighbor search algorithm for kd-trees on many-core processors. The novel algorithm can adapt itself to the dynamically varying load on a many-core processor and can achieve a good response time and avoid frame drops. The results show significant improvements in performance and detection rate compared to the conventional approach while the simplicity of the conventional algorithm is retained in the new model.**

I. Introduction

In computer science, a *k-dimensional tree* or *kd-tree* is a space-partitioning data structure for organizing points in a k-dimensional space. In other words, kd-trees are a special case of binary space-partitioning trees, where every node is a k-dimensional point. Kd-trees are useful data structures for several applications such as range searches and nearest-neighbor searches (*NN-searches*). The NN-search algorithm aims to find the point in the tree that is nearest to a given input point. An efficient search can be implemented by taking advantage of the kd-tree properties leading to a quick search-space reduction. Further speedups can be achieved by using an approximation algorithm. For example, an approximate NN-search can be achieved by simply setting an upper bound on the number of points to examine in the tree, or by interrupting the search process based on a real-time clock (which may be more appropriate in HW implementations). Approximate nearest-neighbor search is useful in real-time applications such as robotics due to the significant speedup achieved by not searching for the best point exhaustively.

The NN-search problem arises in numerous fields of application including computer vision, pattern recognition, statistical classification, computational geometry, data compression, DNA sequencing, cluster analysis, etc. In the context of 3D vision, NN-search is frequently used in 3D point-cloud registration. A comparison between these techniques is available in [8]. However, the registration of large data sets is computationally expensive. As an example, the humanoid robot ARMAR-III [1] is capable of recognizing and tracking textured objects. The recognition algorithm uses a combination of Harris Interest Points and SIFT feature descriptors as described in [2]. Features extracted from the scene are matched against a pre-computed object database using a heuristic NN-search. This enables the processing of large numbers of features on every object to be recognized and tracked. A high recognition frame rate is achieved by using the best-bin-first search algorithm [3].

In recent years, growing interest has been attracted by many-core processors on account of their immense computational power assembled in a compact design. The compute-intensive nature and the high degree of inherent parallelism in the NN-search algorithm makes it suitable for implementation on many-core processors. However, the available resources on a many-core (processing elements (PE), memories, interconnects, etc.) have to be shared among various applications running concurrently, which leads to unpredictable execution time or frame drops during NN-search. To address these challenges, this paper proposes a novel resource-aware NN-search algorithm for kd-trees. This work describes how to distribute the huge workload on the massively parallel PEs for best performance, and how to generate results on time (avoiding frame drops) even under varying load conditions.

II. State of the Art

NN-search on kd-trees is a compute-intensive task with a high degree of inherent parallelism, and can be accelerated using multi-core CPUs or GPUs. A recent study suggested performing NN-search on the GPU using the basic brute-force technique [10]. The brute-force search is a simple search

This work was supported by the German Research Foundation (DFG) as part of the Transregional Collaborative Research Centre "Invasive Computing" (SFB/TR 89) and the Priority Program "Dependable Embedded Systems" (SPP 1500).

technique that compares one element with every other element in the database. GPUs have impressive brute-force search performance. However, GPU architecture makes efficient data-structure design quite difficult. In particular, GPUs are vector-style processors with limited branching ability. Hence, conditional computation typically under-utilizes these devices seriously [7]. Brute-force search on the GPU is still much faster than it is on the CPU, but not much faster than a kd-tree-based NN-search on a CPU [6]. Because of the ubiquity of the NN-search problem, a huge variety of data structures and algorithms have been developed to accelerate this process. Cayton et al. [6] introduced a simple data structure for NN-search on the GPU, with search and build algorithms that are efficient on parallel systems. However, the authors state that a significant effort was required to develop the GPU software in comparison to the well-known and simple NN-search on a CPU.

For almost four decades, better and better integration technology has been allowing to double the number of transistors on a single processor chip every 2 years. Therefore, manufacturers were able to implement 8 to 12 high-end superscalar processors or up to 100 simple cores on a single die. Today it is possible to put on the same chip a large number of general-purpose cores, certainly tens of highly complex cores as on Intel's Single-Chip Cloud Computer [12] or Tilera's 64-core processor [4]. Such architectures can overcome the limitations imposed by multi-core platforms with a limited number of cores, and the high degree of parallelism within the HW can lead to a significant acceleration of the conventional and simple NN-search algorithm for kd-trees [7].

A major challenge associated with future many-core systems is the question of how to program such systems to make best use of their computing power. Heat, power dissipation, reliability, etc. are other issues of future transistor technology which have appeared on the horizon and need to be tackled together with new means for program development of concurrent applications. In order to address these issues [11] propose a new resource-aware operating system (ROS) for many-core HW, with direct support for parallel applications and a scalable kernel. This work describes a resource-management scheme based on resource provisioning which enables system-wide, efficient accounting and utilization of resources. In that work, resources such as cores and memory are explicitly granted to the applications and revoked. The kernel exposes information about a process's current resource allocation and the system's utilization, and allows the application programs to make requests based on this information.

The demand for more stringent (OS-supported) resource awareness was also proposed in [15], put forward by a new programming methodology called *Invasive Computing*. The main idea and novelty of Invasive Computing is that it extends resource-aware programming support to various layers in the many-core system like resource-aware OS, communication interfaces like Network-on-Chip (NoC) and processing elements (PEs). This research also focuses on policies for resource allocation and when and how to revoke resources from a process. Programs running on this HW get the ability to explore and dynamically spread their computations to neighboring processors and execute portions of code with a high degree of parallelism in parallel based on the availability of resources. Once the program terminates or if the degree of parallelism should be lower again, the program may enter a *retreat* phase. At this point, the resources can be deallocated and execution resumed, for example, sequentially on a single processor.

III. Motivation

For applications like real-time object recognition and tracking, the NN-search algorithm has to complete the search process within a predefined time. For fast-moving objects the search interval has to be reduced so that the object can be tracked accurately. This can be achieved by using more PEs of the many-core processor. The actual duration of the search depends on the size of the data set to be processed and the size of data set depends on the number of objects present in the scene, the nature of the background, lighting conditions, etc. Fig. 1 shows the variation in execution time when the NN-search is performed on a kd-tree of SIFT features (used by the ARMAR robot to recognize objects). In this

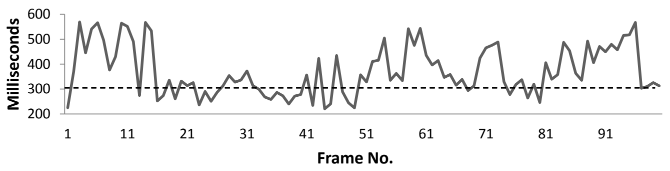


Fig. 1. Execution time for NN-search (static allocation)

case the application is statically scheduled on 16 concurrent PEs to ensure that the available computing power does not vary over time. A sequence of hundred different scenes was processed by the algorithm and the size of the data set for kd-tree search varies in every scene. It can be seen that the execution time varies between 200 and 600 milliseconds, based on the size of the input data set. However, this evaluation is not complete as the static resource allocation is not a recommended approach, and results in poor resource usage. This is evident from Fig. 1 where the resources were allocated such that the application can process one frame every 300 milliseconds (represented by the dotted line). However, the NN-search duration falls below or above the deadline based on the number of features to be processed. Points where the execution time falls below the deadline represent under-utilization of allocated resources, while those above the line indicate a lack of sufficient resources.

The execution time can be equalized by adding more cores to the application when higher computing power is required and vice versa. Furthermore, the impact of other applications running concurrently on the many-core system (audio processing, robot control etc.) has to be considered. These applications create dynamically changing load on the processor based on what the robot is doing at that point in time. For instance, the speech-recognition application is scheduled when the user speaks to the robot or the motor control is activated when the robot has to move or grasp an object that it recognized. The conventional OS scheduler schedules the threads of each application considering the overall system load. As a result, the resources available to each application may vary from time to time. Such a situation is depicted in Fig. 2, where the y-axis represents the number of PEs allocated to the NN-search application for each frame, for a total of 100 frames. The

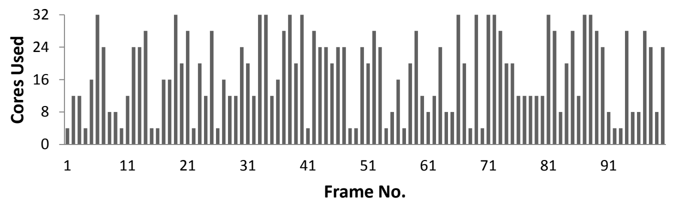


Fig. 2. Dynamic resource usage during NN-search

resulting execution time is depicted in Fig. 3 and the results indicate very high jitter in the execution time. This evaluation reveals the highly unpredictable search duration for NN-search on today's many-core processors. Prolonged search durations would lead to frame drops and tracking errors. The number of frames dropped during this evaluation was as high as **44%**. The robot may even lose track of the object if too many consecutive frames are dropped.

In order to overcome these challenges, we developed a new NN-search algorithm for many-core processors where the application program can request for resources and adapt the current workload based on the available resources. The upcoming sections demonstrate how the resources are claimed and how the search interval is constrained to guarantee better response time, compared to conventional many-core systems. The results indicate a significant improvement in overall results of the recognition process when the NN-search is performed on the new resource-aware platform.

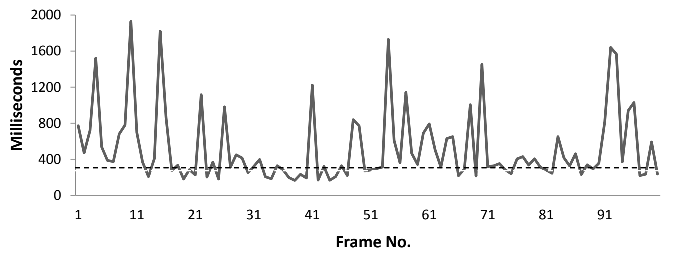


Fig. 3. Execution time for NN-search (dynamic allocation)

IV. NEAREST-NEIGHBOR SEARCH ON KD-TREES

The object-recognition process used on the ARMAR robot consists of two steps. In the first step, the robot is trained to recognize the object. A training data set consisting of SIFT features is created for every object to be recognized. To speed up the nearest-neighbor computation, a kd-tree is used to partition the search space; one kd-tree is built for each object. The second step in the recognition process has real-time requirements as it helps the robot to interact with its surroundings (by recognizing and localizing various objects) in a continuous fashion. In this step, a set of SIFT features, extracted from the real-time images, is compared (using NN-search) with the pre-loaded data set (kd-tree). The computation of the nearest neighbor for the purpose of feature matching is the most time-consuming part of the complete recognition and localization algorithm. This algorithm performs a heuristic search and only visits a fixed number of leaves resulting in an actual nearest neighbor, or a data point close to it. For the NN-search algorithm, the number of kd-tree leaves visited during the search process determines the overall quality of the search process. Visiting more leaf nodes during the search leads to a higher execution time. The search duration per SIFT feature can be calculated from Fig. 4. The values were captured by running the NN-search application on a single PE using a library of input images covering various situations encountered by the robot. From this graph it is clear that the search interval

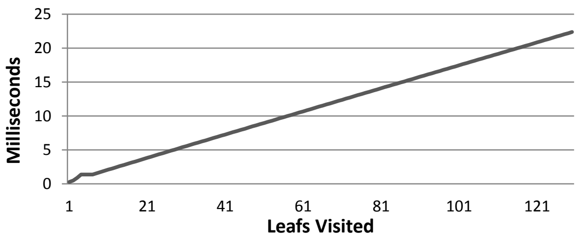


Fig. 4. Variation of execution time vs. leaf nodes visited for NN-search

varies linearly with the number of leaf nodes visited during the search. Moreover, the relation between quality (i.e. the number of features recognized) and leaf nodes is shown in Fig. 5. The quality of detection falls rapidly when the number of leaf nodes is reduced below 20 and increases linearly in the range between 20 and 120. At a further higher leaf count, the quality does not improve significantly as all the possible features are already recognized. In the conventional algorithm used on CPUs, the number of leaf nodes visited is set statically such that the search process delivers results with sufficient quality for the specific application scenario. Using the results from this evaluation, the overall search duration can be predicted based on the object to be recognized, the

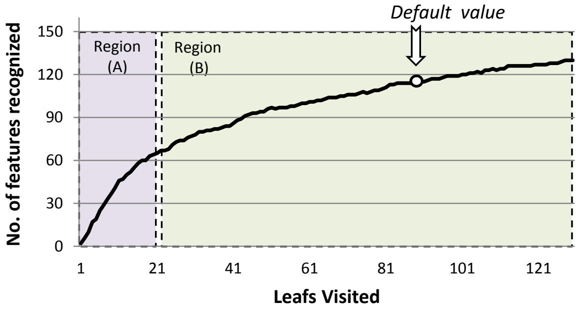


Fig. 5. Search quality vs. leaf nodes visited for NN-search

number of features to be processed and the number of PEs available for NN-search. The first two parameters are decided by the application scenario while the PE count is decided by the runtime system based on the current load situation. As shown in Fig. 2, the resources allocated to the application may vary from time to time, leading to highly unpredictable search durations and frame drops. Two different techniques can be applied to the conventional algorithm to constrain the execution time, as described below.

A. Threshold-Based Search

In order to avoid frame drops and to improve the tracking accuracy, the conventional NN-search can be modified to process the SIFT features based on their quality. The number of leaf nodes visited is fixed (to the default value) and once the deadline is hit, the algorithm can drop the remaining low-quality features and move on to the next frame. This technique is relatively simple, easy to implement and works on any single-/many-core platform. The algorithm would perform well in scenarios where the scene contains only the object to be recognized and all the detected features belong to the same object. The results deteriorate when there are more objects in the frame and also in scenes with cluttered background; this is because some of the high-quality SIFT features may belong to other objects or to the background. Therefore, the features dropped by the algorithm may belong to the target object, leaving it undetected.

B. Iterative Search

An alternative approach to overcome the problems in the threshold-based search is to modify the conventional NN-search to proceed in an iterative manner. This means that the search process starts with the first feature and performs a search until the first leaf node is reached. The results are saved and the algorithm moves on to the next feature and repeats the same process again. Thus, every feature in the data set is processed once (to the first leaf node) and then the algorithm returns to first feature again to continue the search from the first leaf node to the second. This process continues until a default number of leaf nodes is visited or until the next frame is available, whichever occurs first. In this manner, when the deadline is hit, the algorithm will have performed an equal search for all the features in the data set and none of the features will be dropped. However, the overall quality of the NN-search is reduced as described in Fig. 5 as the search process may stop half-way without reaching the default leaf count. Detailed evaluations were performed to compare the results from the threshold-based and iterative NN-search and the results are provided in Section VII.

V. Evaluation Platform

As described in Section II, our work focuses on exploring the benefits of resource-aware NN-search on kd-trees. Therefore, we implemented our algorithms on top of OctoPOS [13], a resource-aware operating system. OctoPOS shares the same view with ROS [11] as far as application-directed resource management of many-core processors is concerned. Also, both approaches resort to an event-based kernel architecture and largely benefit from asynchronous and non-blocking system calls. The main difference, however, is in the execution model of OctoPOS that was specifically designed to support invasive-parallel applications.

A. System Programming Interface

At the OctoPOS interface, resource-aware programming maps to three fundamental system calls: `invade()`, `infect()` and `retreat()`. These calls and their typical usage in the course of an application programm are depicted in Fig. 6. First, the application's resource demand has to be

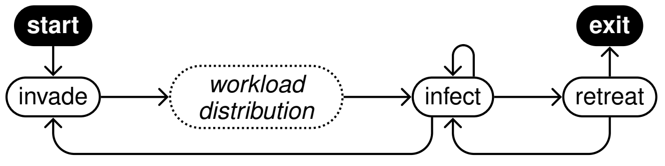


Fig. 6. Structure of an invasive program

expressed to the system. We call this the *invade* phase. As a result from the *invade* call, the application is handed a set of resources in the form of a *claim*. A claim is the central data structure in the system for representing the resources (processors, memory, etc.) associated with an application. When *invade* returns, the application has to distribute its workload according to the resources it acquired. For example, it can tune its algorithm towards the number of processors present in the claim. The actual computation is then started using the *infect* call. After execution finishes, another computation phase can be started on the same set of resources or resources can be released using *retreat* or additional resources can be acquired using *invade*. The basic concept of resource-aware computing states that an application dynamically expands and shrinks its set of resources at runtime according to its own demand and that it can react to undersupply situations where there are not enough resources available. Depending on the current system state, the resulting claim may or may not fulfill the demands specified before. Once an application gets a claim, it has full control over the associated resources. This guarantee on the acquired resources enables the application to balance its workload according to the dynamic runtime state of the system. Assumptions made during workload distribution, right before the *infect* phase hold until the application itself changes resource allocation following the *infect* phase.

B. Application execution model

The main building blocks of applications in OctoPOS are so-called *i*-lets: Fragments of a program potentially executed in parallel with mostly run-to-completion semantics. These are represented by function and data pointers and thus are very lightweight entities. An i-let is like a Cilk procedure [5], but allows for the blocking of its executing thread by creating a "featherweight" continuation when actually releasing a PE. An application can create an arbitrary number of *i*-lets to

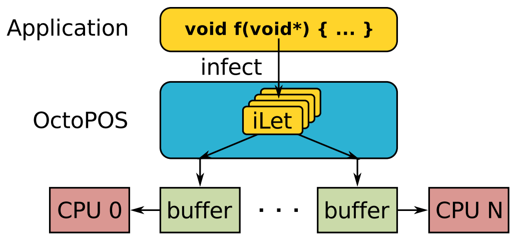


Fig. 7. Execution model of applications in OctoPOS

potentially be executed in parallel using the *infect* system call. As depicted in Fig. 7, OctoPOS forwards them to local buffers for each of the respective processors where they are eventually executed. Overall, this leads to an efficient implementation of *i*-let creation and dispatching. Moreover, with a tiled hardware architecture as described in Section V-C, the buffering scheme is a possible candidate for hardware acceleration: To execute *i*-lets on distant tiles without obstructing the processors in the tile, the buffers can be maintained in hardware and accessed directly through the NoC. This leads to a very scalable system architecture suitable for many-core systems.

C. Hardware Architecture

Our target many-core processor is shown in Fig. 8. The processor comprises 9 tiles interconnected by a NoC. Each tile consists of 4 cores interconnected by a local bus and some tile-local memory (TLM), with a total of 32 cores (LEON3, a SPARC V8 design by Gaisler [9]) spread across 8 tiles. The 9th tile is a memory and I/O tile encompassing a DDR-III memory controller and Ethernet, UART, etc. for data exchange and debugging. Each core has a dedicated L1 cache and all the cores within a tile share a common L2 cache. L1 caches are write-through and L2 is a write-back cache; this setup ensures cache coherency within tile boundary. However, no cache coherency is maintained beyond tile boundary, and data consistency has to be handled by the programmer through proper programming techniques, similar to the Intel SCC.

VI. Resource-Aware Nearest-Neighbor Search Algorithm

Sections III and IV described the conventional algorithm, its limitations, and also presented two modified algorithms for object tracking on conventional many-core platforms. This section describes our new approach towards NN-search using the idea of resource-aware computing. The main idea and novelty of the new algorithm is that the workload is calculated and distributed taking into account the available resources (PEs) on the many-core processor.

A. Adaptive Workload Distribution

As described in Section V, the first step is to allocate sufficient resources to perform a parallel NN-search. The amount of PEs requested by the algorithm is based on the number of SIFT features to be processed, the size of the kd-tree and the available search interval. The number of SIFT features varies from frame to frame based on the nature and

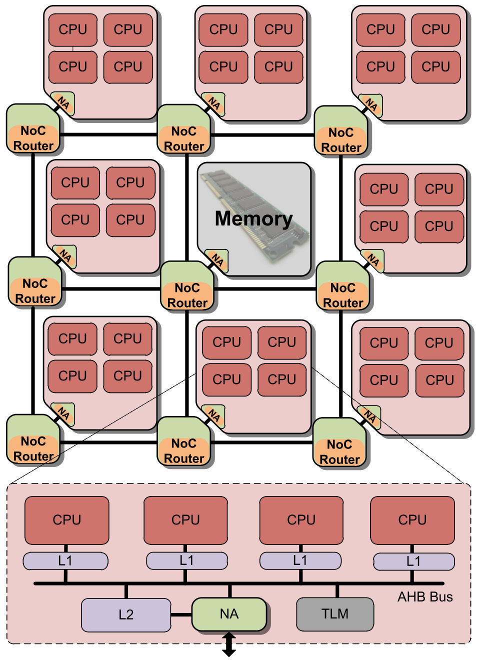


Fig. 8. Resource aware many-core processor

number of objects present in the frame, the nature of the background, etc. The size of the kd-tree is decided by the texture pattern on the object to be recognized and tracked. The search interval or the frame rate is decided by the context where the NN-search is employed. For example, if the robot wants to track a fast-moving object, the frame rate has to be increased or the execution time has to be reduced. Equation (1) represents this relation and can be used to compute the number of PEs (N_{pe}) required to perform the NN-search on any frame within the specified interval T_{search}. N_{fp} is the number of SIFT features to be processed and T_{fp} is the search duration per SIFT feature, a function of the number of leaf nodes visited, as described in Fig. 4. The initial resource estimate is based on the default leaf count (N_{leaf_best}), as described in Section IV.

$$N_{pe} \geq \frac{N_{fp} \times T_{fp}(N_{leaf\_best})}{T_{search}} \quad (1)$$

Note that the function $T_{fp}(N_{leaf\_best})$ is different for every object to be recognized and tracked by the robot, as this is dependent on the number of features forming the kd-tree, the shape of the tree, etc.

B. Efficiency Graphs

Equation (1) assumes that the search interval decreases linearly with increasing PEs. Such an assumption does not hold, considering the limited parallelism within the application program. For example, the NN-search algorithm is highly parallel during the search process. However, the overall execution time also includes the time to load the kd-tree to on-chip memory, combine the results from individual i-lets, filter the best matches, etc. Furthermore, every additional i-let created by the NN-search algorithm also creates an additional load on the external memory and shared communication interfaces, limiting the scalability. Our analysis of NN-search on the proposed HW shows an efficiency graph as shown in Fig. 9. From the graph it can be seen clearly that when the number of i-lets is increased from 1 to 2, the execution time does not improve by 2x, instead by 2×0.98 (98%), i.e. 1.96x. Using this graph, the efficiency factor for various levels of parallelism can be computed. The values shown here are applicable only for the NN-search implementation used in this paper and may vary based on how the original algorithm is implemented. In order

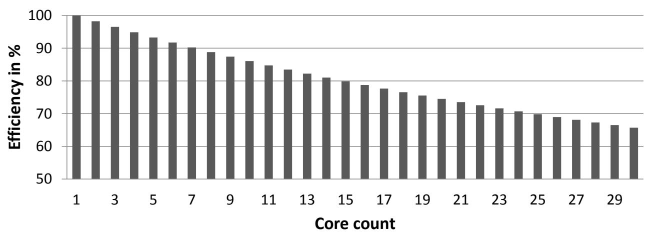


Fig. 9. Efficiency map for NN-Search on target HW

to increase the accuracy of the resource-estimation model, an application-specific efficiency factor or scalability information can be added to (1). The enhanced model is represented by (2), where $\eta(N_{pe})$ represents the algorithm's efficiency as a function of degree-of-parallelism or available resources (N_{pe}).

$$N_{pe} \geq \frac{N_{fp} \times T_{fp}(N_{leaf\_best})}{T_{search} \times \eta(N_{pe})} \tag{2}$$

Using the new model, the application raises a request to allocate PEs (N_{pe}), which is then processed by the operating system. Considering the current system load, the OS makes a final decision on the number of PEs to be allocated to the NN-search algorithm. The PE count may vary from zero (if the system is too heavily loaded and no further resources can be allocated at that point in time) to the total number of PEs requested (provided that there exists a sufficient number of idle PEs in the system and the current power mode offers sufficient power budget to enable the selected PEs). This means that under numerous circumstances the application may end up with fewer PEs and has to adapt itself to the limited resources offered by the runtime system.

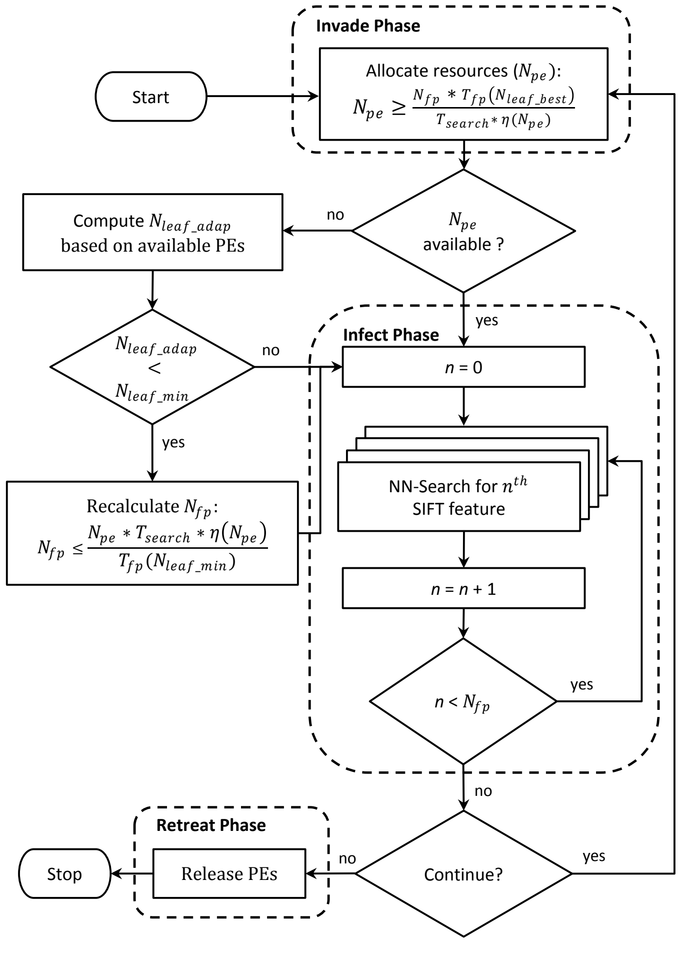
C. Resource-Aware Workload Distribution

In constrained scenarios as explained above, the application has to re-balance the workload in order to complete the NN-search within the search interval specified by T_{search}. This is achieved by recalculating the number of leaf nodes (N_{leaf_adap}) to be visited during the NN-search such that the condition in (3) is satisfied.

$$T_{fp}(N_{leaf\_adap}) \leq \frac{N_{pe} \times T_{search} \times \eta(N_{pe})}{N_{fp}} \tag{3}$$

The algorithm can use the new leaf count for the entire search process on the current image. However, it can be seen from Fig. 5 that the quality drops significantly if the number of leaf nodes calculated in (3) is too low (between 0 and 20, for the particular object used in our evaluation). This region is marked as region(A) in the figure. This issue can be resolved by preventing the leaf count from falling below the minimum leaf count (N_{leaf_min}) or the estimated leaf count should fall within the region(B) as shown in Fig. 5. Under most circumstances the application can process all the features by adapting the leaf count. However, if there are a large number of features to be processed using too few PEs, the leaf count calculated may fall below the minimum limit. In this case, the NN-search will drop a few low-quality SIFT features to complete the search within the predefined search interval.

It should be noted that the resource-allocation process operates once for every frame. Upon completion, the application releases the resources and waits for the next frame to arrive. Therefore, the core algorithm for NN-search retains its simple structure in contrast to the complex iterative search algorithm described in Section IV-B. The flow diagram in Fig. 10 describes the entire process of resource allocation and workload calculation for resource-aware NN-search. It also shows how this maps to the programming interface of the underlying OS. In contrast to the threshold-based search described in Section IV-A, the resource-aware NN-search



Invade Phase
Start
Allocate resources (N_{pe}):
$N_{pe} \geq \frac{N_{fp} * T_{fp}(N_{leaf\_best})}{T_{search} * \eta(N_{pe})}$
N_{pe} available ?
no
yes
Compute N_{leaf_adap} based on available PEs
$N_{leaf\_adap} < N_{leaf\_min}$
no
yes
Recalculate N_{fp}:
$N_{fp} \leq \frac{N_{pe} * T_{search} * \eta(N_{pe})}{T_{fp}(N_{leaf\_min})}$
Infect Phase
$n = 0$
NN-Search for n^{th} SIFT feature
$n = n + 1$
$n < N_{fp}$
yes
no
Retreat Phase
Stop
Release PEs
no
Continue?
yes

Fig. 10. Flow diagram for resource-aware NN-search

can process all features in most cases and hence overcome the challenges raised by scenes with multiple objects or cluttered backgrounds. As the available resources are known in advance, the application can make an early decision on the number of leaf nodes to be visited during NN-search and can avoid the additional complexity encountered by the iterative approach described in Section IV-B.

VII. Evaluation & Results

This section describes the results obtained from the resource-aware NN-search along with a comparison with the conventional search techniques like threshold-based and iterative search. A set of 100 different scenes was used for evaluation, where each frame contains the object to be recognized and localized along with few other objects and changing backgrounds. The position of the objects and their distance from the robot were varied from frame to frame to cover different possible scenarios. Evaluations were conducted on the FPGA-based HW prototype described in Section V. As a single FPGA cannot hold the large many-core design (with 32 LEON3 cores, external memory controllers, multiple debug and I/O interfaces, on-chip memories, NoCs, etc.) a multi-FPGA prototyping platform from Synopsys called CHIPit System [14] was used. This system consists of six Xilinx FPGAs (Virtex-5 XC5VLX330) with a total capacity of 12 million ASIC gates. The design operates at a frequency of 50 MHz. Although the operating frequency of the HW prototype is relatively low compared to an ASIC implementation, it does not affect the evaluation process as all three versions of the algorithm were tested on the same FPGA platform and the results are compared in terms of the quality of the detection process with a fixed search interval.
All three flavors of the NN-search algorithm were tested using the same set of input images for a search interval of 300 milliseconds per frame (3.3 frames per second). Fig. 11 shows a comparison between the resource-aware and the threshold-based NN-search, with the number of features recognized (quality of detection) on the y-axis and the frame number on the x-axis. In order to maintain equality in the evaluation process, the number of PEs allocated to the applications was equalized. The PE distribution varies from frame to frame as shown in Fig. 2. The remaining resources were either idle or allocated to other audio/video or motion-control applications running on the robot. It is clear from Fig. 11 that the resource-aware NN-search algorithm outperforms the conventional algorithm using the same amount of resources. This is because the resource-aware model is capable of adapting the search algorithm based on the available resources compared to the conventional algorithm with fixed thresholds. However, the resource-aware algorithm results in the same number of matched features as the conventional algorithm in some frames. This is because there were a sufficient number of idle PEs and the runtime system allocated sufficient resources to meet the computing requirements of the conventional algorithm and hence the conventional algorithm did not drop any SIFT feature. On the contrary, when a frame contains large number of SIFT features and the processing system is heavily loaded by other applications, the conventional algorithm dropped too many SIFT features, thereby resulting in a low overall detection rate (matched features). This behavior is depicted in Fig. 12 and Fig. 13. Fig. 12 contains the object to be recognized along with a tiny cup and plain background. Hence the features dropped were the low-quality features on the target and most of the high-quality features were still retained, resulting in a comparable detection rate with threshold-based and resource-aware search algorithms. However, in Fig. 13, additional objects and complex background resulted in dropping high-quality features on the target leading to poor overall detection rate using threshold-based search. More details regarding these two test images can be found under Table I. This result points to the ability of the resource-aware application to adapt itself to changing load conditions and generate better results in tightly constrained situations.

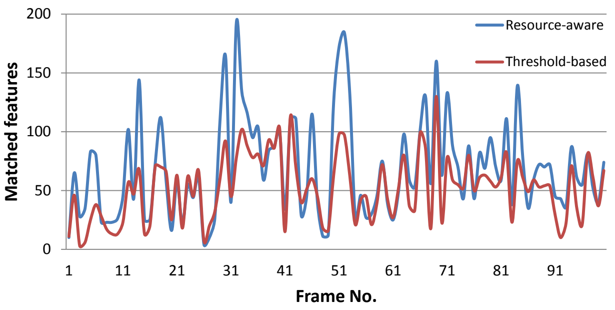


Fig. 11. Comparison between resource-aware and threshold-based NN-search

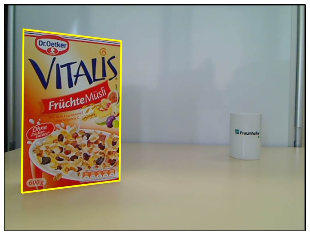


Fig. 12. Image(a): single object

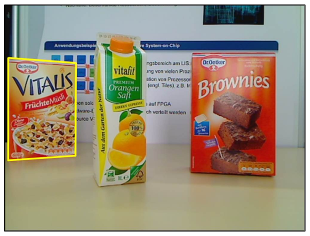


Fig. 13. Image(b): multi-objects

When compared to the threshold-based search, the iterative search algorithm offers improved results in some frames and equal or deteriorated results in other frames. A detailed analysis shows that under circumstances where a large number of features have to be processed using few PEs, the threshold-based model outperforms the iterative model. This is because the iterative model tries to perform a search without dropping any features in a scenario described above and the total number of leaf nodes visited during the search process may drop to very low values (see region(A) in Fig. 5). Hence the overall quality can be too low as the NN-search may result in poorly matched candidates. The iterative model performs well compared to the threshold-based model in other situations as this model can avoid dropping features

TABLE I
COMPARISON BETWEEN RESOURCE-AWARE AND THRESHOLD-BASED NN-SEARCH (FP: FEATURE-POINT, PR:PROCESSED, DR:DROPPED, TR:THRESHOLD-BASED SEARCH, RA:RESOURCE-AWARE SEARCH)

		Image(a)	Image(b)
PE count (alloc / required)	TR	12 / 23	14 / 31
	RA	12 / 23	14 / 31
FPs on object (PR / DR)	TR	177 / 320	**88 / 217**
	RA	320 / 320	217 / 217
FPs on others (PR / DR)	TR	21 / 42	143 / 284
	RA	42 / 42	284 / 284
Total FPs recognized	TR	108	**60**
	RA	111	**115**

using the adaptive techniques described in Section IV-B. A comparison between the resource-aware and the iterative NN-search is provided in Fig. 14, where the resource-aware algorithm outperforms the iterative search in numerous scenarios. The reason for this is the increased complexity within the iterative search resulting from the enhancements described in Section IV-B. The added logic to process the features in an iterative fashion results in higher complexity, leading to higher overall execution time. Moreover, the iterative model has to load every SIFT feature multiple times into the on-chip memory or data cache during the NN-search while the resource-aware model loads each feature once during the entire search process. This means the iterative search algorithm creates a higher load on the external memory and AHB bus, NoC, etc., which reduces the scalability and efficiency during execution. As a result, the iterative model cannot process as many features as the resource-aware model within a fixed interval, reducing the total matched features or quality of the algorithm.

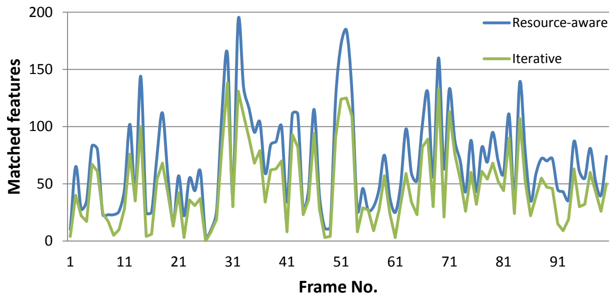


Fig. 14. Comparison between iterative and resource-aware NN-search

VIII. CONCLUSION

This paper presented a resource-aware NN-search algorithm for kd-trees and demonstrated how to estimate the resources required for a specific search operation based on the scene (number of objects present, type of background, etc.) and the texture of the object to be recognized and localized. The application is aware of available resources on the many-core processor and can re-balance the workload if sufficient resources are not available. The ability of the application to bargain for resources and adapt to available resources helps to avoid frame drops and to complete the search process within the specified search interval. Our experiments show that incorporating resource awareness into the conventional NN-search algorithm can improve the quality of the recognition process significantly. A detailed evaluation was conducted on an FPGA-based HW prototype to ensure the validity of the results. Though the evaluations were conducted using the OS and HW explained under Section V, the benefits are expected to be visible on any resource-aware platform including ROS [11]. The newly proposed algorithm is simple and retains all the characteristics of the conventional algorithm. The resource allocation and release happens once per frame and the additional overhead in execution time is negligible when compared to the time taken by NN-search to process thousands of features in every frame.

REFERENCES

[1] T. Asfour, K. Regenstein, P. Azad, et al. ARMAR-III: An integrated humanoid platform for sensory-motor control. In *6th IEEE-RAS International Conference on Humanoid Robots*. IEEE, 2006.
[2] P. Azad, T. Asfour, and R. Dillmann. Combining harris interest points and the sift descriptor for fast scale-invariant object recognition. In *Intelligent Robots and Systems, 2009. IROS 2009*. IEEE, 2009.
[3] J. Beis and D. Lowe. Shape indexing using approximate nearest-neighbour search in high-dimensional spaces. In *IEEE Computer Society Conference on Computer Vision and Pattern Recognition*, 1997.
[4] S. Bell, B. Edwards, J. Amann, et al. Tile64-processor: A 64-core soc with mesh interconnect. In *Solid-State Circuits Conference, 2008. Digest of Technical Papers*, pages 88–598. IEEE, 2008.
[5] R. D. Blumofe, C. F. Joerg, et al. Cilk: An efficient multithreaded runtime system. In *Proceedings of the Fifth ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP '95)*, 1995.
[6] L. Cayton. A nearest neighbor data structure for graphics hardware. *Proceedings of ADMS*, 2010.
[7] L. Cayton. Accelerating nearest neighbor search on manycore systems. In *Parallel & Distributed Processing Symposium (IPDPS), 2012 IEEE 26th International*, pages 402–413. IEEE, 2012.
[8] J. Elseberg, S. Magnenat, R. Siegwart, et al. Comparison of nearest-neighbor-search strategies and implementations for efficient shape registration. *Journal of Software Engineering for Robotics*, 3(1):2–12, 2012.
[9] J. Gaisler and E. Catovic. Multi-core processor based on leon3-ft ip core (leon3-ft-mp). In *DASIA 2006-Data Systems in Aerospace*, volume 630, page 76, 2006.
[10] V. Garcia, E. Debreuve, and M. Barlaud. Fast k nearest neighbor search using gpu. In *Computer Vision and Pattern Recognition Workshops, 2008. CVPRW'08. IEEE Computer Society Conference on*. IEEE, 2008.
[11] K. Klues, B. Rhoden, Y. Zhu, A. Waterman, and E. Brewer. Processes and resource management in a scalable many-core os. *HotPar10, Berkeley, CA*, 2010.
[12] T. Mattson, M. Riepen, et al. The 48-core scc processor: the programmer's view. In *Proceedings of the 2010 ACM/IEEE International Conference for High Performance Computing, Networking, Storage and Analysis*, pages 1–11. IEEE Computer Society, 2010.
[13] B. Oechslein, J. Schedel, J. Kleinöder, L. Bauer, J. Henkel, D. Lohmann, and W. Schröder-Preikschat. Octopos: A parallel operating system for invasive computing. In *Proceedings of the International Workshop on Systems for Future Multi-Core Architectures (SFMA). EuroSys*, 2011.
[14] Synopsys. In *CHIPit Platinum Edition - ASIC, ASSP, SoC Verification Platform*, 2009.
[15] J. Teich, J. Henkel, A. H. andDoris Schmitt-Landsiedel, W. Schröder-Preikschat, and GregorSnelting. Invasive Computing: An Overview. In M. Hübner and J. Becker, editors, *Multiprocessor System-on-Chip – Hardware Design and ToolIntegration*, pages 241–268. Springer, Berlin, Heidelberg, 2011.