INTECH
open science | open minds

OPEN ACCESS
ARTICLE

International Journal of Advanced Robotic Systems

Orthophoto Classification for UGV Path Planning using Heterogeneous Computing

Regular Paper

Robert Hudjakov[1,*] and Mart Tamre[1]

1 Tallinn University of Technology, Tallinn, Estonia
* Corresponding author E-mail: robert.hudjakov@gmail.com

Received 14 Jun 2012; Accepted 12 Apr 2013

DOI: 10.5772/56545

Abstract We have developed a system that uses aerial imagery for improving unmanned ground vehicle navigation capabilities. The current article focuses on the viability of using heterogeneous computing architecture for improving system performance in terms of energy consumption and area analysed per second.

Keywords UGV, Aerial Imagery, Classification, Cost Map, Path Planning, Deep Learning

1. Introduction

The objective of our research is to provide a long-range path planning solution for an off-road unmanned ground vehicle (UGV) by introducing collaboration capabilities between the UGV and an unmanned aerial vehicle (UAV). One of the key issues in off-road UGV long-term path planning is the availability of detailed and up to date maps. While there is no shortage of mobile path planning devices that rely on existing road maps, they will be rendered useless in crisis situations that introduce a significant amount of roadblocks over a wide area such as earthquakes, wildfires, hurricanes or military conflicts. In addition, the availability of detailed off-road maps remains limited due to the changing nature of the environment. Floods, fallen trees or man-made blockades can significantly alter the accessibility of a given terrain, forcing the UGV to rely solely on local navigation capabilities.

We have developed a system that can generate maps from aerial imagery and use them for path planning. The system consists of a classifier, a cost map generator and a path planner; it relies on the availability of overhead imagery and on the environment labelling capabilities of the UGV. We utilize the prior knowledge gathered by the UGV for training a terrain classifier, which is then used on the overhead imagery for feature extraction. The produced feature vectors are used for cost map generation and path planning (Figure 1). As the ground vehicle continues exploration of the terrain and gathers fresh data about the environment we continue training the classifier to accommodate the gathered data and use it for updating the path to target objective.

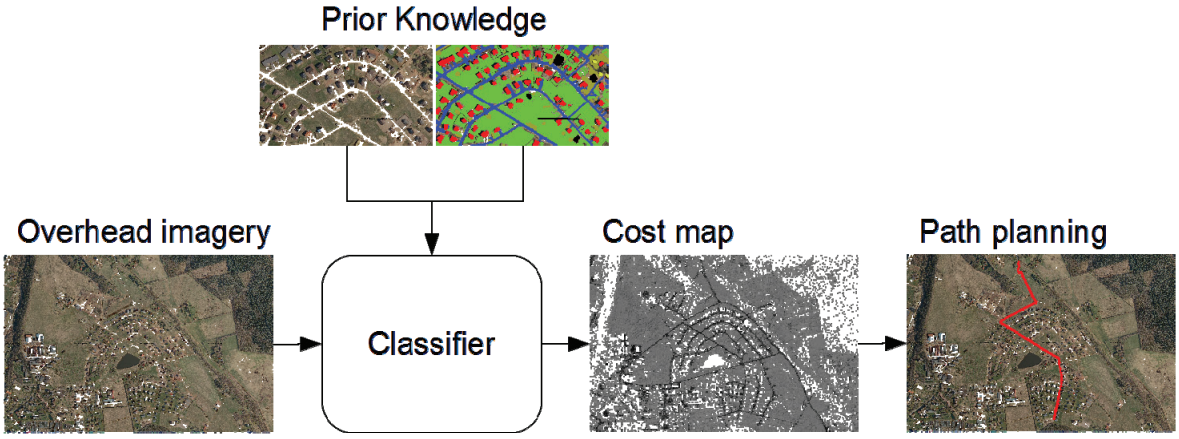

Figure 1. Prior knowledge is used to train an aerial image classifier. The classifier will then be used for large overhead image processing and its output is used for cost map generation and path planning.

The aerial image classification step is computationally intensive, requiring a high performance computer to run for extended periods at maximum power consumption levels. For a battery powered UGV energy efficiency is an important factor [1] and supporting a high performance PC can severely limit its mission capability. To reduce the energy consumption of our system we turned to heterogeneous computing platforms, namely OpenCL (Open Computing Language).

The heterogeneous computing platform supports two programming paradigms: task parallel algorithms and data parallel algorithms. The data parallel approach relies on finding and exploiting parallelism within a function, while the task parallel approach executes multiple instances of the function on multiple inputs. Previous studies [2][3] have focused on data parallel algorithms, but we also explored the task parallel algorithm.

2. The System

To classify a large overhead image we divide it into smaller patterns with a size of 29x29 pixels; for better output resolution we use overlapping patterns. As the UGV navigates through terrain it will gather explicit knowledge about some of these patterns, which are used to compose a training set. The training set will be used to teach the classifier, which will then be used to extrapolate the gathered knowledge to uncategorized patterns.

Our classifier inputs are 29x29 pixel patterns and the outputs are feature vectors; each element in a feature vector represents the likelihood that a corresponding feature is present in the input pattern. The classifier is trained to detect explicit features (houses, roads, trees, etc.) in input patterns; it can be trained to detect all the specified features in the input pattern or to categorize just one pixel of the input pattern [4]. In the first case the output vector elements are independent, for example there can be a house, road and a tree in an input pattern. In the second case the elements are dependent; the specific pixel can only belong to one category be it tree, road or a house.

Our proposed orthophoto classifier is a deep convolutional artificial neural network; it is a modification of the one used by LeCun et al. [5] for handwritten digit classification, that was later simplified by Simard et al. [6]. The convolutional neural networks are especially suitable for image processing because they utilize a key property of images - nearby pixels are more strongly correlated than distant pixels [7].

The network structure [8, 9] (Figure 2) is optimized to detect local features on a small subregion of an input pattern and then combine them in the later stages of processing to detect higher order features. Convolutional neural networks are invariant to shift, rotation and to some extent scale transformations (due to weight sharing and subsampling) [7] and to colour intensity values [5]. In addition, because of the shared weights, the variable space of the convolutional neural network is smaller than that of a fully connected network; it requires fewer samples to train the network.

The reference convolutional neural network that we have included in all our experiments consists of five layers (Figure 2): the input layer contains three or four 29x29 neuron feature maps – one per input pattern colour channel. The second layer contains six 13x13 neuron feature maps, which are connected to the previous layer using 5x5 neuron kernels. The third layer contains 50 5x5 neuron feature maps which are again connected to the second layer using 5x5 kernels. The fourth layer is a linear classifier which contains 100 neurons and the final layer contains one neuron per feature category.

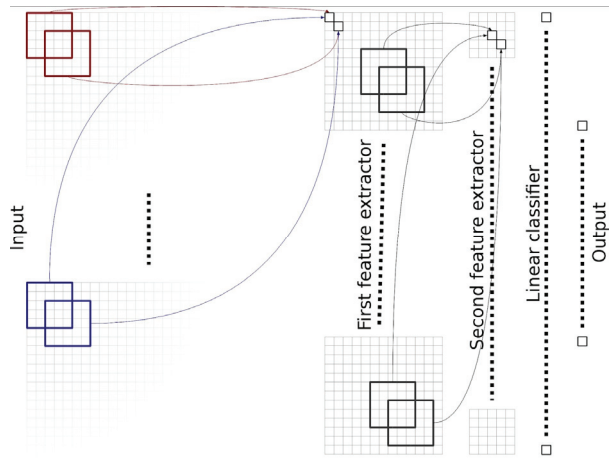

Figure 2. The classifier setup

The feature vectors produced by the classifier for each pattern are used for cost map generation. Before the cost is calculated we convert the feature vectors into probability vectors, where each vector element represents the likelihood that a corresponding feature is present in an input pattern. The classifier outputs are trained to be 1.0 when a feature is present in the input pattern and -1.0 when it is not. The network outputs, however, are rarely 1.0 or -1.0, usually they are somewhere in the range of -1.7159 to 1.7159 (constrained by the chosen sigmoid function). To decide if the feature is present or not depending on the network output we need a threshold, a decision boundary, i.e., if the network output is above the boundary we consider the corresponding feature to be present. To create the probability vector we map the classifier outputs so that -1 equals 0% probability, the threshold equals 50% probability and +1 equals 100% probability (Figure 3).

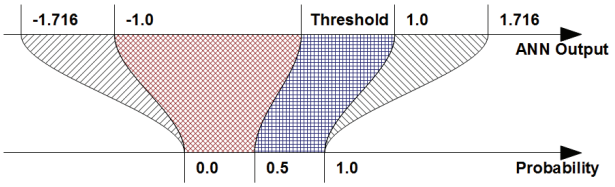

Figure 3. Mapping a classifier output to probability

The algorithm that we use to find a decision boundary for each classifier output above, for which we assume the corresponding feature is present in the input pattern, is best described graphically. First we evaluate all the patterns in the training set and plot two graphs for each classifier output - one graph shows the probability density function of the classifier output when a corresponding feature is known to be present in the input pattern and the other shows the probability density function of the classifier output when the feature is known not to be present (solid lines on Figure 4). In addition we plot the cumulative and reverse cumulative versions of the density functions (dashed lines on Figure 4) and use the crossing point of those cumulative density functions as the output threshold or decision boundary.

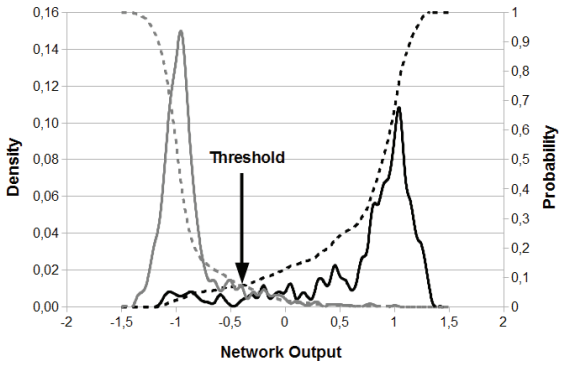

Figure 4. Finding the decision boundary for a single classifier output.

For cost map generation we assign a weight to each feature class such that zero corresponds to unknown terrain, negative weights are assigned to easily passable terrains (roads, grass) and positive weights are assigned to impassable terrains such as buildings. The transit cost of a pattern is a dot product between the pattern feature vector and the weight vector that is offset by a constant that is equal to the cost of unknown terrain:

$$\text{cost} = \mathbf{O} \cdot \mathbf{W} + \mathbf{C}_{\text{unknown}},$$

where
O is the probability vector,
W is the cost vector,
$\mathbf{C}_{\text{unknown}}$ is the cost of unknown terrain.

When our classifier can't detect any features in the input pattern or the probability of features is very low, the transit cost will be equal or near equal to the cost of unknown terrain. When we detect only easily traversable features in the input pattern (roads, grass) the cost will be lower than the cost of unknown terrain. When only impassable features are detected (houses, trees) the cost will be higher than the cost of unknown terrain. In the case where both houses and roads are detected the cost value depends on the chosen weight values (Figure 5).

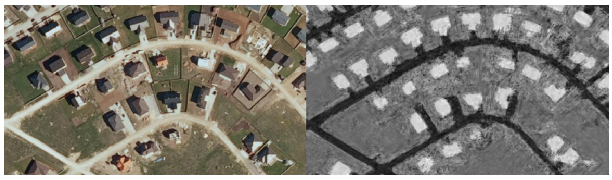

Figure 5. A sample image of suburban terrain (left) and its cost map (right). Dark colours on the cost map correspond to easily traversable areas (roads) and light colours are obstacles (buildings)

3. Heterogeneous Computing

Heterogeneous computing addresses the performance limitations of single threaded applications by making use of a variety of different computing units. The last 40 years of steady processor and compiler development has focused almost exclusively on serial workloads, giving software developers a "free ride"; a program written 30 years ago has received a continuous stream of performance boosts from increasing clock frequencies and improved processor architectures. During the last decade the development of serial workloads has hit a set of walls; namely a power wall, a memory wall and an instruction level parallelism wall [10]. It is no longer reasonable to expect the serial computing workloads that are prohibitively slow today to be exponentially faster in five years time.

Modern processors make use of a host of techniques such as pipelining, speculative execution and superscalar execution that look for instruction level parallelism in order to increase single threaded workload performance. In modern x86 general-purpose processors more die space is dedicated to scheduling of an instruction than to executing the instruction, therefore more power is spent on scheduling said instruction than on executing it. The solution is good for executing serial workloads but suboptimal for applications where a single function needs to be executed on a massively parallel scale - much more computing power can be extracted from the same amount of silicon when optimizing for parallel workloads.

The last decade has seen the emergence of affordable hardware that is dedicated to parallel processing and is driven by the thriving computer gaming market. Initially the relatively simple geometric calculations needed by computer graphics were carried out by fixed function hardware, but the need for better graphics has led to the introduction of programmable hardware. Continuous competitive pressure to produce faster graphics hardware has led to unifying the different programmable steps in a fixed function pipeline into general purpose processing units that can emulate the whole graphics pipeline and do a lot more.

The graphics processing units (GPUs) are optimized for solving massively parallel mathematical problems and as such they dedicate most of the die area to computational units rather than instruction schedulers. Instead of using high clock rates, which are exponentially linked to power draw, they make use of larger dies that can fit more computational units. Instead of using large on die caches they rely on higher memory bandwidth. Modern GPUs support thousands of simple processors that are optimized for executing a function in a parallel and power efficient manner.

Heterogeneous computing emphasizes taking advantage of the different computing units found in a device. For our system it is better to make use of the fixed function texture units that are found in GPUs for parsing the orthonormalized input image, utilize the parallel computing capabilities of GPUs for training and executing the classifier and use the CPUs for a single threaded path planning task. We found evaluation of a large overhead image, consisting of hundreds of thousands, if not millions, of patterns, to be especially suitable for GPUs.

4. Implementation Details

There are multiple application programming interfaces (APIs) available for utilizing the computing resources provided by GPUs and parallel computing architectures in general, such as nVidia CUDA, AMD Stream and Microsoft DirectCompute. We chose to use vendor-independent OpenCL, which is supported by all major GPU and CPU vendors.

An important drawback of using parallel architectures is that they cannot be used to execute existing application codes. To utilize the OpenCL infrastructure a separate application has to be written and the performance critical parts of the application have to be gathered into special functions called "kernels" that are written in a fork of C99 programming language. In order to reap performance benefits the kernels must be aware of multiple limitations imposed by hardware such as count of registers per thread, size of private memory, shared memory, constant memory and global memory, memory access patterns, number of processing units per group, etc. Unconventional programming models and the need to micromanage every aspect of the kernel execution makes the testing and optimization process very slow and time-consuming, prohibiting wider use of heterogeneous platforms.

The computing capacity provided by OpenCL can be utilized in two distinct ways: task parallel algorithms and data parallel algorithms.

1. The task parallel algorithms will execute multiple serial workloads in parallel. For example, we can run multiple instances of our classifier to evaluate a large set of inputs at once. Every processor in the GPU will then execute a classifier with a different set of inputs.
2. The data parallel algorithms break the complex problem into simpler pieces that can be executed in a parallel manner. For example, we can break our classifier into layers, each containing a set of neurons that are connected to neurons in the previous layer. For the data parallel approach we can execute each layer in a data parallel manner by dividing the workload between processors (the output of each neuron could be calculated on a separate processor).

For comparison we implemented our classifier using both approaches. To reduce memory copies from the host memory to the OpenCL device memory and to reduce the device memory usage we loaded the whole overhead image into the device memory before executing kernels. The extraction of relevant patterns from overhead imagery and the initialization of the classifier input layer is done by kernels using fixed function texture processing hardware. In addition, we pre-allocated a device side memory buffer large enough to contain classifier outputs for all the patterns in a batch - effectively eliminating the host-to-device and device-to-host memory copying bottlenecks for large batches (10000+ patterns).

The data parallel classifier algorithm requires complete rethinking of how the classifier is evaluated; we chose to implement it using five steps. We used 29x29=841 threads to initialize the input layer of the classifier. Each thread is responsible for extracting a single pixel from the input pattern and copying the colour channel values to designated memory buffers.

The first hidden layer of our classifier has 13x13x6=1014 neurons and each neuron is connected to all three previous layer feature maps using 25 weights (5x5 convolution kernel + 1 bias connection). Theoretically it would be efficient to dedicate 1014x3x26=79092 threads to calculating all the weight connections of every neuron in parallel and to use parallel reducing steps to evaluate neuron outputs, but the semi-random nature of how convolutional layer neurons are connected to previous layers would destroy coalesced memory access and kill the performance of the GPU. We found that it is better to use just 1014 threads (one per neuron) on the convolutional layer both on the CPU and the GPU and sum the connections with a simple FOR loop. The same limitations affect the second convolutional layer, so we dispatch 5x5x50=1250 threads to evaluate the layer.

The last two layers of our classifier are fully connected layers, meaning that each neuron in a given layer is connected to all the neurons in the previous layer with a unique (non shared) weight. The third hidden layer has 100 neurons that all are connected to 1250 neurons on the second convolutional layer. In order to evaluate the layer we dispatch 100 groups of 512 threads and each group is responsible for evaluating a neuron in the layer. For each neuron we need to calculate the dot product of the previous layer output vector and weight vector - both have a length of 1250 (excluding bias) and reside in the global memory - using 512 threads. We chose to use 512 threads as it means that each thread must only fetch three neuron outputs and three weights from the slow global memory and store the dot products of the three element vectors in the fast shared memory buffer. The 512 elements in the shared memory are then reduced to single values using two additional steps and the network outputs are calculated.

In comparison to the data parallel algorithm the task parallel algorithm is a straightforward serial algorithm that uses a single execution thread. The programming of the task parallel algorithm is very reminiscent of embedded programming - there are lots of hardware limitations that have to be avoided but coding is straightforward. When compared to our reference implementation of the classifier (written in C#), the OpenCL code is very low level and rigid - the specific classifier configuration (how many layers, count of neurons per layer, etc.) needs to be defined at the compile time, all buffers need to be preallocated and all relations between neurons and weights have to be hardcoded.

5. Experiences

The primary performance bottleneck in our application is the classifier followed by the path planner; classification of a large overhead image is particularly slow. There are two usage scenarios for the classifier:

1. For training we feed training patterns into the classifier one by one, propagate the neural network and compare the classifier output against a known value to find the output error. The output error is then back propagated through the network and used for adjusting the network weights.
2. For path planning we need to re-evaluate overhead imagery and to generate an updated path to target objective.

Since the training is done one pattern at a time we need a method that can process a single pattern in minimal time. From the comparison of single pattern evaluation times (Table 1) we can see that the fastest algorithm is the reference implementation written in high level language, mostly because it does not include data transfers from the host memory to the OpenCL device memory and back. The data transfers can be eliminated when both evaluation and training is done on the device, but unfortunately we don't have an OpenCL optimized training algorithm available to directly compare the OpenCL learning cycle against a reference implementation learning cycle. We do, however, have a suboptimal data parallel OpenCL solution that includes both evaluation and training steps and from there we can see that training step is about 2.2 times slower than the evaluation step in the data parallel kernel. Given that our peak data parallel kernel execution rate on a single GPU device is about 12000 patterns/s we should be able to achieve a learning rate of about 5500 patterns/s, which is about 7x faster than CPU reference implementation. The

predicted learning rate is comparable to what others have achieved [2, 3].

System configuration	Latency [ms]
Dual GTX 285 with data parallel kernel	20.3
Dual GTX 285 with task parallel kernel	211
Dual GTX 570 with data parallel kernel	3.4
Dual GTX 570 with task parallel kernel	51
i7-920 CPU with data parallel kernel	12
i7-920 CPU with task parallel kernel	3.7
i7-920 CPU with reference implementation	1.3

Table 1. Time it takes to evaluate a single input pattern using different methods and architectures.

An overhead image that covers one square kilometre of terrain with a resolution of 0.5m/pixel has a size of 2000x2000 pixels. To generate a cost map that has an equal resolution we need to analyse nearly four million overlapping patterns. For classification of the image the task parallel approach is preferred as it has the highest throughput (Figure 6). The high throughput of the task parallel kernel can be attributed to memory access efficiency - when there are enough parallel threads scheduled for a GPU processing unit the memory latencies will be effectively hidden. If evaluation of a pattern on a processing unit is blocked by memory access, the processing unit moves on with the evaluation of other patterns and resumes the evaluation of the pattern later, when the memory operation is completed.

For evaluating the four million patterns on an i7-920 CPU with reference classifier implementation that had an evaluation rate of about 800 patterns per second we needed 5000 seconds. Given that the max power consumption of an i7-920 processor is 130W and our reference implementation pushes all cores to nearly 100% usage, it takes 650kJ of energy to evaluate the image. The max throughput of a dual GTX570 GPUs is 90600 patterns per second, so we only need 44 seconds to evaluate the same image. Even though the two GPUs consume up to 440W, it only takes 19.3kJ to evaluate the image. By implementing the classifier of the GPU we have gained over 100x pattern throughput and over 30x reduced power consumption. The power gains are even more significant if we include the power consumption of the whole system.

It must be noted, however, that for path planning we don't need a cost map that has a resolution identical to the overhead image resolution. In our experience it is sufficient when both the horizontal and vertical resolution of the cost map is 3 to 10x smaller than the input image. By reducing the amount of patterns to be evaluated in a batch by 9 to 100x, the evaluation time of the batch reduces proportionally. We do, however, need to evaluate more than one square kilometre during a mission and we plan to re-evaluate the overhead imagery as fresh images and training data becomes available, bringing the total amount of patterns to be evaluated during a mission back to millions.

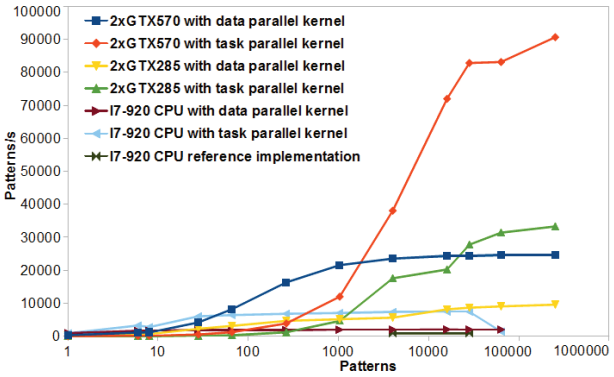

Figure 6. Throughput of the classifier using different methods and architectures.

The OpenCL solution is 10x faster than our highly optimized reference solution using the same hardware (Figure 6), largely because the OpenCL data structures are explicitly laid out, avoiding indirect object field accessors in favour of direct array access, reducing memory accesses and improving cache efficiency. In addition, the OpenCL solution avoids safety features such as memory bounds checks. The drawback is reduced flexibility; the OpenCL classifier configuration has to be specified at the compile time, making it difficult to find the best network configuration for a task.

6. Summary and Future Work

The two advantages of using a heterogeneous computing platform are dramatically improved performance and reduced energy requirements. By running the slowest part of our system - classification of a large overhead image - on GPUs we gained over 100x performance in terms of patterns evaluated per second and reduced energy requirement of the system by more than 30x when compared to CPU implementation. We found the task parallel programming approach to be more than 3x more efficient than the data parallel approach when processing large datasets.

7. References

[1] M. Hiiemaa and M. Tamre, "Intelligent Energy Management of Unmanned Ground Vehicle" *7th International Conference of DAAAM Baltic Industrial Engineering*, Tallinn, 2010.

[2] D. Strigl, K. Kofler and S. Podlipnig, "Performance and Scalability of GPU-Based Convolutional Neural Networks" 2010.

[3] K. Chellapilla, S. Puri and P. Simard, "High Performance Convolutional Neural Networks for Document Processing" 2006.

[4] R. Hudjakov and M. Tamre, "Aerial Imagery Based Long-Range Path Planning for Unmanned Ground Vehicle" *7th International Conf. Mechatronics Systems and Materials*, Kaunas, Lithuania, 2011.
[5] Y. LeCun, L. Bottou, Y. Bengio and P. Haffner, "Gradient-Based Learning Applied to Document Recognition" *Proceedings of the IEEE*, 1998.
[6] P. Y. Simard, D. Steinkraus and J. C. Platt, "Best practices for convolutional neural networks applied to visual document analysis" 2003.
[7] C. M. Bishop, "Pattern recognition and machine learning" Springer, 2006.
[8] R. Hudjakov and M. Tamre, "Aerial Imagery Terrain Classification for Long-Range Autonomous Navigation" *Proc. of International Symposium on Optomechatronic Technologies*, Istanbul, 2009.
[9] M. T. R. Hudjakov, "Ortophoto analysis for UGV long-range autonomous navigation" *Estonian Journal of Engineering*, Vol. 17, No. 1, pp. 17 - 27, 2011.
[10] P. N. Glasowsky, "NVIDIA's Fermi: the first complete GPU computing architecture" 2009.