Mapping parallel programs to heterogeneous CPU/GPU architectures using a Monte Carlo Tree Search

Goli, Mehdi; McCall, John; Brown, Christopher; Janjic, Vladimir; Hammond, Kevin

Published in:
2013 IEEE Congress on Evolutionary Computation, CEC 2013

DOI:
10.1109/CEC.2013.6557926

Publication date:
2013

Document Version
Peer reviewed version

Mapping Parallel Programs to Heterogeneous CPU/GPU Architectures using a Monte Carlo Tree Search

Mehdi Goli
and John McCall
School of Computing
Robert Gordon University
Scotland, UK
Email: {m.goli, j.mccall}@rgu.ac.uk

Christopher Brown,
Vladimir Janjic
and Kevin Hammond
St Andrews University,
Scotland, UK
Email: {cmb21, vj32, kh}@st-andrews.ac.uk

***Keywords*—Montecarlo Tree Search; Heuristic Algorithm; Static Mapping; Parallel Programming; Heterogeneous Architecture;**

***Abstract*—The single core processor, which has dominated for over 30 years, is now obsolete with recent trends increasing towards parallel systems, demanding a huge shift in programming techniques and practices. Moreover, we are rapidly moving towards an age where almost all programming will be targeting parallel systems. Parallel hardware is rapidly evolving, with large *heterogeneous* systems, typically comprising a mixture of CPUs and GPUs, becoming the mainstream. Additionally, with this increasing heterogeneity comes increasing complexity: not only does the programmer have to worry about where and how to express the parallelism, they must also express an efficient mapping of resources to the available system. This generally requires in-depth expert knowledge that most application programmers do not have. In this paper we describe a new technique that derives, *automatically*, optimal mappings for an application onto a heterogeneous architecture, using a Monte Carlo Tree Search algorithm. Our technique exploits high-level design patterns, targeting a set of well-specified parallel *skeletons*. We demonstrate that our MCTS on a convolution example obtained on average 95% of the speed up achieved by the hand tune optimisation developed by expert user.**

I. Introduction

Ongoing rapid progress in technologies such as GPGPU and CPU/GPU clusters and emerging platforms such as CUDA and OpenCL are widening applicability of high performance computing. Data and stream-based parallelism in particular can now be brought to bear on programming tasks in many engineering and manufacturing operations. However, the process of synchronisation and communication among components in a given parallel architecture, essentially to obtain maximal performance gains, is still a challenging issue and requires expert parallel programmers. In this paper we take an approach that exploits high-level parallel patterns, known as *Algorithmic Skeletons*, which are a high-level abstraction for parallelising applications [1]. They have long been considered as a viable approach to encapsulating coordination, communication and synchronisation among components. [2]

Stream-based parallelism can be regarded as a generic form of pipeline in which the parallel program operates repeatedly on a stream of data items, e.g. an audio or video stream. Typically a parallel task performs in a way that on each execution step it reads one or more data items from one or more input streams, processes the input data and passes the processed data to one or more output streams. [3]

The problem of mapping a stream-based parallel program to a given parallel processing architecture can generally be divided into two stages:

- partitioning the program into different tightly encapsulated computational components and allocating each to a thread. This is a design problem for the programmer, possibly assisted by the software designer.
- scheduling the program to run on a specific architecture in terms of allocating an optimum number of processors per component and dedicating appropriate flows of input data to each component.

Separating these two tasks from each other to some extent provides a level of abstraction from underlying resource which increases the portability of parallel programs, because there is no need for reshaping the whole written program for each targeted platform. Once the skeletal structure has been generated, the mapping can be retuned for different specific targets. The main focus of this paper is to develop an intelligent algorithm to automatically map a given program structure to a given heterogeneous architecture to optimise performance.

In a homogeneous architecture, for each processing node in the partition graph there is one type of component. Multiple instances of that component can be run as a thread over multi-core systems in which each core has the same power. However, this is not the case for heterogeneous (CPU/GPU) architecture. There it is desirable to take advantage of GPU processing for suitable components when possible. Thus some components can be written to run on either GPU or CPU or a combinations of both. In this case, finding the optimal mapping choice becomes more complex. The throughput of each processing node depends on selecting either one version of the component or using both of them, depending on the limitations of the underlying architecture and the computational costs of the components. Also, where there is a choice of running a component on CPU or GPU, the proportion of flow of the

input stream sent to each of them will affect the overall throughput of the system. Moreover, there is a constraint on the number of tasks that can be run simultaneously on a single GPU. Therefore, deciding which nodes to assign to a GPU is combinatorially complex when there are multiple nodes that can be mapped as such. Taking all of these factors into account, the problem of mapping a parallel program to a heterogeneous architecture becomes considerably more complex than that of mapping to a homogeneous architecture.

Static mapping involves calculating the proportion of work and resources needed to be allocated for each component of the parallel program. The problem associated with static mapping is then finding the optimal mappings from the set of all possible static mappings. Due to the combinatorial nature of the mapping problem, the solution space grows quickly, indicating suitability of a heuristic approach.

In this paper, we investigate the applicability of Monte Carlo Tree Search (MCTS) for the mapping problem in the heterogeneous (CPU/GPU) architectures. Monte Carlo Tree Search (MCTS) is an approach originating from AI Games to efficiently select moves in large game trees where full evaluation of the tree is computationally intractable. MCTS uses Monte Carlo random walks to evaluate states only at the endpoints of each walk [4]. Here we create mappings by random-walks where at each step a decision is made to allocate a particular package of processing to a particular processor. The walk is biased by the parameters of the static mapping. We use a simulation of the computation to evaluate each mapping at the end of the walk. Objectives such as throughput can be calculated through sampling and summing the simulated function performance distributions on the assigned processors.

The remainder of this paper is structured as follows. In section III we explain the preliminary definitions and concepts. In section IV our proposed MCTS algorithm for mapping problem is explained in detail. We demonstrate the applicability of our proposed algorithm for a real world application as a case study in section V. In sectionVI we analyse the suitability of MCTS for the static mapping problem. SectionII summarises the current research in this area and finally, section VII provides some concluding remarks and future work.

II. Related work

A. Mapping process

The static mapping problem is by no means a new challenge and there is an extensive body of work on mapping task, data and pipeline parallelism to parallel architectures providing static partitioning [5], [6], [7], using runtime scheduling [8], heuristic based mappings [9], analytical models [10], [11], or ILP solvers [12]. Each of these can improve the performance of the system. There are some heuristic based approaches which automate the process of mapping to multi-core architectures for specific frameworks, such as the learning approach used for partitioning streaming in the StreamIt framework [13] or the runtime adaptation approach used in FlexStream [14] framework. Also, a series of research works have been produced aiming at optimising the use of resources on multi-core embedded platforms linking both design-time optimisation and simulation with run-time optimisation using lightweight heuristics [15], [16], [17]. In [18], the use of platform simulators has been considered to identify Pareto-optimal design configurations of parallel applications (incorporating code versions, resource mappings, constraints and costs).

Despite the amount of work done in the homogeneous environment, to our best knowledge there is little work done for mapping to heterogeneous(CPU/GPU) architectures. Most of the work on GPUs are primarily focused on application performance tuning [19] rather than orchestration.

However, in [20] a method is provided to orchestrate the execution of heterogeneous StreamIt program presented on a multi core platform equipped with an accelerator. They use integer linear programming (ILP) formulations to perform partitioning over the combination of CPU/GPU. Formulating ILP models, however, requires expert knowledge of the underlying architecture. Finding a solution under certain constraints for the ILP formula can be time-consuming as well.

Our aim in this paper is automatic orchestration of CPU/GPU component for FastFlow applications by using MCTS which usually requires less knowledge about the system. Therefore, minimum information is needed to run the result on the system. Moreover, applying it in a static manner has a much lower overhead for deployment.

B. Monte Carlo Tree Search

Monte Carlo Tree Search have classically been applied to challenging game playing, for example the GO and Bandit problem [21], [22], [23], [24].

Recently MCTS has been applied to planning and scheduling problems. In [25] a MonteCarlo search algorithm has been applied to produce management problems which can be dened as single-agents selecting a sequence of actions with side effects, leading to high quantities of one or more goal products.The result shows that they achieve a successful solution in less time than already existing learning method. In [26] a Monte Carlo Random Walk (MRW) planning algorithm is used in deterministic classical planning achieving results comparable with the other state of the art algorithms. In [27] an extended version of UCT approach has been successfully applied in continuous stochastic problems with continuous action space.

In this paper we establish the applicability of MCTS to the seamless orchestration of heterogeneous components over a hybrid(CPU, GPU) platform. Although we work on the FastFlow framework, the technique can easily be applied to a different framework by changing the evaluation method used here, which makes the algorithm framework independent.

III. Background

A. FastFlow

Fastflow [28] is a skeleton-based parallel programming framework for multi-core platforms, implemented in C++. Fastflow's streaming patterns are coordinating mechanisms that control the flow of work between multiple concurrent threads. This allows programmers to focus on application-specific computational components by abstracting over complex coordination and communication layers. Fastflow supports

both CPUs and GPUs as part of a heterogeneous parallel system [29].

B. Skeletons

An *algorithmic skeleton* is an abstract computational entity that models some common pattern of parallelism (such as the parallel execution of the sequence of computations over the set of inputs, where the output of one computation is the input to the next one). A skeleton is typically implemented as a high-level function that takes care of the parallel aspects of a computation (e.g., the creation of parallel threads, communication and synchronisation between these threads, load balancing etc.), and where the user provides a sequential code specific to his concrete problem.

In this paper, we restrict ourselves to two fundamental, heterogeneous skeletons, that we consider to be the most popular and most useful:

- The *Pipeline* skeleton models the application of a composition of functions $f_1, f_2, \ldots, f_n$ to a sequence of independent inputs $x_1, x_2, \ldots, x_m$, where the output of f_i is the input to f_{i+1}. The parallelism arises from the fact that, for example, $f_1(x_n)$ can be executed in parallel with $f_2(f_1(x_{n-1}))$. We will denote the pipeline skeleton by $Pipe(f_1, f_2, \ldots, f_n, x)$, where $f_1, f_2, \ldots, f_n$ are the functions of a pipeline and x is the array of inputs.
- A *Farm* skeleton models the application of a function, f, to a sequence of independent inputs, $x_1, x_2, x_3, \ldots, x_n$. We will denote the farm skeleton by $Farm(nw, f, x)$, where nw is the number of worker threads, f is a function that is to be applied to the inputs, and x is the array of inputs.

C. Monte Carlo Tree Search

MCTS uses four main steps. (i) in the *selection* step the tree is traversed from the root node to the leaves; (ii) next, in the *expansion* step, a new node is added to the tree; (iii) subsequently, during the *sampling* step, moves are selected at random until a leaf is reached and evaluated; and (iv) finally, in the *back-propagation* step, the result of an evaluation is propagated backwards, through the previously traversed nodes. [30]

IV. The Proposed Approach

We assume that we are given a parallel program, together with its associated *skeleton tree structure*. A skeleton tree structure provides the high-level view of the skeletons used in a program, together with the information about their nesting and the sequential *components* used in these skeletons. We also assume that we are given, for each component, the information about whether it is a CPU or GPU component, and the estimated runtime of that component for one input on a given hardware. Figure 1 shows an example skeleton tree structure, with three components (A, B, G, D). The components A, B and C are CPU components, whereas the component G is a GPU component. The components B and G belong to the same farm (i.e. they correspond to two implementations of the same function, one for CPUs and one for GPUs).

We define a *static mapping* of a parallel program to a given hardware as an assignment of the amount of appropriate resources (CPUs or GPUs) to each component in the program skeleton tree structure. The *static mapping problem* is the problem of finding, from the set of all static mappings, the static mapping which will be optimal with respect to some metric. In order to find the optimal static mapping, we use the MCTS algorithm to calculate the distribution of computational resources to the components.

The problem of finding an optimal static mapping for the given program is then divided into two phases:

- Converting the program's skeleton tree structure to its associated *process graph*
- A Monte Carlo Tree Search for selecting an optimal mapping of the process graph to the target architecture

A. Converting the Skeleton Tree Structure to the Process Graph

The skeleton tree is a nested combination of farms and pipelines which is created in an IDE interactively by the user. Figure 1 shows the skeleton tree structure of a 3-stage pipeline. Each stage is a farm and the second stage contains a combination of CPU and GPU components.

The skeleton tree structure is converted into a first order logic formula from which a process graph can be created. In case the computational resources available for a particular factorisation are not sufficient, the process graph is reshaped by removing the leaves for which the formula remains correct after their removal. Any skeleton tree can be converted to a first order logic formula, using the following scheme:

$$Farm(A_1, ..., A_n) := \vee_{i=1}^{n} A_i$$

$$Pipe(A_1, ..., A_m) := \wedge_{i=1}^{m} A_i$$

$$A_i := Pipe(A_1^{'}, ..., A_m^{'}) | Farm(A_1^{'}, ..., A_n^{'}) | f$$

where $f \epsilon \{\text{set of components}\}$.

As an example, for the skeleton tree structure presented in figure 1, the first order logic formula is as follows. $FOL(TS) = (A \wedge (B \vee G) \wedge C)$.

To create a process graph from the formula, each preposition represents a component, each operator $\wedge$ represents a queue connecting the components of the systems together and the operator $\vee$ is used to represent different components of the system streaming data from the same queue. Figure 2 shows the process graph generated from $FOL(TS)$. The process graph has three *task queues* for the top-level pipeline. Queues Q_0 and Q_1 connect the subsequent pipeline stages, and the queue Q_2 accumulates the output of the whole pipeline. The task queues are the main factor for finding the optimal solution. For each task queue a queue level has been considered, where the queue level is the number of tasks waiting to be processed by the next stage. In an ideal case the queue level for all task queues is equal to zero, because as soon as a tasks enter in the task queue, it will be processed by the next stage. However, this does not occur in reality specially when we have heterogeneous componenet running on heterogeneous resources. In this case the number of resouces alocated to each component are the key factor to create that queue level balance.

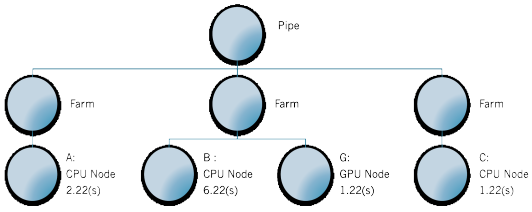

Fig. 1. Tree Structure.

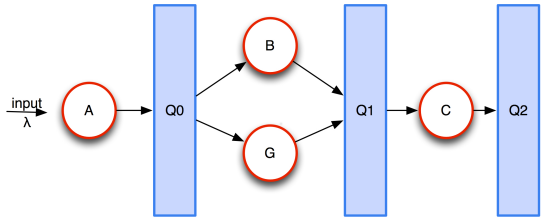

Fig. 2. Process Graph.

The more the queue levels of task queues in the system are similar and close to zero, the better the provided mapping is. The queue level together with the troughput of the system, which can be achieved by monitiring the number of tasks put in last task queue in time interval T, are the main evaluation factors for choosing optimised mapping. Therefore, in each step the MCTS approach, selects a mapping soluion, simulates the execution of the problem for the selected mapping and rewards it by analising the simulation based on the above main factors. The result of the reward will be backpropagated based on the applied backpropagation policy.

In the following we explain the proposed MCTS approach in more details.

B. A Monte Carlo Tree Search from the Process Graph

C. Decision Tree

The nodes of the decision tree of our model are the mapping decisions, where a single decision corresponds to allocating the amount of remaining resources to one or more components. At each decision point, some resources will be allocated, limiting the total resources available to the yet unallocated components. A leaf of the tree corresponds to the complete static mapping, where all components have been allocated an amount of resources. At this point the mapping can be evaluated by simulation, where we apply MCTS to search for an optimal path through this decision tree.

At each step, a set of resources is allocated to components of a single farm. We can represent the set of possible decisions as:

$$\{A(w_i, k) \mid w_i \in M, k \in \{0, 1, \ldots, L\}\},$$

where $A \in \{\text{ADD}, \text{REMOVE}\}$; M is the set of components in the farm (containing one element if there is only a CPU or a GPU component, or two if there are both a CPU and a GPU component); k is the amount of resources to be allocated or deallocated to a component, w_i; and, L is the maximum amount of resources that can be allocated to the component, w_i.

D. Selection Strategy

The selection strategy that we use is the *Upper Confidence bounds applied to Trees* (UCT) [31], [22]. The formula for UCT is:

$$UCT = \overline{X_j} + 2C_P\sqrt{\frac{2\ln n}{n_j}}$$

where n is the number of times the current node has been visited; n_j is the number of times the child, j, has been visited; $C_P > 0$ is a constant value; and, $\overline{X_j}$ is the average reward value given to child node, j.

E. Simulation of Static Mapping

There are two ways to evaluate the performance of a selected path in the decision tree (which correspond to one static mapping): mathematical abstraction of the system (such as a cost model) and simulation. Here we use simulation because the accuracy of a cost model depends on large number of parameters that may be difficult to obtain for certain skeletons and hardware systems. Here we chose an approach that is intended to be generic, rather than completely precise. However, the precision gained using simulations is good enough for comparing different static mappings allowing us to demonstrate the principles of our technique.

We have developed a simulator for FastFlow that mimics the behaviour of a given FastFlow application running on the target architecture. The simulator outputs the metrics (queue Level, component utilisation, throughput) that we use to evaluate the static mappings.

F. Reward Function

Once a static mapping has been simulated, a reward for the mapping is calculated. The reward function is based on the throughput of the system, denoted by T. There are two balancing factors related to the overall utilisation of the system:

1) We define *utilisation* of a component, w, to be denoted by $U(w)$, as the utilisation of the resources allocated to the component, w. We denote by SD_U the standard deviation of the component utilisation from the mean utilisation of all components in the system:

$$SD_U = \sqrt{\frac{\Sigma_N (U_{w_i} - Uw_m)^2}{N}},$$

where N is the total number of components in the system; U_{w_i} is the utilisation of the component, w_i; and, U_{w_m} is the average utilisation of all components in the system. Using SD_U as a reward function discourages the allocation of additional resources to a component in the case where this results in only a minor gain in the overall program speedup (and in reduced utilisation).

2) We define *throughput of a queue* as
SD_Q is the standard deviation of all queues in the system, defined as follows.

$SD_Q = \sqrt{\frac{\Sigma_L (T_{Q_i} - T_{Q_m})^2}{L}}$
Here, L is the total number of queues in the system, T_{Q_i} is the throughput of the Q_i and T_{Q_m} is the average value of the throughput of all queues in the system. Adjusting the reward for this factor discourages the allocation of additional resource to the components of a queue when they are no longer bottlenecks on that queue.

In the case where a program is executed on an environment where the resources are not necessarily free-of-charge (e.g. cloud infrastructure), we may add two penalty factors:

1) GPU penalty, $P_{GPU} = \sqrt{\frac{N_{UG}}{N_{TG}}}$, where N_{UG} is the number of GPUs in the system that have not been used by the static mapping, and N_{TG} is the total number of available GPUs in the system.
2) CPU penalty, $P_{CPU} = \sqrt{\frac{N_{UC}}{N_{TC}}}$, where N_{UC} is the number of CPU cores in the system that have not been used by the static mapping, and N_{TC} is the total number of available CPU cores in the system.

The main effect of P_{GPU} and P_{CPU} is to force the system to use all of the available resources, by introducing a penalty for unused resources. If we add these two factors to the definition of the reward function, then in the case where more than one mapping achieves the same throughput and utilisation, the one that uses the smallest amount of resources is chosen. Conversely, if we omit these two factors, the mapping that uses the largest amount of resources is chosen.

Since our assumption is that the system on which programs are executed is dedicated private machine, we do not use P_{GPU} and P_{CPU} in the reward function, which is therefore

$$Q(v) = T - (SD_U + SD_Q),$$

for a selected path, v, of the decision tree. If we were to use the two penalty factors, the reward function would become $Q(v) = T - (SD_U + SD_Q + P_{CPU} + P_{GPU})$.

G. Back-propagation, Termination Condition and Final Move Selection

We have considered two back-propagation policies [30]: the *Max* policy, where the maximal reward of all the children is propagated to their parent, and the *Average* policy, where the average reward of all the children is propagated to their parent.

The MCTS algorithm finishes if no new moves have been made for K iterations. The final move selection is based on the robust-max child. The robust-max child is the child with both the highest visit count and the highest value. If there is no robust-max child at the moment, more simulations are run until a robust-max child is obtained [30].

V. Case Study: Image Convolution in FastFlow

In order to evaluate our approach, we have used an *image convolution* [32] as our example parallel program. Image convolution is widely used in image processing applications, e.g. for blurring, smoothing and edge detection. Our version of image convolution consists of reading a stream of images into the memory and applying the *convolution* function (with a given filter) to each of these images. Applying the convolution function to an input image conists of calculating, for each pixel of the input image, a scalar product of the "window" surrounding that pixel with the filter weights, and storing the resut in the output image at the same position:

$$out(i,j) = \sum_m \sum_n in(i-n, j-m) \times filter(n,m),$$

where $out(i,j)$ is the pixel of the output image at position (i,j), $in(i,j)$ is the pixel of the input image at position (i,j), m and n are the dimensions of the filter, and $filter(i,j)$ is the pixel of the filter at position (i,j).

We emphasise that our intention here is to demonstrate the applicability of our MCTS approach to deriving optimal static mapping for computationally-demanding real-world FastFlow applications, not to develop a new version of the CPU/GPU convolution algorithm. The application that we use is implemented in the GPU-extended version of FastFlow [29], where it is shown that the hand-tuned version of the application achieves a 45 speedup compared to the sequential version for large-scale inputs. Here, as a proof of concept, we demonstrate that our MCTS approach can derive the static mapping that delivers a comparable speedup, therefore opening up the prospect for automating tuning for non-experts.

We consider two different skeleton tree structures, or factorisations, for the image convolution application [29]. In both factorisations there are two kinds of components: r, which reads a single image into memory, and p which applies the convolution function to a single image. r only has a CPU version, whereas p has both CPU and GPU versions. Therefore, there are two different p components, which we will denote by P_{CPU} and P_{GPU}. The two factorisations that we consider are:

1) Pipe(Farm(r), Seq(p_{GPU})) and
2) Pipe(Farm(r), Farm(p_{CPU}, p_{GPU})).

Table I shows the information about the hardware we used in the evaluation of the MCTS algorithm. Table II shows the two instances of the problem that we consider. We use the Factorisation 1 for large-scale problems where we cannot run more than one instance of a component on a single GPU. Since in our system we have only one GPU, it means that we can have at most one instance of p_{GPU}. We use Factorisation 2 for small-scale problems where we can run more than one instance of a component on a single GPU.

Figure 3 compares the average absolute speedup obtained with a mapping calculated by the MCTS algorithm over 1000 executions with the hand-tuned version for both factorisations, compared with a sequential version. The speedups obtained with the mapping calculated by the MCTS algorithm are within 5% of the speedups of the hand-tuned version. Figure 4 shows the utilisations of the components of the application. We can observe that the utilisation of all components is above 0.6. Considering the fact that each component is assigned to a resource in the system this implies an overall 0.6 utilisation of the system. Figure 5 shows the throughput of the queues in the application. We can observe that the throughput of both queues is the same, which indicates perfect balance of the pipeline.

Parameter	Value
No. of CPUs	2
Cores per CPU	6
CPU Clock	3.07 GHz
Physical Memory	50 GB
No. of GPUs	1
GPU Model	NVIDIA Tesla M2090
GPU Memory	6 GB
GPU Cores	512
CUDA Version	4.0 V0.2.1221
GPU Driver Version	290.10

TABLE I. HARDWARE SPECIFICATION FOR RUNNING ACTUAL CONVOLUTION PROBLEM

.

Category	Input Image Size	Filter Size	No. Input Images
Small Scale	2048*2048	15*15	1000
Large Scale	8192*8192	48*48	1000

TABLE II. PROBLEM CATEGORIES

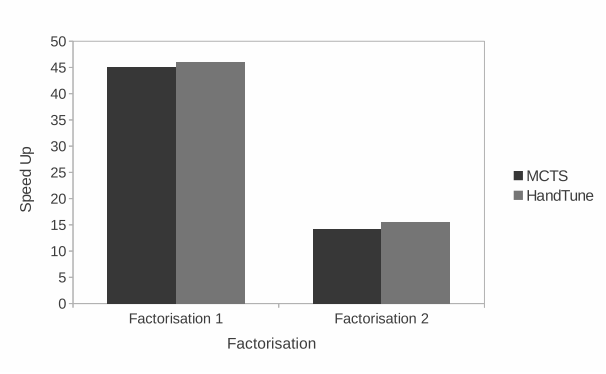

Fig. 3. Speedups obtained with a static mapping calculated by the MCTS algorithm and with a hand-tuned version

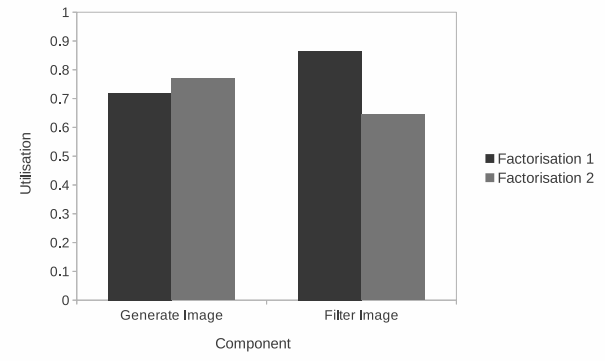

Fig. 4. Utilisation of the components in the application

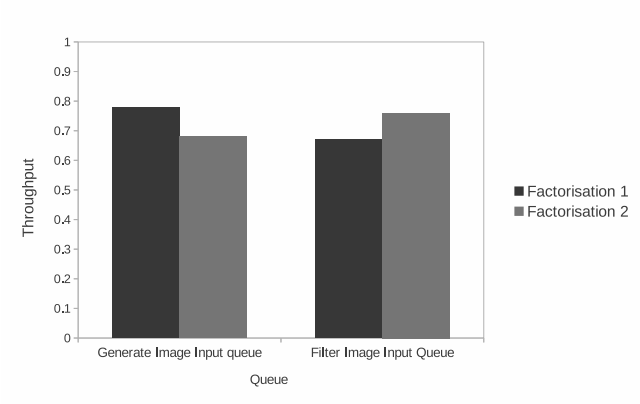

Fig. 5. The throughput of queues in the application

VI. ANALYSIS OF MCTS ON STATIC MAPPING PROBLEM

In this section we discuss in more detail the choice of MCTS for the static mapping problem.

The proposed approach aims to target FastFlow applications which are computationally heavy and contain a nested combination of farms and pipelines. Such applications usually take hours or even days to run [29], [33]. For example, running the application presented in Section IV for an input stream of 8000 images on a machine with 2 GPUs and 24 CPUs takes around 4 hours. Therefore, the effort required to run the MCTS algorithm to obtain an optimal mapping, even for a relatively simple problem like convolution, is amply justified in terms of the savings in time and energy realised by achieving speedups over such long run times.

The solution space is also sufficiently complex and expensive to justify an advanced search approach. The size of the solution space size depends on the number of components of the application and the amount of resources available in the target architecture. For example, consider the skeleton tree structure in Figure 1, presented in Section IV, which can be categorised as a small problem.For an architecture with 16 CPUs and 1 GPU the solution space size is 1240; for an architecture with 24 CPUs and 2 GPUs problem size would be 6624 and for a 64 cores machine with 2 GPU the problem size would be have 129204 combinations. The costly operation here is the performance evaluation of each possible solution in terms of throughput and resource utilisation. For this example, using the provided service time for each function, the evaluation of each solution can vary from 12 seconds to 107 seconds. So running all the possible solutions for even 1240 possible solutions for the 16 cores machine with 1 GPU can take more than a day to analyse. The simulation of this application for 24 cores CPU and 2 GPUs depends on the quality of the selected path but can vary from 14 to 200 seconds. Exhaustive search of this solution space will take almost a week. Finally, in the case of a system with 64 CPU and 2 GPU, the solution space will contain 129204 solutions, which is impractical for exhaustive investigation. We emphasise again that this is a small problem on a small target architecture. Large machines in operation today can have thousands of processors, so for large scale problem with large resources there is a need to search intelligently for high quality mappings.

TABLE III. RUNTIME RESULTS FOR DIFFERENT MCTS VARIATIONS IN FIGURE 1 OVER A PLATFORM WITH 24 CPU 2 GPU.

Name	CPU/GPU number	Percentage of Population Seen
MCTS-AVG	24/2	0.20
MCTS-AVG	16/1	0.32
MCTS-Max	24/2	0.09
MCTS-Max	16/1	0.19

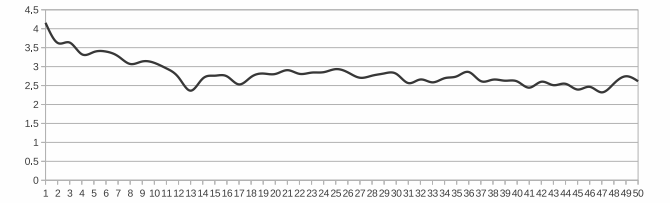

Fig. 6. Average fitness over 50-step random walks from the global optimum. the x-axis represents the walk steps and the y-axis represents the system throughput in each walk step.

In order to analyse the solution space we study the fitness-distance correlation (FDC) and landscape correlation (AC1) for the above example over a platform with 16 cores CPU and 1 GPU. Exhaustive evaluation of the example indicates that the best solution is "a:5","b:8", "g:1" and "c:3". As stated in figure 6, we run 50-step random walks from the global optimum, 100 times, to calculate the FDC and AC for the solution space.

Table IV, displays the FDC obtained for this problem. This indicates a clear gradient in solution quality as distance increased, showing that it is reasonable to apply heuristic search. [34]. However, AC1 value is comparable with the reported AC1 on the NK landscape problem [35] with k=50 and N=100 which is quite rugged. This counter-indicates the use of simple heuristics such as hill-climbing which is likely to become trapped in local optima.

In the UCT approach applied here, the factor C_P can be considered as a greediness factor of the algorithm. Therefore, considering the ruggedness of the solution space, selecting too small a value for C_P can result in searching smaller regions of the solution space, with a higher risk of becoming trapped in local optima. However, choosing too large a value for C_P slows or even prevents convergence. In our experience values of C_P around $\frac{1}{5}$th of the average throughput gives good convergence to the optimal solution.

As shown in tableIII all of the optimum results were found by exploring $9 - 32\%$ of the solution space. We expect that, on larger problems with deeper trees, these percentages will drop dramatically as the number of unevaluated paths will increase exponentially. Thus the justification for MCTS grows with search space size.

We also observe that, since the MCTS-AVG is a less greedy approach than the MCTS-MAX, MCTS-AVG is a better algorithm for this problem. As stated in figures 7 and 8, the run

TABLE IV. THE FDC AND AC1 VALUE FOR PROBLEM SIZE 1240.

Name	Value	Standard Deviation
AC1	0.51	0.18
FDC	-0.33	0.32

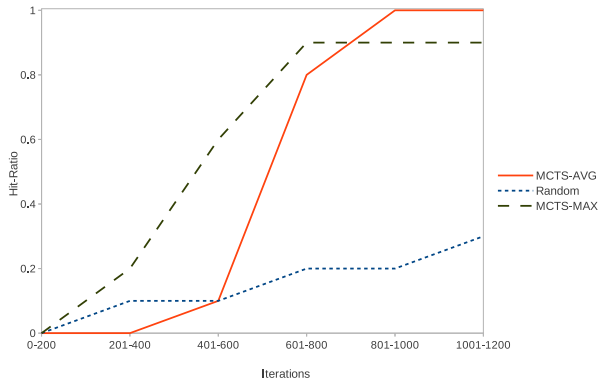

Fig. 7. Run Length Distribution on 16CPU 1GPU platform

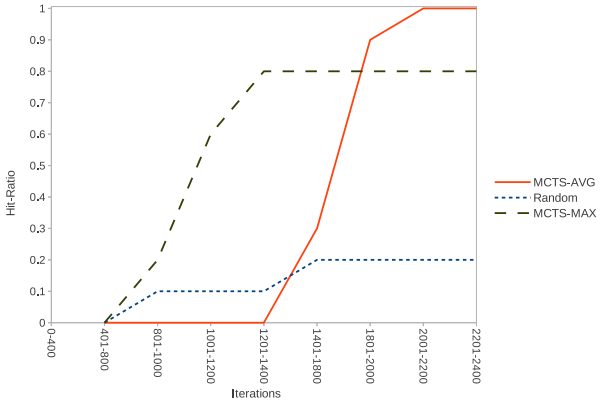

Fig. 8. Run Length Distribution on 24CPU 2GPU platform

length distribution, shows that MCTS-AVG is more accurate than MCTS-MAX for larger problems. Figure 8 shows that the MCTS-MAX converges on local optima on 20% of runs. However, when the solution space is small, the probability of MCTS-MAX converging on a local optimum is reduced (figure 7).

Moreover, the convergence of MCTS-AVG is more predictable than that of MCTS-MAX, as we can see in figures 7 and 8, for both cases 80% of the convergence are in certain iteration, while in MCTS-MAX this is less predictable. However, the number of population visited by MCTS-Max is less than that in MCTS-AVG. Since the simulation is a costly function here, this means that the cost of MCTS-MAX is less than the cost of MCTS-AVG.

Moreover, Due to the large differences between the speed of CPU and GPU, a set of solutions with almost the same throughput as the global optimum exist. Since the static mapping is based on estimation over the simulated value, it is reasonable to accept all of these as best solutions. During runtime of the parallel program, dynamic factors come into play which affect the performance of the system and a light weight dynamic remapping technique would be employed to further tune the solution at runtime if needed. However, the process of dynamic remapping is expensive. Hence, the selected static mapping solution should be robust enough to require the least possible effort at runtime. MCTS-AVG tends

to select solutions with robust sub-trees. By this we mean that, at each decision step, the number of good solutions found in the selected sub-tree (child) of a parent is more than other sub-trees (children) of that parent. This is useful because it increases the chances of needing less effort for further tuning of selected path at runtime, since it increases the chance of reaching another good solution by making small changes to the current selected solution. MCTS-MAX is less likely than MCTS-AVG to produce robust subtrees, since its evaluation is based solely on the best result of its children.

Finally, comparing the run length distributions of the MCTS algorithms with those of a simple random search demonstrates the considerable extent to which learning assists in finding a high quality solution with non-exhaustive effort.

VII. Conclusion

This paper has presented an intelligent approach to automating the mapping of a FastFlow program over a heterogeneous multicore architecture accelerated with GPU. The approach uses Monte Carlo Tree Search and provides solutions competitive with hand tuned expert solutions (upto 95% speed up over hand tuned approach) without requiring specialist knowledge.

We also analysed the applicability of MCTS to the mapping problem in terms of its suitability, scalability and solution quality. Versions of MCTS with different back-propagation policies were analysed. We have shown that MCTS with average back-propagation provides robust solutions, most suitable as a basis for dynamic remapping at runtime.

In the future, we are planning to integrate the idea of streaming parallelism with data parallelism and to extend the applicability of our algorithm to other algorithmic skeletons, especially the Map skeleton for FastFlow. We also intend to implement the dynamic remapping of the FastFlow program as an extension over the generated result of the static mapping approach.

A remaining issue for further investigation is the applicability of this approach over a distributed heterogeneous architecture, where the cost of transferring both components and data over a network is accounted for in the simulation, as it affects the system performance and utilisation of components.

References

[1] M. Cole, *Algorithmic Skeletons: Structured Management of Parallel Computation*, ser. Research Monographs in Parallel and Distributed Computing. London: MIT Press/Pitman, 1989.

[2] H. González-Vélez and M. Leyton, "A Survey of Algorithmic Skeleton Frameworks: High-Level Structured Parallel Programming Enablers," *Software–Practice and Experience*, vol. 40, no. 12, pp. 1135–1160, 2010.

[3] Z. Wang, "Machine Learning Based Mapping of Data and Streaming Parallelism to Multi-cores," Ph.D. dissertation, School of Informatics, The University of Edinburgh, Edinburgh, UK, May 2011.

[4] C. B. Browne, E. Powley, D. Whitehouse, S. M. Lucas, P. I. Cowling, P. Rohlfshagen, S. Tavener, D. Perez, S. Samothrakis, and S. Colton, "A survey of monte carlo tree search methods," *Computational Intelligence and AI in Games, IEEE Transactions on*, vol. 4, no. 1, pp. 1–43, 2012.

[5] Y.-K. Kwok and I. Ahmad, "Static Scheduling Algorithms for Allocating Directed Task Graphs to Multiprocessors," *ACM Computing Surveys (CSUR)*, vol. 31, no. 4, pp. 406–471, 1999.

[6] V. A. Saraswat, V. Sarkar, and C. von Praun, "X10: Concurrent Programming for Modern Architectures," in *Proceedings of the 12th ACM SIGPLAN symposium on Principles and practice of parallel programming*, ser. PPoPP '07. New York, NY, USA: ACM, 2007, pp. 271–271. [Online]. Available: http://doi.acm.org/10.1145/1229428.1229483

[7] J. Subhlok, J. M. Stichnoth, D. R. O'hallaron, and T. Gross, "Exploiting Task and Data Parallelism on a Multicomputer," *ACM SIGPLAN Notices*, vol. 28, no. 7, pp. 13–22, 1993.

[8] K. Ramamritham and J. A. Stankovic, "Dynamic Task Scheduling in Hard Real-Time Distributed systems," *IEEE Softw.*, vol. 1, no. 3, pp. 65–75, Jul. 1984. [Online]. Available: http://dx.doi.org/10.1109/MS.1984.234713

[9] M. I. Gordon, W. Thies, and S. Amarasinghe, "Exploiting Coarse-Grained Task, Data, and Pipeline Parallelism in Stream Programs," in *ACM SIGOPS Operating Systems Review*, vol. 40, no. 5. ACM, 2006, pp. 151–162.

[10] D. E. Culler, R. M. Karp, D. Patterson, A. Sahay, E. E. Santos, K. E. Schauser, R. Subramonian, and T. von Eicken, "LogP: A Practical Model of Parallel Computation," *Communications of the ACM*, vol. 39, no. 11, pp. 78–85, 1996.

[11] A. Navarro, R. Asenjo, S. Tabik, and C. Cascaval, "Analytical Modeling of Pipeline Parallelism," in *Parallel Architectures and Compilation Techniques, 2009. PACT'09. 18th International Conference on*. IEEE, 2009, pp. 281–290.

[12] M. Kudlur and S. Mahlke, "Orchestrating the Execution of Stream Programs on Multicore Platforms," in *ACM SIGPLAN Notices*, vol. 43, no. 6. ACM, 2008, pp. 114–124.

[13] Z. Wang and M. F. O'Boyle, "Partitioning Streaming Parallelism for Multi-Cores: a Machine Learning Based Approach," in *Proceedings of the 19th international conference on Parallel architectures and compilation techniques*. ACM, 2010, pp. 307–318.

[14] A. H. Hormati, Y. Choi, M. Kudlur, R. Rabbah, T. Mudge, and S. Mahlke, "Flextream: Adaptive Compilation of Streaming Applications for Heterogeneous Architectures," in *Parallel Architectures and Compilation Techniques, 2009. PACT'09. 18th International Conference on*. IEEE, 2009, pp. 214–223.

[15] C. Ykman-Couvreur, "Exploration Framework for Run-Time Resource Management of Embedded Multi-Core Platforms," in *Embedded Computer Systems (SAMOS), 2010 International Conference on*. IEEE, 2010, pp. 333–340.

[16] C. Ykman-Couvreur, V. Nollet, T. Marescaux, E. Brockmeyer, F. Catthoor, and H. Corporaal, "Design-time application mapping and platform exploration for MP-SoC customised run-time management," *Computers & Digital Techniques, IET*, vol. 1, no. 2, pp. 120–128, 2007.

[17] C. Ykman-Couvreur, P. Avasare, G. Mariani, G. Palermo, C. Silvano, and V. Zaccaria, "Linking Run-Time Resource Management of Embedded Multi-Core Platforms with Automated Design-Time Exploration," *Computers & Digital Techniques, IET*, vol. 5, no. 2, pp. 123–135, 2011.

[18] P. Avasare, G. Vanmeerbeeck, C. Kavka, and G. Mariani, "Practical Approach for design space explorations using simulators at multiple abstraction levels," in *Proc. Design Automation Conf. User Track, Anaheim, CA*, 2010.

[19] S. Agrawal, W. Thies, and S. Amarasinghe, "Optimizing Stream Programs Using Linear State Space Analysis," in *Proceedings of the 2005 international conference on Compilers, architectures and synthesis for embedded systems*. ACM, 2005, pp. 126–136.

[20] A. Udupa, R. Govindarajan, and M. J. Thazhuthaveetil, "Software pipelined execution of stream programs on gpus," in *Code Generation and Optimization, 2009. CGO 2009. International Symposium on*. IEEE, 2009, pp. 200–209.

[21] R. Coulom, "Efficient selectivity and backup operators in monte-carlo tree search," *Computers and Games*, pp. 72–83, 2007.

[22] L. Kocsis and C. Szepesvári, "Bandit based monte-carlo planning," *Machine Learning: ECML 2006*, pp. 282–293, 2006.

[23] G. Chaslot, J.-T. Saito, B. Bouzy, J. Uiterwijk, and H. J. Van Den Herik, "Monte-carlo strategies for computer go," in *Proceedings of the 18th BeNeLux Conference on Artificial Intelligence, Namur, Belgium*, 2006, pp. 83–91.

[24] S. Gelly and Y. Wang, "Exploration exploitation in go: Uct for monte-carlo go," 2006.

[25] G. Chaslot, S. De Jong, J.-T. Saito, and J. Uiterwijk, "Monte-carlo tree search in production management problems," in *Proceedings of the 18th BeNeLux Conference on Artificial Intelligence*, 2006, pp. 91–98.

[26] H. Nakhost and M. Müller, "Monte-carlo exploration for deterministic planning," in *IJCAI*, 2009, pp. 1766–1771.

[27] A. Couëtoux, J.-B. Hoock, N. Sokolovska, O. Teytaud, and N. Bonnard, "Continuous upper confidence trees," *Learning and Intelligent Optimization*, pp. 433–445, 2011.

[28] M. Aldinucci, M. Danelutto, P. Kilpatrick, M. Meneghin, and M. Torquati, "Accelerating Code on Multi-cores with FastFlow," in *Euro-Par 2011*, ser. LNCS, vol. 6853. Bordeaux: Springer, Aug. 2011, pp. 170–181.

[29] M. Goli and H. González-Vélez, "Heterogeneous Algorithmic Skeletons for Fast Flow with Seamless Coordination over Hybrid Architectures," in *PDP*. IEEE Computer Society, 2013, pp. 148–156.

[30] G. Chaslot, "Monte-carlo tree search," Ph.D. dissertation, PhD thesis, Maastricht Univ, 2010.

[31] L. Kocsis, C. Szepesvári, and J. Willemson, "Improved monte-carlo search," *Univ. Tartu, Estonia, Tech. Rep*, vol. 1, 2006.

[32] V. Sobolev, "Convolution of functions," in *Encyclopedia of Mathematics*, M. Hazewinkel, Ed. Springer-Verlag, 2001.

[33] M. Goli, M. T. Garba, and H. González-Vélez, "Streaming Dynamic Coarse-Grained CPU/GPU Workloads with Heterogeneous Pipelines in FastFlow," in *14th International Conference on High Performance Computing and Communications (HPCC)*. Liverpool: IEEE Computer Society, Jun. 2012, to appear.

[34] T. Jones and S. Forrest, "Fitness Distance Correlation as a Measure of Problem Difficulty for Genetic Algorithms," in *Proceedings of the 6th International Conference on Genetic Algorithms*. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 1995, pp. 184–192. [Online]. Available: http://dl.acm.org/citation.cfm?id=645514.657929

[35] W. Hordijk and S. A. Kauffman, "Correlation Analysis of Coupled Fitness Landscapes: Research Articles," *Complex.*, vol. 10, no. 6, pp. 41–49, Jul. 2005. [Online]. Available: http://dx.doi.org/10.1002/cplx.v10:6