Bridging the Divide: Intelligent Mapping for the Heterogeneous Parallel Programmer

Christopher Brown*, Vladimir Janjic*, Kevin Hammond*
*School of Computer Science
University of St Andrews, St Andrews, UK
E-Mail : {cmb21,jv32,kh8}@st-andrews.ac.uk

Mehdi Goli†, John McCall†
†School of Computer Science
Robert Gordon University, Aberdeen, UK
E-Mail : {m.goli1,j.mccall}@rgu.ac.uk

***Abstract*—Heterogeneous shared-memory systems, comprising a mixture of CPUs and GPUs, are increasingly common. Current techniques for programming such systems (such as OpenCL or CUDA+OpenMP) offer very little abstraction, however, relying on the programmer to deal with many low-level details such as thread creation, data transfers, scheduling etc. An even bigger problem is that the division of work between CPUs and GPUs needs to be performed manually by the programmer. In this paper we describe a new approach for programming heterogeneous systems using a combination of programmer-directed *refactoring* and *Monte Carlo Tree Search (MCTS)* techniques. Refactoring eases the task of writing parallel programs by offering the programmer step-by-step guidance for transforming theier sequential program into an equivalent parallel program, and/or for transforming parallel programs into better ones. The MCTS algorithm enables us to derive good mappings of the resulting parallel programs to the available heterogeneous hardware. We demonstrate the applicability of our methodology using an Image Convolution benchmark, obtaining a speedup of 40.91 on a machine comprising 2x 2.4Ghz 12-core AMD Opteron 6176 CPUs coupled an NVidia Tesla C2050 with 448 GPU cores running at 1.16GHz. We also demonstrate that the speedups that we obtain using the mappings that the MCTS algorithm suggests are within 5% of the best possible speedups that can be obtained.**

I. Introduction

Heterogeneous multicore systems, comprising a mixture of CPUs and GPUs, are increasingly common, combining perhaps dozens of CPU cores with perhaps hundreds of GPU cores. Programming such systems remains difficult, however. State-of-the-art programming techniques, such as OpenCL or CUDA+OpenMP, generally involve exploiting low-level programming techniques that require precise in-depth knowledge of the GPU architecture together with information about data-transfers, scheduling decisions etc., usually do not scale easily or well, and cannot easily be retargeted to alternative hardware architectures. In this paper, we introduce a mew methodology for programming heterogenous parallel systems, based on a combination of high-level parallel patterns, template-based implementations (skeletons), advanced refactoring technology, and a new static mapping approach that exploits Monte Carlo Tree Search (MCTS) to find good mappings of tasks to heterogeneous architectures. Parallel patterns provide design-time *choice* and parallel program *structure*; skeletons provide correspondingly structured implementations and a decomposition into parallel program components; the MCTS-based mapping model provides the programmer with *decisions* that ensure good static mappings of these components to the available heterogeneous hardware; and refactoring provides the *discipline* and *tool-support* to introduce and tune the parallelism, abstracting over the detail of the skeletons, and exposing performance information based on the decisions provided by the static mapping. We demonstrate how to use our methodology on a realistic Image Convolution benchmark, targeting a 24-core machine, comprising 2x 2.4Ghz 12-core AMD Opteron 6176 CPUs, plus an NVidia Tesla C2050 GPU with 488 1.16GHz cores. Using this machine, we obtain scalable speedups of up to 40.91 over the sequential implementation. Our results show that these speedups are within 5% of the best possible speedup using any possible static mapping. Our concrete experiments are conducted using C++ and the FastFlow skeleton library. However, the principles and techniques that we describe are not limited to C++ or FastFlow, but are *completely generic*, and can easily be transferred to other languages/paradigms, such as C, Java, Fortran or Haskell, to other skeleton libraries, such as SkePU, and to other parallel implementations, such as direct CUDA, pThreads or Cilk.

The paper makes the following research contributions:

1) We introduce a new semi-automated methodology for building parallel programs, based on parallel patterns, semi-automatic refactoring, and high-level algorithmic skeletons;
2) We introduce a new static-mapping framework, based on Monte-Carlo Tree Search simulations, for deriving efficient mappings of parallel application threads to heterogeneous CPU and GPU hardware;
3) We introduce a number of novel refactorings for introducing parallelism for heterogenous many-core systems, implemented in the CDT refactoring plugin for Eclipse;
4) We show how our methodology applies to a convolution benchmark, obtaining promising and scalable speedups on a heterogenous parallel system.

II. Methodology

Our general methodology is shown in Figure **??**. We assume that the initial application is already in a "hygienic" state, i.e., that the units of computation are defined as *components* whose dependencies and interactions are clearly specified. We also

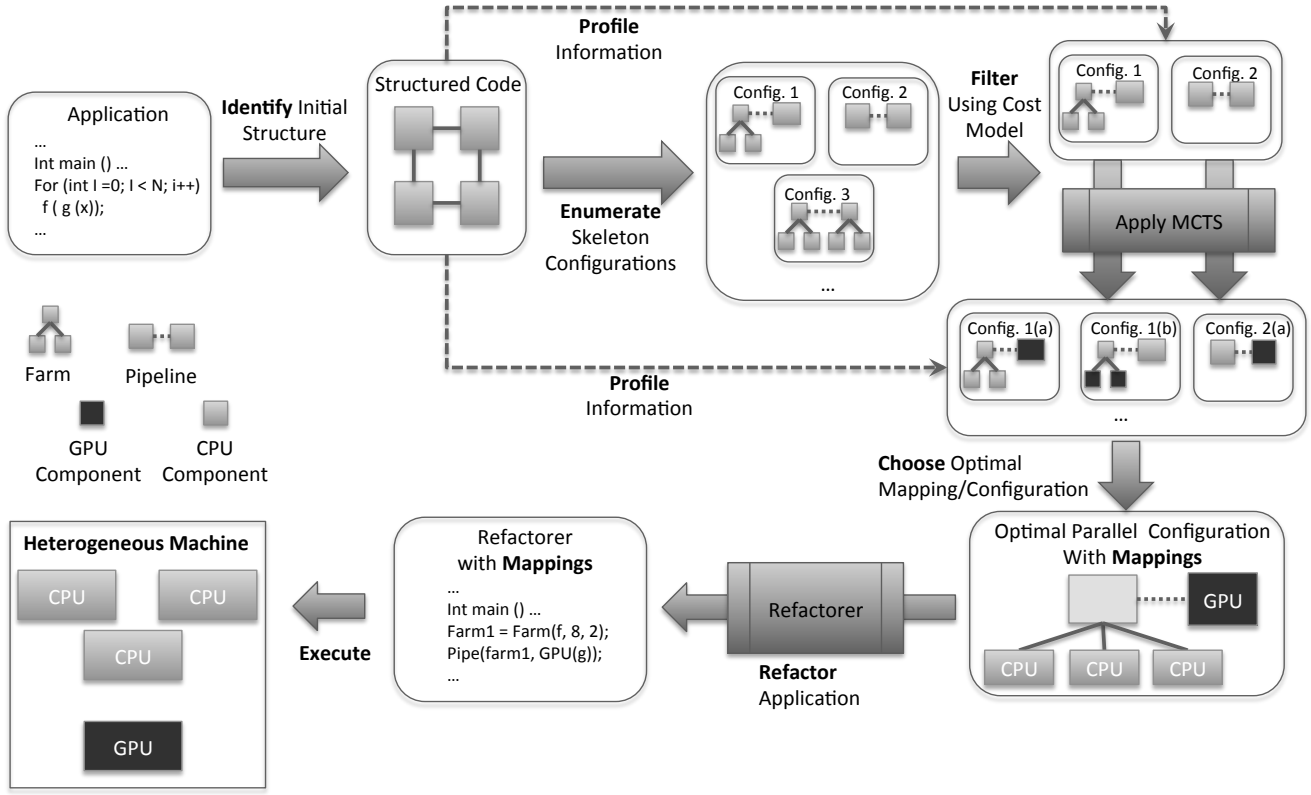

Fig. 1. Overview of our methodology. We start with a (sequential) program with an identified skeletal structure. All configurations of the structure are then enumerated, and can be filtered using cost models. The remaining configurations are given to a Monte-Carlo Tree Search algorithm, which determines mapping and skeletal parameters. This information is then given to a refactoring tool, which semi-automatically applies the skeletal transformation.

assume that CPU and GPU implementations are available for each component, in a way that allows them to be linked (perhaps using automatically provided marshalling/unmarshalling code, where it is necessary to offload a component); and that profiling information is also available.

1) Assuming that we start with a program that has not previously been parallelised, it is first necessary to *identify* and extract the algorithmic structure in the form of an abstract *skeletal configuration*, which encapsulates the structure of the algorithm together with its component information, showing the relationships and dependencies between the components, and eliminating/exposing any shared state through interfaces to the component.
2) Given the identified skeletal configuration, all possible equivalent skeletal configurations of the program are *enumerated* up to a given depth of nesting, resulting in a number of possible parallel *factorisations*.
3) Using *profiling* information, these factorisations are *filtered* using a cost model, if desired, to restrict the number of possibilities that need to be considered. This allows us to eliminate potential factorisations that give little or no speedup at an early stage of development. In Section **??**, we use a simple high-level cost model to predict the best possible run times for each configuration. At this stage, exact timing information is not needed: it is more important for the cost model to predict a minimal runtime than to be completely accurate.
4) The remaining configurations are then given to the *MCTS* model, together with the profiling information and hardware information. In order to predict runtimes, we also supply mapping parameters (such as the division of work between CPUs and GPUs) and any skeletal parameters (such as the number of worker threads for a *farm* skeleton) for each configuration. This stage returns a ranking for the configurations (together with mapping and skeletal parameters), based on the estimated runtime.
5) The programmer then selects one of the factorisations, based on the output of the MCTS model, any profiling information, and their insight into the nature of the program. The refactoring tool *automatically* transforms the source code of the application into the chosen factorisation, introducing calls to skeletons and instantiating any relevant parameters, and indicating which components should be executed on a CPU or GPU.
6) Finally, the refactored program can be executed on the available heterogeneous hardware.

A program may be refactored as many times as necessary. This may be useful if the application is to be ported to a different architecture, for example. The programmer may restart the process with the refactored application, or *undo* any previous refactorings already made, starting with the original sequential program. Where a program has already been parallelised,

it must first be structured into a component-based program. Obviously, this might involve undoing some design designs, so that a more structured approach can be used. However, it might also involve encapsulating existing paralellism so that it can be easily reused. While this is an interesting problem, we will not consider this further in this paper, focusing instead on the primary problem of introducing and exploiting parallelism, as a necessary first step.

A. 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 programmer supplies sequential problem-specific code and any necessary skeletal parameters. In this paper, we restrict ourselves to three fundamental, heterogeneous, skeletons:

- The *Comp ($\circ$)* skeleton models sequential function composition, where $f1 \circ f2$ denotes a sequential composition of two functions, $f1$ and $f2$.
- The *Pipeline ($\|$)* skeleton $(f_1 \parallel f_2 \parallel \cdots \parallel f_n)(x)$ applies the f_i in turn to the input stream $x_1, x_2, \ldots, x_m$, and each f_i is executed in parallel to form a pipeline. For each stage of the pipeline, f_n, can either be executed on a CPU or a GPU.
- A *Farm (Δ)* skeleton, $\Delta(nwCPU, nwGPU, f, x)$, models the application of a function, f, to the input stream $x_1, x_2, x_3, \ldots, x_n$. Here *nwCPU/nwGPU* are the number of worker threads executed on CPUs/GPUs, respectively[1].

We also allow nested skeletons. It is, therefore, possible to e.g. nest a pipeline inside a farm, $\Delta(nwCPU, nwGPU, f1 \parallel f2, x)$. A *skeletal configuration* focuses only on the structure of the nesting of skeletons, abstracting details of e.g. the inputs to the skeletons and the number and type of workers in a farm. In a skeletal configuration, we denote $\Delta(nwCPU, nwGPU, f, x)$ simply by $\Delta(f)$, and $(f_1 \parallel f_2 \parallel \cdots \parallel f_n)(x)$ by $f_1 \parallel f_2 \cdots \parallel f_n$. For example, the skeletal configuration $\Delta(f) \parallel (g \circ \Delta(h))$. denotes a pipeline of two stages, i) a farm whose worker function is f, and ii) a sequential composition of function g with a farm whose worker function is h.

B. FastFlow

FastFlow [**?**] 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 heteogeneous parallel system [**?**].

[1]Note that, for each GPU worker thread, we actually need one CPU thread that moves the input data to the GPU memory, sends the kernels to the GPU, and retrieves the results.

III. Monte Carlo Tree Search Model for Deriving Static Mappings

In this section, we describe a model that we use to derive, for a given skeletal configuration, the best static mapping of its components to the available hardware. A static mapping in our case corresponds to a particular choice of values for the parameters of skeletons, i.e. the number of workers in each farm, the type (CPU or GPU) of each worker in each farm and each stage of each pipeline. We consider the best static mapping to be the one that yields the best runtime for that particular skeletal configuration.

Our model accepts as an input a skeletal configuration and the the timings for its sequential components (both for CPU and GPU versions, if the GPU version of a component is available). As an output, it produces the best static mapping (according to a specific evaluation function, Q) and the estimated runtime of the skeletal configuration with that static mapping. Since considering all possible static mappings for a given skeletal configuration may be computationally intractable, our model uses the Monte Carlo Tree Search (MCTS) approach for generating and evaluating large game trees in Game theory. In our case, in the generated tree, the inner nodes correspond to the partial mappings (with some of the skeleton parameters chosen) and the leaves of the tree (with all parents to the root) corresponds to the complete mappings (with all parameters of all skeletons chosen). The children of a node, v, represent different possible assignments of a yet unassigned skeleton parameter of incomplete static mapping, v.

The MTCS approach starts from a tree that consists only of a single root node (i.e. a static mapping where no parameters for any skeleton are chosen). It proceeds by repeating the following three steps:

1) *Expansion step* – A node that corresponds to some partial static mapping is selected, and one of its children is added to the tree. This corresponds to assigning value for one parameter of one skeleton in our skeletal configuration;
2) *Selection step* – Starting from the newly added node, a complete static mapping is generated by randomly choosing the remaining skeleton parameters. Once this static mapping is generated, its runtime is estimated using simulations, and the evaluation function, Q, is applied to that mapping and the runtime, giving a valuation, v;
3) *Propagation step* – The valuation, v, is propagated back to the node added in step 1, and that node is assigned a valuation v.

The algorithm will terminates, when after K iterations, no new solutions have been generated. At this point, from the root node to the leaf, the children with both the highest visit count and the highest valuation, v, will be selected. This branch corresponds to the static mapping that is returned by the model.

The evaluation function that we use to evaluate how good static mappings are is based on the estimation of the runtime

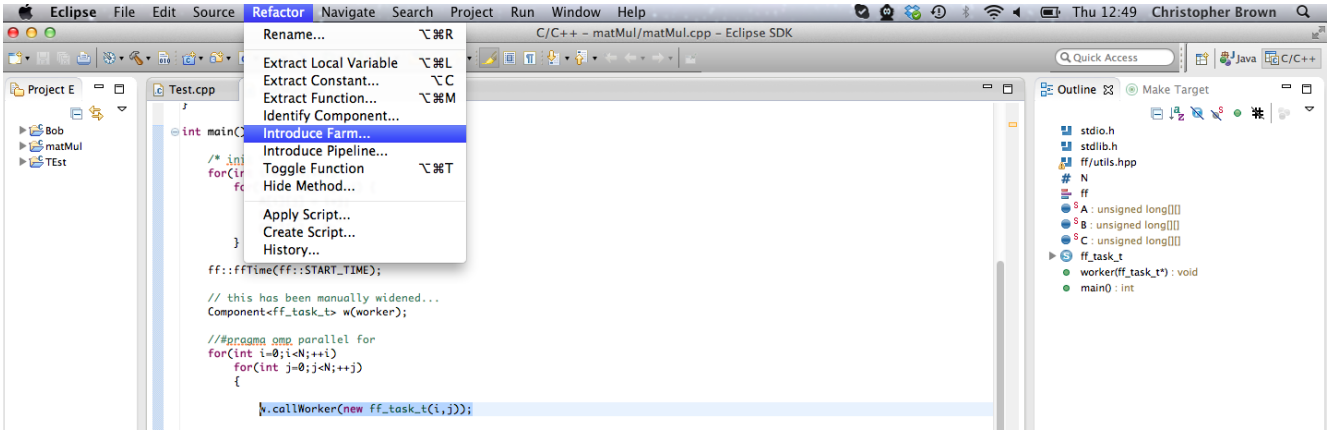

Fig. 2. C++ Parallel Refactoring Tool in Eclipse

for that static mapping that we obtain using simulations, and the utilisation of the system. The function is

$$Q(v) = S(v) - (\delta_U(v) + \delta_Q(v)),$$

where $S(v)$ is the estimated throughput of the whole system, $\delta_U(v)$ is the standard deviation of the utilisation of components and $\delta_Q(v)$ is the standard deviation of the utilisation of the connecting queues between the components in the skeleton. Using this function, $Q(v)$, rather than using just the throughput, $S(v)$, as an evaluation function, discourages our model for allocating more resources to the skeleton configuration, if it only results in marginally improved runtime, which may be important in settings where resources are paid for (e.g. clouds).

IV. Refactoring

Our refactoring prototype is implemented in Eclipse, under the CDT plugin. The programmer is presented with a menu of possible refactorings to apply. The decision to apply a refactoring and identify a possible skeleton to introduce is made by the programmer. Once a decision has been made, any required transformation and mapping is performed automatically. In this way, we can rely on programmers making informed decisions about which refactorings to apply, but do not rely on them necessarily having expertise with parallelism or skeletons. Figure **??** shows a sample screenshot of our refactoring tool, where the programmer is presented with a menu of possible parallel refactorings to apply, such as *Introduce Farm* and *Introduce Pipeline*, described below:

- **Introduce Pipeline**. For this refactoring, the programmer must select a `for` loop, e.g., the `for` loop that is shown in the first column of Figure **??**. The second column in the figure shows the effect after refactoring. Here, the original `for` loop has been re-written into a `StreamGen` stage. This is a C++ class instance that models the streaming input behaviour to the pipeline. The refactoring is not limited to only C style arrays, however, as C++ STL data structures may also be considered, such as `std:vector< >` objects. In the second column, an instance of a FastFlow *Pipe*line skeleton has been introduced, named `pipe`. The first stage, a function, `ff_pipe_first_stage`, is added as a pipeline stage; the second stage, `CPU_Stage`, is added as the final stage. Finally, the pipeline waits for the result of the computation, using a `run()` method call. The dependency between the output of `ff_pipe_first_stage` and the input of `CPU_Stage` is detected automatically.
- **Introduce Farm**. Here the refactoring has two variants: farming a *Pipe*line stage; or, alternatively, introducing a new *Farm*. Farming a pipeline stage is the simplest of the two variants. The middle column of Figure **??** shows a stage of an already defined FastFlow *Pipe*line skeleton that the programmer has highlighted. The result of the refactoring is shown in the right column: the *Pipe* skeleton has been modified to replace the selected stage—here, the function `ff_pipe_first_stage`—with a FastFlow *Farm*, called `global_farm`. All other stages in the *Pipe*line are preserved. The refactoring requires that `nworkers` (the number of *Farm* workers) to be either previously defined, or given as an explicit argument to the refactoring.

Each of the introduction refactorings has a corresponding inverse: *Farm Elimination* inverts *Introduce Farm*; *Pipeline Elimination* inverts *Introduce Pipeline*. This allows any combination of rules to be inverted, or undone, as part of a parallel refactoring process. The refactorings are also fully nest-able, allowing, for example, a farm, such as $\Delta(f1 \circ f2)$ (a task farm where the worker is a function composition of two components, *f1* and *f2*), to be refactored into $\Delta(f1 \parallel f2)$.

V. Results

In this section we demonstrate our methodology on a convolution algorithm, which we describe in Section **??** showing a number of different potential configurations of the algorithm in Section **??**. We then take the three most optimal configurations, based on a pre-filtering stage by applying high-level abstract cost models (Section **??**) to eliminate weak candidates. The remaining configurations are then passed through our MCTS model, which predicts optimal mappings for each configuration. The Application is then refactored for each configuration and mapping, where we show that the actual performance

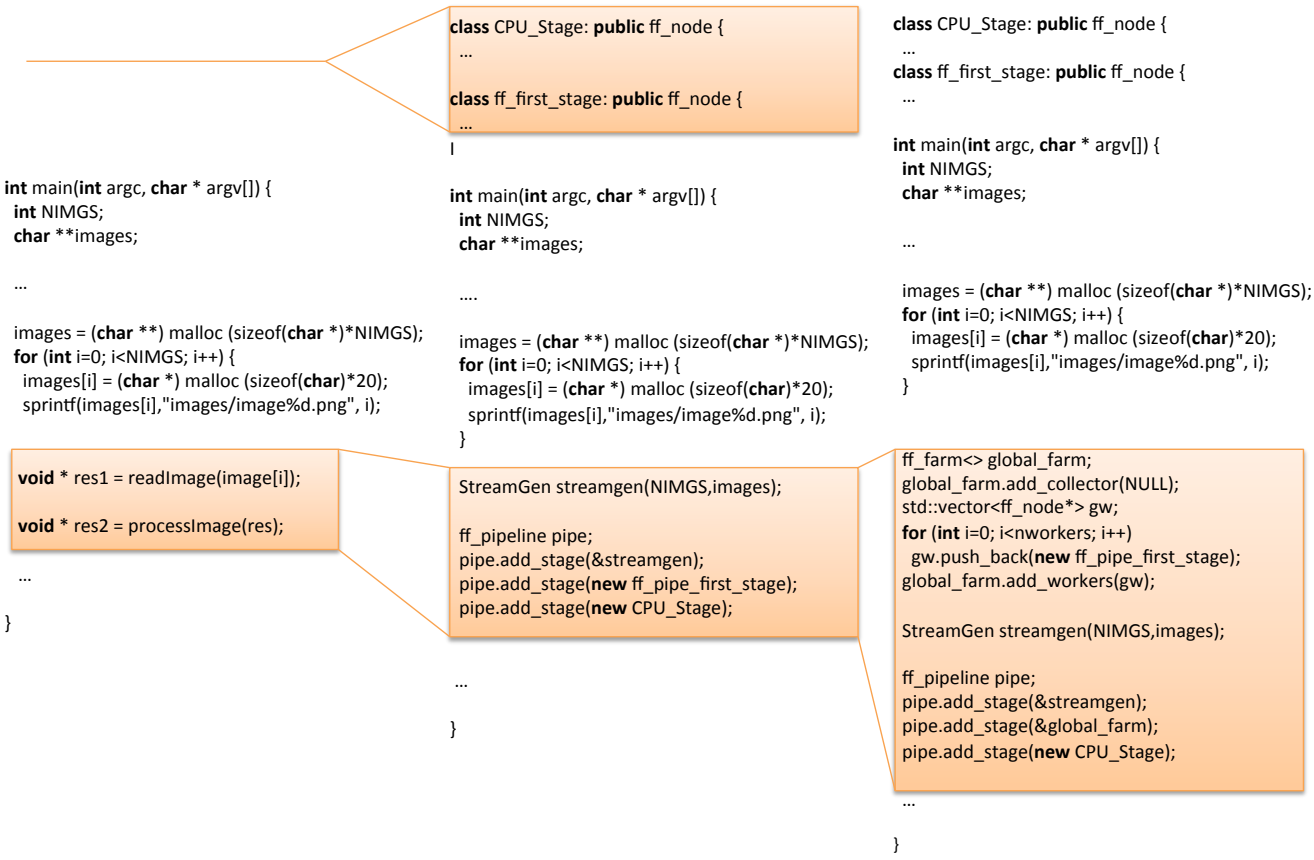

Fig. 3. The Evolution of a Skeleton Program Using Refactoring: Introduce a Pipeline Skeleton and then Introduce a Farm Skeleton for a Convolution Algorithm.

speedups of the different configurations match the mappings from the MCTS in Section **??**.

A. The Convolution Algorithm

Image convolution is a technique widely used in image processing applications such as blurring, smoothing or edge detection. The basic structure of our convolution algorithm is a two-stage function composition, $r \circ p$, where r is a stage that reads in an image from a file, and p is a stage that processes the image, by applying a filter to the image. This convolution process is typically applied to a stream of input images, where the output is also a stream of (filtered) images. Computationally, the filtering stage requires a scalar product of the filter weights with the input pixels within a window surrounding each of the output pixels:

$$output\_pixel(i,j) = \sum_m \sum_n input\_pixel(i-n, j-m) \times filter\_weight(n,m) \quad (1)$$

In order to develop the convolution problem for a Heterogeneous system, using FastFlow, we created a subclass `ff_node` to implement r (which cannot be executed on a GPU). To implement the second stage, p, to execute on a GPU, we created a subclass, `ff_gpu_solve` which is derived from `ff_oclNode`.

B. Cost Models

In this section we give a corresponding high-level cost model for each skeleton in Section **??**, derived after those presented in [**?**], [**?**]. These cost models capture the service time of our skeletons and will be used to filter the enumerated skeleton configurations, eliminating candidates that will obviously yield poor performance results. In order to demonstrate the principles of our methodology, the cost models that we consider here are intentionally high-level and simple, abstracting over many language- and architecture-specific details. Their only purpose is to give us an rough performance estimate for each configuration. If desired, more complex models could be used to yield more accurate predictions for a specific architecture, without changing the general methodology.

Function *Comp*osition ($\circ$) can be costed trivially:

$$T_{C_\circ}(L) = l * (\sum_1^m T_{stage_m}) \quad (2)$$

where L represents the maximum size of the input list and m represents the number of stages. A suitable average-case cost model for the parallel *Pipe*line is ($\|$):

$$T_{C_\|}(L) = (\sum_{1}^{m} T_{stage_m}) + (L - m) * \left(max_{(i=1..m)} T_{stage_i}\right) \quad (3)$$

This defines the cost of a steady-state pipeline in terms of the initial cost of filling the pipeline, plus the maximum execution cost for any of the stages of the pipeline multiplied by the number of inputs minus the number of stages (we have already accounted for m stages in the initial stage of filling the pipeline).

For the *Farm* skeleton (Δ), assuming that each worker task has a similar granularity and that all workers are fully occupied, the corresponding cost model is:

$$T_{C_\Delta}(N_p, N_w, L) = T_{worker} * \lceil \frac{L}{min(N_p, N_w)} \rceil + (T_{dist} + T_{gather}) * min(p, L) \quad (4)$$

For our FastFlow definition, more accurate definitions of T_{dist} and T_{gather} are:

$$T_{dist}(N_w, L) = N_w \cdot T_{spawn} + N_w \cdot (T_{setup} + T_{copy}(\frac{L}{N_w}))$$
$$T_{gather}(N_w, L) = N_w \cdot (T_{setup} + T_{copy}(\frac{L}{N_w})) \quad (5)$$

Here, N_p is the total number of cores (including GPU elements), N_w is the number of workers, and L is the size of the input list. The cost of a farm is defined as the cost of the $worker$ function times the ceiling of the number of inputs divided by the minimum of the number of cores and the number of workers. This is added to the time to distribute and gather the results, multiplied by the minimum of the number of inputs or the number or workers.

C. Configurations

The first step in our methodology is to take the initial *sequential* pattern tree that we have derived from the convolution algorithm, and enumerate all possible configurations of the pattern tree up to a given depth, N. In order to demonstrate the principles of our methodology, we choose to enumerate up to 2 nestings, resulting in 10 possible pattern-tree configurations, where each of the configurations represents a refactoring candidate. For our example, we consider introducing a *Farm* (Δ) in both stages of the *Pipe*line ($\|$), this is justified by the fact that it is plausible to read in multiple images from a database at the same time, for example, when processing multiple medical images, or when reading from multiple CCTV videos simultaneously [**?**]. In Figure **??**, we show the different possible pattern-tree configurations for the convolution algorithm, up to a nesting depth of 2. We also show the predicted runtimes

	Configuration	CPU Time	GPU Time
1	$r \circ p$	136.00	5.60
2	$r \parallel p$	125.60	3.88
3	$\mathbf{\Delta(r) \parallel p}$	**132.00**	**1.60**
4	$r \parallel \Delta(p)$	13.20	4.00
5	$\mathbf{\Delta(r) \parallel \Delta(p)}$	**13.20**	**0.40**
6	$\mathbf{\Delta(r \parallel p)}$	**13.60**	**0.56**
7	$\Delta(r \circ p)$	13.60	5.60
8	$\Delta(r) \circ \Delta(p)$	13.60	2.00
9	$\Delta(r) \circ p$	132.40	2.00
10	$r \circ \Delta(p)$	17.20	5.60

Fig. 4. Cost predicted runtimes (in milliseconds) for both CPU and GPU versions of the possible configurations of convolutions (up to depth 2)

	$\Delta(r) \parallel \Delta(p)$	$\Delta(r) \parallel p$	$\Delta(r \parallel p)$
GPU	3	1	5
CPU	6	4	5

Fig. 5. MCTS predicted optimal mappings for three configurations showing number of workers for both CPU and GPU

of each configuration based on the results of our high-level cost models for both a CPU and GPU. The times shown are the best predictions for our 24-core testbed machine, based on the following settings (obtained by profiling the application): $L = 20$, $r = 0.2ms$, $p_{CPU} = 6.6ms$, $r_{GPU} = 0.08ms$. Using these profiling results, we can instantiate the cost models from the previous section, using profiling, to give a minimal $T_{gather} = 0.001ms$ and $T_{dist} = 0.001ms$. The high-level cost models give us an estimated minimal runtime for each configuration, which we can use to filter out weak candidates. Based on the cost model predictions, we isolate the top three configurations as candidates for our MCTS model. These candidates are shown in bold in Figure **??**.

D. Performance Results

All measurements have been made on an 2x 2.4Ghz 12-core AMD Opteron 6176 CPUs coupled an NVidia Tesla C2050 with 448 GPU cores running at 1.16GHz, running Centos Linux 2.6.18-274.el5. and g++ 4.1.2, averaging over 10 runs with an image size of $4096 * 4096$ applied to a stream of images.

Figure **??** shows the actual speedup results for $\Delta(r) \parallel p$ (version 3, from Figure **??**). For this configuration, the second stage of the *Pipe*, p, is a single worker, so only 1 GPU worker is considered. The MCTS predicts for this configuration (from Figure **??**) an optimal mapping allocating 4 CPU workers and 1 GPU worker. As Figure **??** shows, 4 CPU workers and 1 GPU worker gives an actual speedup of 39.14, compared with a best performance speedup of 39.43, for 8 CPU workers and 1 GPU worker, this mapping prediction is within 1% of the best speedup obtained for this configuration. The difference between 4 CPU workers and 8 CPU workers is a minimal speedup of 0.29. This is rejected as a candidate solution since the MCTS also takes into account hardware utilisation. Using an additional 4 CPUs to gain such a small increase in speedup will usually not be worthwhile.

In Figure **??** we show the speedups for $\Delta(r) \parallel \Delta(p)$ (configuration 5, as shown in Figure **??**). The figure shows

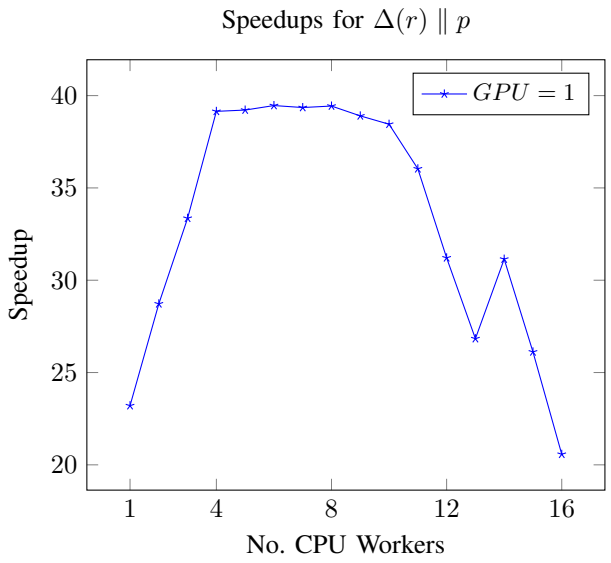

Fig. 6. Speedup graph for configuration 3, for 1 GPU worker

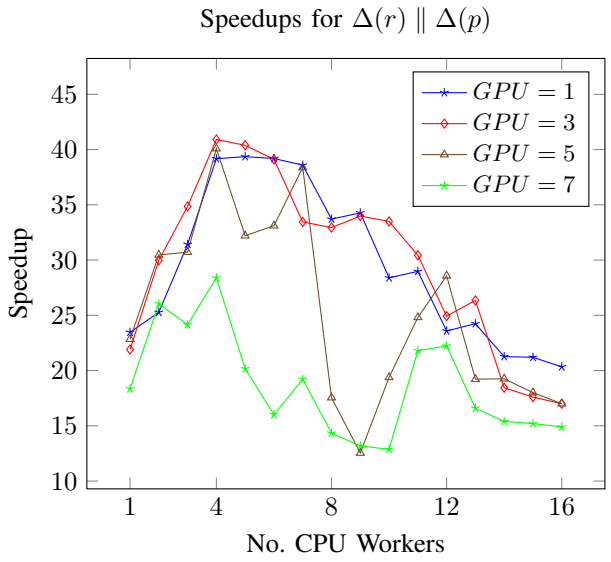

Fig. 7. Speedup figures for configuration 5 for 1,3,5, and 7 number of GPU workers

speedup figures for the configuration with a variety of GPU workers (here, 1,3,5 and 7 GPU workers), where we decided to show the best speedups, where other runs with a different numbers of GPU workers (up to 16 GPU workers) showed much poorer speedup results and were deemed uninteresting. For this configuration, the MCTS prediction (from Figure **??**) shows optimal speedup for 6 CPU workers and 3 GPU workers, however Figure **??** shows, for 6 CPU and 3 GPU workers, a speedup of 39.12, where the best speedup is for 4 CPU and 3 GPU workers, with a speedup of 40.91. This demonstrates that our MCTS prediction for this configuration is within 4% of the best speedup obtained. From Figure **??**, we can also see that from the remaining performance measurements, the best speedup for 1 GPU worker is 39.35 (for 5 CPU workers), 5 GPU workers is 40.08 (for 4 CPU workers) and 7 GPU

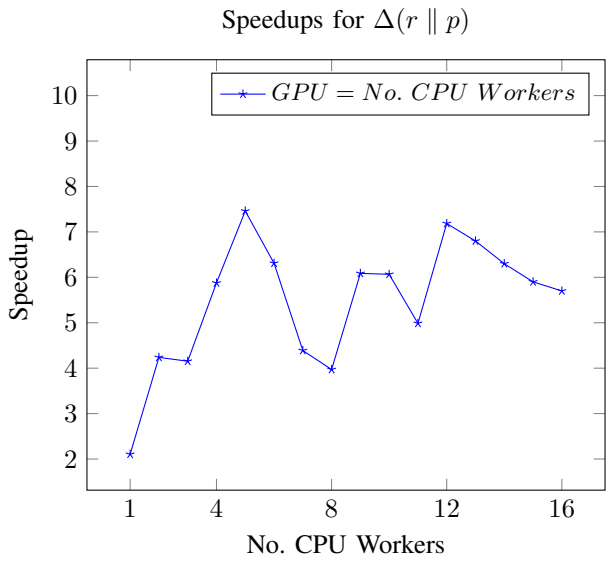

Fig. 8. Speedup figures for configuration 6, where the number GPU workers = number CPU workers

workers is 28.39 (for 4 CPU workers).

Figure **??** shows the speedups for $\Delta(r \parallel p)$ (configuration 6, as shown in Figure **??**). Here we show for each speedup of n CPU workers and also n GPU workers. Other results showed much poorer speedups and we therefore have not shown them here. As Figure **??** demonstrates, the best speedup obtained for this configuration is 7.45 for 5 CPU and 5 GPU workers, which confirms the prediction given by our MCTS model (Figure **??**).

It is noted that the speedup figures here do not always show smooth performance improvements by increasing the number of workers, and there are points where we even get poor performance (such as in Figure **??** for 9 CPU workers and 5 GPU workers). One possible reason for this could be that we have a combination of nodes where each node gives a different service time: increasing the number of workers only to the *bottleneck* stages increases global throughput, and therefore performance. Once the bottleneck has been removed, adding any extra workers to a non-bottleneck stage in FastFlow doesn't always seem to improve the performance, since there is extra overhead in the system from creating the extra thread. FastFlow is designed for fine-grained parallelism, with an assumption that most of the stages are CPU bound. This gives rise to a problem when addressing computations that are assigned to the GPU, where increasing the number of GPU workers can actually harm the performance rather than improve it, especially when the allocation of tasks to the GPU no longer fits into the GPU model. Our MCTS model compensates for this, by finding an optimal result by considering the utilisation of stages as a penalty factor in the system. In addition, the cost model predictions for the configurations, from Figure **??**, predicted that the second most optimal configuration is $\Delta(r \parallel p)$. However, looking at the actual performance measurements, $\Delta(r) \parallel p$ performs better, with an increased speedup of 31.98 over the $\Delta(r \parallel p)$ configuration. This shows that using high-level cost models alone is not enough to predict optimal parallelisations.

VI. Related Work

A. Static Mapping

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 [**?**], [**?**], [**?**], using runtime scheduling [**?**], heuristic-based mappings [**?**], analytical models [**?**], [**?**], or ILP solvers [**?**]. 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 [**?**] or the runtime adaptation approach used in FlexStream [**?**] framework. Also, a recent series of research aims 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 [**?**], [**?**], [**?**]. In [**?**], 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 is primarily focused on application performance tuning [**?**] rather than orchestration. In [**?**] 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 [**?**], [**?**], [**?**], [**?**]. Recently MCTS has been applied to planning and scheduling problems. In [**?**] 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 [**?**] 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 [**?**] 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 techniques introduced here can easily be applied to a different framework by changing the evaluation method. This makes the algorithm framework independent.

C. Refactoring

There has so far been only a limited amount of work on refactoring for parallelism [**?**]. Hammond *et. al* [**?**] used Template Haskell [**?**] with explicit cost models to derive automatic farm skeletons for Eden [**?**]. Unlike the approach presented here, Template-Haskell is compile-time, meaning that the programmer cannot continue to develop and maintain his/her program after the skeleton derivation has taken place. Other work on parallel refactoring has mostly considered loop parallelisation in Fortran [**?**] and Java [**?**]. However, these approaches are limited to concrete structural changes (such as loop unrolling) rather than applying high-level pattern-based rewrites as we have described here. However, it does not use pattern-based rewriting or MCTS based direction, as discussed in this paper. A rich set of skeleton rewriting rules have been studied and proposed in [**?**], [**?**], [**?**], [**?**]. When using skeleton rewriting transformations a set of functionally equivalent programs exploiting different kinds of parallelism is obtained. Cost models and evaluation methodologies have also been proposed that can be used to determine the best of a set of equivalent parallel programs [**?**], [**?**]. The methodology presented in this paper extends and builds on this and similar work by providing refactoring tool-support supplemented by a programming methodology that aims to make structured parallelism more accessible to a wider audience.

VII. Conclusions and Future Work

This paper has described a new methodology that employs new refactoring and static mapping technology. Refactoring allows programmers to refactor their programs, introducing and tuning parallelism, where the choice of the refactoring and hardware mapping is derived automatically. We have described a number of novel refactorings implemented in Eclipse, for C++ and FastFlow, and also described an MCTS algorithm capable of delivering accurate mapping predictions for a heterogenous machine (within 5% of the best speedup obtained). Our convolution benchmark has shown that even for an algorithm with relatively little structure, there would be many different *manual* refactoring choices to choose from. Instead, our approach provides accurate mapping information, and refactoring predictions, outperforming a standard set of cost models, allowing the programmer to concentrate on the the correctness of the application, rather than the parallelisation.

Although our methodology is described in terms of C++ using FastFlow, the approach taken here is, in fact, completely generic. The refactorings and skeleton could easily be carried over to different languages and frameworks, such as Cilk, OpenMP, etc. In particular, the MCTS algorithm is designed to be used with any skeleton framework, being fully parameterised over the overheads in the implementation. Our intention is that we will, in time, develop a generic refactoring and mapping methodology capable of using a common set of refactoring rules and skeletons

In the near future, we expect to extend our methodology to cover a wide range of parallel skeletons including parallel workpools, divide-and-conquer, map-reduce, bulk synchronous

parallelism, and other domain-specific parallel patterns, such as parallel Orbit enumerations. In addition, we intend to demonstrate the use of our methodology on a further set of benchmarks, showing greater skeleton nesting and potential heterogeneity. 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.

ACKNOWLEDGMENT

This work has been supported by the European Union grants IST-2010-248828 "ADVANCE: Asynchronous and Dynamic Virtualisation through performance ANalysis to support Concurrency Engineering", and IST-2011-288570 "ParaPhrase: Parallel Patterns for Adaptive Heterogeneous Multicore Systems", and by the UK's Engineering and Physical Sciences Research Council grant EP/G055181/1 "HPC-GAP: High Performance Computational Algebra."

REFERENCES

[1] 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.

[2] K Ramamritham. Dynamic Task Scheduling in Hard Real-time Distributed Systems. in J. of Software, IEEE 1(3), pages 65–75. 1984.

[3] 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*, pages 126–136. ACM, 2005.

[4] M. Aldinucci, M. Coppola, and M. Danelutto. Rewriting Skeleton Programs: How to Evaluate the Data-Parallel Stream-Parallel Tradeoff. In S. Gorlatch, editor, *Proc of CMPP: Intl. Workshop on Constructive Methods for Parallel Programming*, pages 44–58. Fakultät für mathematik und informatik, Uni. Passau, Germany, May 1998.

[5] M. Aldinucci and M. Danelutto. Stream Parallel Skeleton Optimization. In *Proc. of PDCS: Intl. Conference on Parallel and Distributed Computing and Systems*, pages 955–962, Cambridge, Massachusetts, USA, Nov. 1999. IASTED, ACTA press.

[6] M. Aldinucci, M. Danelutto, P. Kilpatrick, M. Meneghin, and M. Torquati. Accelerating Code on Multi-cores with FastFlow. In *Euro-Par*, pages 170–181, 2011.

[7] M. Aldinucci, S. Gorlatch, C. Lengauer, and S. Pelagatti. Towards Parallel Programming by Transformation: The FAN Skeleton Framework. *Parallel Algorithms and Applications*, 16(2–3):87–121, Mar. 2001.

[8] 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.

[9] D. Caromel and M. Leyton. Fine Tuning Algorithmic Skeletons. In A.-M. Kermarrec, L. Bougé, and T. Priol, editors, *Euro-Par*, volume 4641 of *Lecture Notes in Computer Science*, pages 72–81. Springer, 2007.

[10] G. Chaslot. *Monte-carlo Tree Search*. PhD thesis, Maastricht Univ, 2010.

[11] G. Chaslot, S. De Jong, J. Saito, and J. Uiterwijk. Monte-carlo Tree Search in Production Management Problems. In *Proceedings of the 18th BeNeLux Conference on Artificial Intelligence*, pages 91–98, 2006.

[12] A. Collins, C. Fensch, and H. Leather. Optimization Space Exploration of the Fastflow Parallel Skeleton framework.

[13] A. Couëtoux, J. Hoock, N. Sokolovska, O. Teytaud, and N. Bonnard. Continuous Upper Confidence Trees. *Learning and Intelligent Optimization*, pages 433–445, 2011.

[14] R. Coulom. Efficient Selectivity and Backup Operators in Monte-carlo Tree Search. *Computers and Games*, pages 72–83, 2007.

[15] 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*, 39(11):78–85, 1996.

[16] D. Dig. A Refactoring Approach to Parallelism. *IEEE Softw.*, 28:17–22, January 2011.

[17] S. Gelly and Y. Wang. Exploration Exploitation in Go: Uct for Monte-carlo Go. 2006.

[18] 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*, volume 40, pages 151–162. ACM, 2006.

[19] S. Gorlatch, C. Wedler, and C. Lengauer. Optimization Rules for Programming with Collective Operations. In *Proceedings of the 13th International Symposium on Parallel Processing and the 10th Symposium on Parallel and Distributed Processing*, IPPS '99/SPDP '99, pages 492–499, Washington, DC, USA, 1999. IEEE Computer Society.

[20] K. Hammond, M. Aldinucci, C. Brown, F. Cesarini, M. Danelutto, H. Gonzalez-Velez, P. Kilpatrick, R. Keller, T. Natschlager, and G. Shainer. The ParaPhrase Project: Parallel Patterns for Adaptive Heterogeneous Multicore Systems. FMCO 2012. Submitted, February 2012.

[21] K. Hammond, J. Berthold, and R. Loogen. Automatic Skeletons in Template Haskell. *Parallel Processing Letters*, 13(3):413–424, September 2003.

[22] 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*, pages 214–223. IEEE, 2009.

[23] L. Kocsis and C. Szepesvári. Bandit based Monte-carlo Planning. *Machine Learning: ECML 2006*, pages 282–293, 2006.

[24] M. Kudlur and S. Mahlke. Orchestrating the Execution of Stream Programs on Multicore platforms. In *ACM SIGPLAN Notices*, volume 43, pages 114–124. ACM, 2008.

[25] Y.-K. Kwok and I. Ahmad. Static Scheduling Algorithms for Allocating Directed Task Graphs to Multiprocessors. *ACM Computing Surveys (CSUR)*, 31(4):406–471, 1999.

[26] R. Loogen, Y. Ortega-Mallén, and R. Peña-Marí. Parallel Functional Programming in Eden. *Journal of Functional Programming*, 15(3):431–475, 2005.

[27] H. Gonzalez-Valez, and M.Goli. Heterogeneous Algorithmic Skeletons for Fastflow with Seamless Coordination Over Hybrid Architectures. In *21st Euromicro International Conference on Parallel, Distributed and Network-Based Processing.* IEEE, 2013.

[28] H. Nakhost and M. Müller. Monte-carlo Exploration for Deterministic Planning. In *IJCAI*, pages 1766–1771, 2009.

[29] 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*, pages 281–290. IEEE, 2009.

[30] S. Pelagatti. *Structured Development of Parallel Programs*. Taylor and Francis, 1999.

[31] 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*, pages 271–271. ACM, 2007.

[32] T. Sheard and S. P. Jones. Template Meta-Programming for Haskell. *SIGPLAN Not.*, 37:60–75, December 2002.

[33] D. B. Skillicorn and W. Cai. A Cost Calculus for Parallel Functional Programming. *J. Parallel Distrib. Comput.*, 28(1):65–83, 1995.

[34] J. Subhlok, J. M. Stichnoth, D. R. O'hallaron, and T. Gross. Exploiting Task and Data Parallelism on a Multicomputer. *ACM SIGPLAN Notices*, 28(7):13–22, 1993.

[35] 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, pages 200–209. IEEE, 2009.

[36] Z. Wang. *Machine Learning Based Mapping of Data and Streaming Parallelism to Multi-cores*. PhD thesis, PhD thesis, The University of Edinburgh, 2011.

[37] 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*, pages 307–318. ACM, 2010.

[38] J. Wloka, M. Sridharan, and F. Tip. Refactoring for Reentrancy. In *ESEC/FSE '09*, pages 173–182, Amsterdam, 2009. ACM.

[39] C. Ykman-Couvreur. Exploration Framework for Run-time Resource Management of Embedded Multi-core Platforms. In *Embedded Computer Systems (SAMOS), 2010 International Conference on*, pages 333–340. IEEE, 2010.

[40] 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*, 5(2):123–135, 2011.

[41] 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*, 1(2):120–128, 2007.