A Framework for Performance-aware Composition of Applications for GPU-based Systems

Usman Dastgeer
Dept. of Computer and Information Science
Linköping University
Email: usman.dastgeer@liu.se

Christoph Kessler
Dept. of Computer and Information Science
Linköping University
Email: christoph.kessler@liu.se

Abstract—User-level components of applications can be made performance-aware by annotating them with performance model and other metadata. We present a component model and a composition framework for the performance-aware composition of applications for modern GPU-based systems from such components, which may expose multiple implementation variants. The framework targets the composition problem in an integrated manner, with particular focus on global performance-aware composition across multiple invocations. We demonstrate several key features of our framework relating to performance-aware composition including implementation selection, both with performance characteristics being known (or learned) beforehand as well as cases when they are learned at runtime. We also demonstrate hybrid execution capabilities of our framework on real applications. Furthermore, as an important step towards global composition, we present a bulk composition technique that can make better composition decisions by considering information about upcoming calls along with data flow information extracted from the source program by static analysis, thus improving over the traditional greedy performance-aware policy that only considers the current call for optimization.

Keywords—Global composition; implementation selection; hybrid execution; GPU-based systems; performance portability;

I. INTRODUCTION

Modern GPU-based systems, containing multicore CPUs and one or more programmable GPUs with disjoint physical memory, expose parallelism and heterogeneity in today’s computing. On the software side, both heterogeneity and parallelism result in programming and performance problems [1]. Programming is difficult as the programmer often needs to use different programming models to program CPU and GPU devices present in the system. Besides the programming issue, architectural differences between and across these CPU and GPU devices make getting and sustaining performance across different systems a non-trivial task.

In recent years, significant research work has been carried out to tackle this complex problem. One important part of this multi-faceted problem arises from the possibility of doing computation work on either of these devices. For example, there exist different sorting implementations for both CPUs and GPUs that can provide the same functionality. With modern GPUs getting more general-purpose and better programmable with time, more and more applications are facing this choice. Choosing which implementation to use for a given execution context is known as the implementation selection problem\(^1\).

Implementation selection is not new to mainstream computing. In the sequential homogeneous computing paradigm, implementation selection normally considers which implementation is (expected to be) the fastest one for a certain input among all implementation variants. The decision in this case could solely be made or prepared before actual execution (e.g., by building performance models from training executions). In GPU-based and other heterogeneous systems, as implementation variants are possibly written for different types of compute units, the selection of the fastest implementation variant is also concerned with resource allocation and run-time (system workload, data locality) information. For example, suppose that we have two (1 CPU and 1 CUDA GPU) implementation variants to perform same computation, and suppose we know before actual execution that the GPU variant performs better on the target system for a particular execution context. However, it may happen that the GPU is already busy or the data is not yet located in the GPU memory; in this case, executing the suboptimal CPU variant may actually be a better decision, i.e., result in overall shorter execution time.

There exist multiple aspects of this implementation selection problem. Some applications can rely on performance models calibrated via earlier executions [2] (or have analytical models [3]) while others require online learning with no previously calibrated performance models [4]. Also, an application can have more than one type of (componentized) computation, with arbitrary data dependency and resource state flow that can result in more complex scheduling and selection tradeoffs. A greedy, locally optimal implementation selection in this case, may result in a sub-optimal global solution. Furthermore, instead of selecting which implementation to use on which computing device, some computations may benefit from the possibility of simultaneously using multiple (or all) computing devices (CPUs and GPUs) by dividing the computation work into parts and processing them in parallel on different devices. Data parallel computations can especially benefit from this hybrid execution capability.

In this work, we propose an integrated Global Composition Framework (GCF) that aims at addressing the implementation selection problem for heterogeneous systems in a variety of contexts. To the best of our knowledge, our framework is the first implementation selection framework for modern GPU-based systems that targets both local and global composition while relying on existing well-established programming models (OpenMP, PThreads, CUDA, OpenCL etc.) rather than proposing any new programming model (like [5]–[9]).

\(^{1}\)In this article, we use the terms implementation, implementation variant and variant interchangeably.
Specifically, we make the following contributions:

- An integrated Global Composition Framework (GCF) based on the ROSE source-to-source compiler that can do performance-aware composition for modern heterogeneous GPU-based systems in an automated and portable manner.
- A component model that defines the notion of component, interface and implementation variants along with specifications of annotations for extra meta-data.
- A generic performance modeling API that enables usage of both analytical and empirical performance models while supporting online feedback about runtime execution information to the composition framework.
- A global bulk scheduling heuristic for composing multiple component calls constrained in a data dependency chain.
- Evaluation of different composition capabilities of our GCF framework with real applications taken from RODINIA benchmarks and other sources, showing implementation selection, hybrid execution, online tuning as well as global composition capabilities of our framework on two different GPU-based systems.

Although discussed for GPU-based systems in this paper, both the designed framework and the component model are generic and can be used whenever doing implementation selection, e.g. on homogeneous CMP systems, and even can easily be extended to other heterogeneous systems in the future.

This paper is structured as follows: Section II describes central concepts of the GCF component model. Section III describes the proposed framework in detail. Section IV presents an evaluation of the developed framework with different applications. Some related work is listed in Section V. Section VI concludes and suggests future work.

II. GCF COMPONENT MODEL

The GCF component model is partially influenced by the PEPPHER component model, which we have designed and implemented in our earlier work [10].

A. Components, interfaces and implementation variants

A GCF component consists of an interface that describes a functionality and multiple implementations or implementation variants of that interface that actually implement the functionality.

Both interface and implementations have attached meta-data, as shown in Figure 1, which provides extra information. In the PEPPHER component model, the meta-data is provided externally in the form of an XML descriptor file; this is non-intrusive to source-code but requires some effort of writing and maintaining XML files. Moreover, mapping XML information to specific locations in the source code is highly vulnerable in case the source-code gets modified. In the GCF component model, we use pragmas and comments to represent meta-information, which is more flexible and easy to write and maintain during code evolution. The component model is currently realized in C/C++ language.

In C/C++, an interface is represented by a C/C++ function declaration that describes the functionality. If not specified otherwise, the function name becomes the name of the interface. The meta-information for an interface includes:

- Access mode of each parameter (is it read, written or both inside the component).
- Relationship between different parameters (e.g., one parameter may describe the size of another parameter).
- Performance model associated with the interface.

Figure 2 shows annotations for interface and implementation meta-data. We use source comments and pragmas to represent interface and implementation meta-data respectively.

The performance model can be specified at both interface and implementation-variant level. A performance model specified for an interface will be shared by all variants of that interface and will supersede individual performance models of any variant if they exist. As we will see later, the ability to specify performance models at different levels gives us the flexibility to effectively address different application needs.

B. Composition Technique

Composition is the selection of a specific implementation variant for a call to component-provided functionality and the allocation of resources for its execution. It is made context-aware for performance optimization and can be done in one or
several stages. By *static composition*, preselection of a subset of appropriate implementation variants is made depending upon the statically available information about target system and call context. The final selection among the remaining component variants is then done at runtime (*dynamic composition*).

Composition points are restricted to calls on general-purpose execution units only. Consequently, all component implementations using hardware accelerators such as GPUs must be wrapped in CPU code containing a platform-specific call to the accelerator. A GCF component call is non-preemptive and may be translated to a task for a runtime system. Components are stateless; however, the parameter data that they operate on do have state.

The composition points can also be annotated with extra information relevant to the call context. This includes information about constraints on the execution (e.g., execution *must* be carried out on a GPU) or on the call properties (e.g., limit on problem size). Figure 3a shows annotation syntax for a component call.

As mentioned earlier, a component implementation in the GCF component model assumes operand data to be present in default memory associated with that execution platform. Whether and when to perform the possibly required operand data transfer is not hardcoded but exposed to the composition framework, as this is an important optimization aspect in a heterogeneous system containing multiple execution platforms with disjoint memory address spaces. For example, factoring out the GPU data transfers from a GPU implementation yields two important benefits. First, it avoids the overhead of data transfers for each component invocation which can have a serious impact on performance [11]. The data transfers can thus be optimized across multiple component invocations. Secondly, it enables further optimizations in data transfers (e.g., overlapping communication with computation) without requiring any changes in the implementation code.

This however requires runtime support for (in-time) data transfers to ensure that data is available in the right address space when an implementation gets called. The runtime support would *place* (i.e., schedule) data transfers (and other data handling code) at the right place to ensure that data is only communicated when really needed while ensuring data consistency and correctness of execution. In our case, we build a static data access analyzer to analyze the program data flow to find out where to *place* data handling code (more on this in the next section). As shown in Figure 3b, we also provide annotations to assist the data access analyzer in cases where data access may not be accurately found out using static analysis.

### III. GLOBAL COMPOSITION FRAMEWORK

Figure 4 shows different components of our framework along with their interaction. Following are the five major parts of the framework.

1. A component repository manager (*gcfmanager*) with support for both registering new components (interfaces, variants, performance models) as well as managing already registered components.
2. A component tree builder (*builder*) that, based on the ROSE compiler [12], parses the application source code AST (Abstract Syntax Tree) with one or more component calls and analyses data accesses for component operand data to build a component tree.
3. A *composer* which parses the component tree along with the program control flow obtained via the ROSE compiler to generate the composition code. As a component call is made by interface name, a wrapper (proxy) function is generated to intercept the call and internally delegate the call to a certain implementation variant. The composer is generic in the underlying run-time system; currently, the composition code can be generated for both the GCF runtime library, which we have developed in this work, as well as for the StarPU runtime system [13].
4. A performance modeling API that specifies a generic and extensible interaction interface between the performance models and the runtime library.
5. The GCF runtime library that can do performance-aware composition by using the
performance modeling API along with data handling for component operand data.

In the following we explain each part in more detail.

A. Componentization and Component Repository

For practical usage, a component model must be complemented with a mechanism for managing component information in an intuitive and scalable manner. We have provided the tool gcfmanager that can be used to manage the component repository. It has a simple interface, as shown in Listing 1, and can be used to register and manage both interfaces and implementation variants. It internally stores information about registered interface and implementation variants at a specific place specified at the installation time.

Registering a new interface is done by specifying a path to a file containing the interface declaration with annotations marked as comments. For example, the following command registers an interface of a component from a SPH (Smoothed Particle Hydro-dynamics) application:

```
$ gcfmanager -i sph_update_position.h
```

where the interface declaration in file looks as follows:

```c
void sph_update_position(Particle *fluidArr /* accessMode(readwrite) size(arrSize) */, int arrSize /* accessMode(read) usedInPerfModel */);
```

Similarly an implementation of the interface is registered in the following way:

```
$ gcfmanager -v sph_cuda_update_position.cu
```

where the implementation definition in file looks as follows:

```c
#pragma gcf variant target(gpu)
interface(sph_update_position) type(cuda)
perfmodel(pm_sph_update_position_cuda.h)
void sph_update_position_cuda(Particle *fluidArr, int arrSize) {
    ...
}
```

B. Component tree builder

As shown in Figure 4, the input program AST contains one or more calls to GCF components which operate on certain operand data. As implementation variants assume that operand data reside in their memory address space, certain data handling must be carried out to ensure that data is available at the right place at the right time. As data handling is carried out by the runtime system, we need to place data handling calls for the runtime system at appropriate places. We achieve this by analyzing the program control and data flow using the ROSE compiler. We have written a custom pass that provides this functionality; it works in the following manner.

```
1 $ gcfmanager -h
2 The global composition framework (GCF) manager
3 can be used to manage GCF component repository.
4
5 $ Usage:
6 gcfmanager [options] <rose-options>
7
8 $ options:
9   -h	Shows this message.
10  -i <filename> Register a new interface.
11  -v <filename> Register a new variant.
12  -u <inf-name> ... Update existing interface.
13  -r <inf-name> Remove existing interface.
14  -l <inf-name> ... Remove existing variant.
15  -l <inf-name> List all interfaces.
16  -L <inf-name> List all interface variants.
17  -p <inf-name> Shows interface' performance model.
18  -P <inf-name> Shows variant' performance model.
20
21 * = option names are case-insensitive.
```

Listing 1. The gcfmanager options.

First, all component calls in the application code are collected and a list of all variables is maintained that are used (as operands) inside one or more component calls. The only exception is scalar variables that are passed by value (e.g., the size variable in the input code in Figure 4)\(^2\). The data access analysis is then carried out for each variable in the list. We record data creation, accesses (read and/or written in non-component code) and usage inside a component call. Currently, data usage for array variables is recorded at the variable level. This means that, for example, a read access to a certain array element is considered a read operation on the complete array. This simplifies data flow analysis and data handling code that is inserted [14], [15]. We analyze data accesses in a recursive

\(^2\)We don’t need any explicit data handling for these scalar passed-by-value variables as inside components, a copy of original contents gets passed.
manner\(^3\) for variables that are passed by reference or pointer to non-component function calls by tracing their usage inside that function body. The analysis carried out is conservative to ensure program correctness. However, as the output of our framework is a modified source file with extra C/C++ source code added, the user can verify and debug the output source code for error-checking. As a fallback solution, we introduce explicit data annotations (see Figure 3b) that allow the user to assist data analysis for complex scenarios\(^4\).

As per now, data usages are correctly tracked for regular C data types including pointers as well as any combination of C composite types using structures (e.g., structure with array member variables, array of structures etc.). For composite types, if a specific member variable is used inside a component call, the access can be separately tracked only for that member variable. For an array member variable, data usage is tracked at the array level, as said previously.

At this level, all these data operations and component calls are recorded in an AST tree representation which makes data analysis and optimizations simple to carry out. However, using the ROSE compiler, data operations recorded in the tree can be traced back to their location in the program control flow graph. In the next stage, this tree is complemented by the program control flow graph to insert data handling code at the appropriate places in the program control flow.

C. Composer

The composer takes the component tree generated by the component tree builder as input along with the program control flow graph. By mapping operations listed in the component tree to their source in the program control flow graph, it generates the actual composition code for a runtime system. Currently, we support composition using either our own GCF runtime library or using the StarPU runtime system. The choice of a runtime system is specified by the user as a simple switch\(^5\) when doing the composition. Recursive components are not supported yet; partially because the StarPU \([13]\) system does not support them.

At this stage, certain optimizations are carried out at the component tree (i.e. not in the actual source code) with the help of the program control flow graph to simplify the process of placing data handling code. For example, if we have two write accesses to the same variable in the same (or nested) scope without any component call in between, they are considered as one write access. Similarly, if we have one read access followed by a write access to the same variable with no component call in between, we can consider them a single readwrite access instead.

Finding appropriate places for placing data handling code becomes quite tricky in a program with complex control flow. For instance, the data accesses and component calls can be placed in a way that it preserves the program correctness. The composer follows the following rules when placing the data handling code:

- Calls for registering\(^6\) and unregistering a variable to the runtime system are placed at the same scope level. In case component calls and data accesses span over multiple non-nested scopes, calls can be placed at their immediate common parent scope.
- Placing data handling code inside a conditional statement is avoided, when possible, to simplify execution flow. When not possible, the code may need to be replicated across different control flows (e.g., if-else statement).
- Placing code inside loop constructs is avoided, when possible, to optimize data transfers. For example, if a variable is read inside a loop and no component call using that data is present inside the same (or nested) loop, the necessary code to ensure that data is available for the read purpose is placed just before the loop body to avoid the overhead of executing that code for each loop iteration.

In the following, we will briefly describe both runtime environments that we currently support for composition.

1) StarPU: StarPU \([13]\) is a C based unified runtime system for heterogeneous multicore platforms, especially for GPU-based systems. It supports the explicit notion of CPU and GPU workers, a data management API as well as multiple scheduling policies to decide which implementation to execute at which computing device for a given execution context. It can use the information about runtime workload of each worker to do dynamic load balancing and can effectively use multiple CPU and GPU devices in parallel. We use this capability of StarPU to support hybrid execution in our framework. Moreover, it has a performance-aware scheduling policy which considers information about current system workload and data locality, and uses estimate of data transfer cost as well as previous historical execution information to decide which implementation would be faster.

2) GCF runtime library: To get more control over runtime decisions, we have developed a simple and light-weight C++ based runtime library. Currently, the library is designed for doing performance-aware implementation selection where selection is mainly concerned with which implementation variant to use at which execution platform\(^7\). The main design goals of this runtime library are to be:

- **light-weight**: We want to reduce the runtime overhead to the minimum possible extent. Our library considers all CPUs as one combined worker, instead of creating one CPU worker for each CPU as StarPU does.
- **configurable**: The library uses a configurable performance modeling API which allows it to be used for applications with different kinds of performance

\(^3\)We consider a recursive function as a special scenario to avoid combinatorial explosion problem of the solution space. General interprocedural optimization of composition is an issue of future work.

\(^4\)We have not encountered any such scenario yet in any application that we have ported to our framework.

\(^5\)By default, the system generates composition code for our own GCF runtime library. The user can set the `-starpu` switch to generate code for the StarPU runtime system.

\(^6\)Registering a variable to the runtime system creates a unique data handle (with information about size, memory address etc.) for that data in the runtime system which can be used for controlling its state and data transfers.

\(^7\)In future, we may extend the GCF runtime library to support asynchronous component executions like StarPU.
models. By specifying his/her own performance models (empirical, analytical etc.), the user can control implementation selection in an effective manner.

There exists naturally some overlap in functionality offered by both runtimes; however there are several major differences. The GCF runtime library supports a configurable performance modeling API with multiple types of performance models available; it also allows a user to easily plug any new performance model with his/her components. Also, the GCF library mainly targets performance-aware implementation selection. This is more of an offloading decision, i.e., making a choice between using CPU (e.g. OpenMP) and GPU (e.g. CUDA) rather than simultaneous usage of both employed by StarPU, i.e., unlike StarPU, the GCF library does not currently support concurrent component executions on CPU and GPU devices in parallel.

D. Performance Modelling API

Considering that no single performance model can suit every application need, we have designed a generic API to handle interaction between a performance model and the runtime library, using C++ classes and polymorphism. Listing 2 shows the abstract performance model (PerfModel) interface which all concrete performance models must inherit. The performance-relevant call context properties are represented by an array of integers. This allows to model any number and kind of performance-relevant properties (e.g., problem size(s), sparsity structure, degree of sortedness of data etc.). Careful selection of performance-relevant properties is important to achieve good prediction accuracy. The PerfModel interface can be used to specify either of the following two kinds of performance models:

1) A **Per-variant performance model** targets a single implementation variant. In this kind of performance model, the prediction function `predict` returns a floating-point cost value. A common example of such kind of performance model is the historical execution based performance model used by the StarPU runtime system [2]. For now, we are concerned with performance in terms of execution time so the return value is predicted execution time. However, this could easily be used for other performance objectives such as energy consumption or code-size.

2) A **Per-interface performance model** is common for all variants implementing an interface. This kind of model is at a higher granularity level as the performance model internally handles the choice between different implementations. In this case, the prediction function `predict` returns the expected best implementation name, managing the complete selection process internally. Many performance models can fit here, including models based upon decision tree, SVM and one presented in [16].

When comparing different variants for implementation selection, potential data transfer cost must also be considered besides the computation cost predicted for each variant. For a CUDA implementation variant, the potential data transfer cost is added to its computation cost if input data is not already present in the target GPU memory. The same should be done for CPU or OpenMP implementations if the most updated data resides in some GPU memory. The implementation selection can be configured in two modes. (1) In **pessimistic** mode it considers a component call as a standalone call. The data transfer cost for a variant is estimated assuming that data must be made available in the main memory after the component call. This for example could mean that, for a GPU variant, the cost of both transferring input operands from main memory to GPU memory (if not present already) as well as the cost for transferring modified data back to the main memory must be added to its computation cost. (2) In **optimistic** mode, data transfer cost is estimated by assuming that a component call is part of multiple component calls using that data. The data transfer cost, in this case, is only estimated for input operand data. The pessimistic and optimistic modes are suitable for an application with single and multiple component calls using a certain data respectively. The modes can be automatically determined by the framework by its component tree and control flow analysis.

We have provided a simple library that can estimate data transfer cost given the number of bytes, and source and destination memory unit. For each architecture, the library builds a simple model using latency (`latency`) and average cost of transferring one byte (`costPerByte`) determined by actual measurements on that architecture. Afterwards, it uses a simple formula for predicting data transfer cost between two memory units `a, b` for `B` bytes in direction `a → b`:

\[
\text{time} = \text{latency}[a][b] + \text{costPerByte}[a][b] * B.
\]

The framework not only uses the performance model to do prediction (using the `predict` method) but also feeds actual measured cost value (execution time in our case) back to the performance model used by the `addEntry` method. This enables runtime calibration where the performance model can learn/improve and predict during the same program execution. One example is online learning where the performance model does not have any previous calibration data available and thus learns and predicts at runtime. The performance models can internally control switching between calibration and prediction mode. One key feature of the API is its genericity as it does not require an empirical model and can work for, e.g., analytical models such as [3], [17].

The process works as follows: The framework creates references to the performance model for each component interface/variant used inside the application during the initialization phase. The performance model can internally load previously

```
1 class PerfModel
2 {
3   /* for per variant performance models */
4   virtual double predict(int keys[], int numKeys);
5   virtual void addEntry(int keys[], int numKeys, double timeVal);
6   /* for per interface performance models */
7   virtual string predict(int keys[], int numKeys, double *commCost);
8   virtual void addEntry(int keys[], int numKeys, string impl, double timeVal);
9   /* common for both kinds */
10  virtual void close();
11  }
```

Listing 2. The PerfModel interface.
calibrated information if any exists. During the actual program execution the performance model is used to predict the best variant as well as the actual execution data is fed back to the performance model. At termination, the framework calls the close method for each performance model. This allows a performance model to persist calibration information for future usage. Currently, the following four performance models are implemented in our framework. All models can learn and predict both within a program execution or across different program executions.

1) Exact entry model: This performance model targets a single variant and is designed for numerical computations and other applications where few combinations of performance-relevant properties get repeated over different executions. The performance model is similar to the history-based performance model in StarPU. It predicts execution time if an exact match is found with calibration data recorded earlier. Otherwise it switches to calibration mode to gather information for possible future prediction.

2) Euclidean normalized model: This performance model also targets a single variant and is more flexible (but usually less accurate [2]) than the previous model as it can predict for intermediate points based on measured points using Euclidean distance where the Euclidean distance between any two points \( a \) and \( b \) is the length of the line segment connecting them \( \langle ab \rangle \). The performance relevant properties are normalized to the same scale before calculating the euclidean distance.

3) Convex model: This model is at the interface level and thus internally manages selection between different variants. It is based on the model from Kicherer et al. [16]. The model aims for execution contexts with limited or no repetitions in performance relevant properties (i.e., almost every execution has a different value for performance relevant properties). When no calibration data is found, it starts calibration by trying out different variants in a round-robin fashion and collecting their execution time. When a reasonable amount of performance data is collected it builds the initial performance model in the following way: Considering that measured points\(^8\) for different variants would be possibly different, it takes the union of all recorded points for the different variants. It then interpolates measured points for each variant to get potential cost values for missing points in the union. The optimal variant at each point in the union is then recorded by comparing the potential cost values (along with data transfer costs) of each variant at that point. This model is further optimized by making a convexity assumption that if a single variant is found best for two neighboring points, then the same variant is predicted to be best for intermediate points [18].

4) Generic model: This model does not have any inherent semantics but rather relies on the user to provide an actual implementation. It can be used to plug in any performance model by specifying function pointers for the predict, addEntry and close methods. Two variations of this model are designed for plugging-in per-interface and per-variant performance models respectively.

\(^8\)A point represents a single execution with a certain performance relevant properties.

### IV. Evaluation

For evaluation, we implemented seven applications from the RODINIA benchmark suite [19], two scientific kernels (sgemm and spmv), and several other applications from different domains (image processing, fluid dynamics etc.). Two evaluation platforms are used: System A with Xeon\(^R\) E5520 CPUs running at 2.27GHz with 1 NVIDIA\(^R\) C2050 GPU with L1/L2 cache support. System B with Xeon\(^R\) X5550 CPUs running at 2.67GHz with a lower-end GPU (NVIDIA\(^R\) C1060 GPU) is used for showing performance portability.

#### A. Implementation selection

Figure 5 shows execution of several applications over multiple problem instances on two different GPU-based systems. The main decision here is to choose the implementation that performs better in a given execution context (application, architecture and problem instances etc.). As shown in the figure, different implementations can perform better for the same application on a given architecture but at different problem sizes (e.g. pathfinder, convolution) or the choice can be different between different architectures for same application and problem sizes (e.g. nw.b). The performance-aware implementation selection effectively adjusts to these differences without requiring any modifications in the user code.

#### B. Online learning

There exist some applications where the same operation is applied multiple times over different operand data. Examples of such bulk operations include applying sorting, image filters, or compressions/de-compressions over multiple files in a loop. In these applications, performance models can be learned at runtime [4], [16]. Our framework provides support for online feedback which allows performance models to be learned/improved during a program execution. Here, we show two applications (sorting, image convolution) to demonstrate online learning capabilities of our framework.

<table>
<thead>
<tr>
<th>System A</th>
<th>System B</th>
</tr>
</thead>
<tbody>
<tr>
<td>Quick sort CPU</td>
<td>3444.96</td>
</tr>
<tr>
<td>Radix sort CUDA</td>
<td>1134.25</td>
</tr>
<tr>
<td>TGPA</td>
<td>997.24</td>
</tr>
</tbody>
</table>

### Table I. Online learning - sorting (2000 files).

<table>
<thead>
<tr>
<th>System A</th>
<th>System B</th>
</tr>
</thead>
<tbody>
<tr>
<td>Convolution OpenMP</td>
<td>12276.79</td>
</tr>
<tr>
<td>Convolution CUDA</td>
<td>4997.54</td>
</tr>
<tr>
<td>TGPA</td>
<td>4467.44</td>
</tr>
</tbody>
</table>

### Table II. Online learning - image convolution (500 images)
applications, we can see that our tool-generated performance-aware code can effectively learn completely online without any previous performance data being available on both systems and can perform up to 20% better than the best performing version.

C. Hybrid execution

An implementation variant of a component can internally specify parallelism, e.g., by using OpenMP and Pthreads for CPUs and CUDA for GPUs. However, the implementation is still bound to either CPUs or GPUs. For certain computations, more parallelism can be spawned from a single component invocation by partitioning and dividing the work into several chunks that all can be processed concurrently, possibly on different devices by different implementation variants. This feature is implemented in our framework for data-parallel computations where the final result can be produced either by simple concatenation or simple reduction (using plus, max etc. operator) of intermediate output results produced by each subpart (e.g. blocked matrix multiplication, dotproduct). When applicable, this feature can be used in a transparent manner (controlled using the partition option in the interface annotations, see Figure 2) for an application, as it does not require any modification in the user code and/or component implementations.

Figure 6 shows how this feature can help in achieving better performance, for different applications, than using any one of the OpenMP and CUDA implementation variants. This feature is enabled with the help of the StarPU runtime system as our runtime library currently does not support this feature yet\(^3\). More than 100% performance gain is achieved for some applications over the best variant on both systems. This (super-linear) speedup is achievable considering that dividing the computation work of a GPU with multicore CPUs not only divides the computation work but also reduces the communication overhead associated with GPU execution [11].

Besides giving performance improvements, this technique can also enable execution with much larger problem sizes than

\(^3\)Switching between composition using StarPU and our runtime library does not require any change in the user code. It is controlled by just a simple switch to our composition tool (gcfcompose).

D. Global composition

The global composition is a multi-faceted problem involving resource allocation, (in-)dependence between component calls, program control flow as well as performance characteristics of each component [20]. To approach global composition for GPU-based systems in a practical manner, we propose partitioning a program containing multiple component calls into subgroups of component calls based on a grouping construct. Initially, we propose two simple grouping constructs.

A farm group of one or more component calls can be formed when those component calls are independent and there exist no data accesses (that require synchronization) in the user-code confined within those component calls. Similarly, component calls constrained in a data dependency chain (i.e. no parallelism) can be grouped together into one group (a sequence group).

By grouping component calls into sub-groups of different coordination types with well-defined semantics, we can reduce the complexity of the composition problem. As our framework analyses the component calls in a program along with data accesses and control flow, we can find out such farm and sequence groups of calls automatically. For composing farm calls, the runtime system (StarPU etc.) can provide a scalable and practical solution. Even without any performance characteristics known beforehand, a runtime system can execute independent calls effectively by employing dynamic load-balancing and work-stealing techniques. However, this requires the application to submit component calls in an asynchronous manner to the runtime system which expose concurrency to the application programmer. Our framework can transparently execute component calls asynchronously using the StarPU runtime system while ensuring data consistency in the main program by placing data handling (implicit synchronization) code at appropriate places.

For the sequence calls, the StarPU runtime system can make poor decisions as it considers a single component call with no knowledge of dependent component calls that are
about to come. This greedy selection/scheduling could result in overall sub-optimal decisions in the presence of multiple sequence component calls. For example, when considering just the first call alone, we may find it better to execute it on CPUs because of the data transfer cost to GPU. However, if we consider and know that there are other component calls after this which also operate on same data, going to GPU at start would perhaps be a better decision overall. In the following, we demonstrate one such sequence calls scenario in context of modern GPU-based systems along with our proposed composition technique for such scenarios.

Listing 3. Lbsolve source code with component calls.

Listing 3 shows a code portion of a Runge-Kutta ODE Solver from the LibSolve library [21] containing multiple component calls with data dependency between them. One choice could be considering all possible call combinations; however, this could result in big overhead especially considering that the decision is done at runtime. For example, for seven component calls with each call having just two options (OpenMP, CUDA), there exist 128 different possible combinations. Considering this and the fact that data transfers between different memory units are normally expensive, we propose bulk performance-aware composition for such a sequence calls group that only considers combinations possible within implementation variants with same memory address space.

In this case, as OpenMP and CUDA variants have different address space, there exist only two choices; either execute everything on 1) OpenMP or 2) CUDA. By using bulk composition, as shown in Figure 7, we were able to perform better than pure local scheduling policy that considers a single component call at a time. The figure also shows that over different problem sizes, the choice between using OpenMP and CUDA also changes, which the bulk composition was effectively able to adjust.

In some applications, interaction (or switching) between farm and sequence construct can be more complex. In future work, we will investigate such scenarios between different group constructs as well as extend our approach by adding new group constructs.

V. RELATED WORK

Kicherer et al. [16], [22] consider implementation selection (no hybrid execution) for a single call with GPU data transfers as part of CUDA variants which can have serious performance implications. PetaBricks [5], EXOCHI [6], Merge [7], IBM’s Liquid Metal [8] and Elastic Computing [9] propose/rely on a unified programming model/API for programming. In contrast, we do not propose any new programming model but rather support established programming models/APIs (OpenMP, PThreads, CUDA, OpenCL etc.) and can easily support new programming models in future. STAPL [23] uses machine learning techniques to do algorithmic selection.

In PEPPHER [24], we have also studied implementation

Fig. 6. Performance benefit of hybrid execution by using our tool-generated performance aware (TGPA) code with respect to OpenMP and CUDA implementation variants. This feature is enabled using the StarPU runtime system. The baseline is hybrid execution time.

Fig. 7. Difference between performance-aware bulk (global) composition and performance-aware local composition for ODE solver component calls [21].
selection for a single component call by using XML to represent metadata (i.e. no program analysis) and StarPU runtime system. The current framework provides much richer capabilities for doing (both local and global) composition by using program analysis capabilities of the ROSE compiler and a light-weight runtime library coupled with a flexible performance modeling API. Recently, many directive based approaches [25], [26] for GPU programming are introduced, targeting automatic generation of GPU code from an annotated source code. This is different and somewhat complementary to our work as we consider composition problem between different implementation variants rather than automatic code-generation for GPUs.

The context-aware composition [17], [20] is a global optimization technique where each implementation variant specifies its execution constraints (resource requirements etc.) and the composition system tries to efficiently map component calls to given system resources. It relies on performance models provided by the programmer and statically computes dispatch tables for different execution contexts using interleaved dynamic programming. In GPU-based systems that we target, such scheduling decisions could be efficiently made with the help of a runtime system (StarPU etc.) by doing asynchronous component executions. If significant information is known beforehand, the global composition problem can be formulated and solved as an ILP problem, as Eriksson and Kessler [27] did for integrated code generation.

VI. CONCLUSIONS

We have presented a component model and a global composition framework that addresses the implementation selection problem in an integrated manner. We have shown its effectiveness for GPU-based systems with the help of several application, by showing: (1) automatic implementation selection for a given execution context for cases with pre-calibrated models to one with pure online learning; (2) hybrid execution capabilities for data parallel computations, giving up to two times performance improvement for some applications over the best performing variant; and (3) bulk scheduling and its effectiveness with a set of different component calls with data dependency between them.

Although considered for GPU based system in this work, both the proposed component model and the framework (performance modeling API etc.) are generic and can be used in other implementation selection contexts with some more work. There exists several future work possibilities: the implementation can be improved to do more comprehensive program analysis to extract more information for decision making. In future work, global composition will be extended with more groups constructs as well as more global composition scenarios will be considered. Runtime library can also be extended as well as more runtime systems can be supported for composition.

REFERENCES
