Towards global composition of performance-aware components for GPU-based systems

Usman Dastgeer and Christoph Kessler

IDA, Linköping University, 58183 Linköping, Sweden.
<firstname>.<lastname>@liu.se

Abstract

An important program optimization especially for heterogeneous parallel systems is performance-aware implementation selection which is (static or dynamic) selection between multiple implementation variants for the same computation, depending on the current execution context (such as currently available resources or performance-affecting parameter values)\(^1\). Doing it for multiple component calls inside a program while considering interferences between call executions due to resource sharing and data flow is referred to as the global component composition problem.

In this work, we study the HEFT (Heterogeneous Earliest Finish Time) greedy heuristic scheduler which considers one component call at a time and is used by many GPU-based runtime systems for performance-aware implementation selection. We discuss its effectiveness for component composition in programs containing more than one component call, on a GPU based system. Composition scenarios with both independent and data-dependent component calls where this heuristic might produce an overall sub-optimal schedule are shown. Furthermore, we describe four coordination constructs that can be used to model relationships between different component calls in a hierarchical manner and can be used for making better composition decisions. We discuss a component composition scenario with two or more component calls constrained inside a data dependency chain and propose a bulk scheduling heuristic that can make better decisions by considering data dependency between different component calls inside the chain. With our global composition framework supporting the program control and data flow analysis when injecting the composition code, we can implement such global heuristics in an automated way. Effectiveness of our bulk scheduling heuristic is evaluated using two examples on a GPU-based system.

1 Introduction

The switch to parallel and heterogeneous computing is considered a sea-change in computer architecture. Modern heterogeneous systems such as GPU-based systems expose parallelism of unprecedented scale, different type of compute devices as well as disjoint memory address spaces to the programmer. On the software side, getting performance on these systems is a non-trivial task that often results in programming and portability

\(^1\) It is also called optimized composition if considered together with resource allocation and scheduling of call executions.
trade-offs. Programming different devices present in these systems often requires usage of different programming models for CPU (Pthreads, OpenMP etc.) and GPU (CUDA, OpenCL etc.) devices respectively. Furthermore, low-level and architecture specific optimizations employed to get performance often limits the (performance) portability of a solution to other architectures.

A component-based approach can help in addressing some of the programming issues associated with these architectures. Software components can be used to build complex software in a modular fashion, allowing code-reuse across different software projects. A software component consists of an interface that describes the functionality and one or more implementations of that functionality. The implementations of a component provide functional equivalence and thus can be used interchangeably. The user (caller) of a component needs to know just the interface of a component to use that component. Therefore, the selection of the implementation for a given component call can be made transparently behind the scene. High performance can be achieved by having well-optimized architecture-specific implementations of a component available for a given platform. Portability of a program using components can be ensured by having implementations of a component available for different architectures and programming models. Also, the component-based approach can reduce the programming efforts considering that component implementations can be written by expert programmers and made available to large audiences (e.g., in the form of libraries). However, the right selection of an implementation for a given component call, with respect to a specified optimization objective\(^2\), is a non-trivial problem as we will discuss it.

The basic idea of having multiple implementations is used mostly for GPU-based systems [2, 4, 6–12]. On heterogeneous GPU-based systems, the component implementations are written either for CPU or GPU execution which integrates resource allocation with the implementation selection problem. Making this implementation selection for an arbitrary number of component calls (i.e., a directed acyclic graph (DAG) of tasks) at runtime is normally referred to as task scheduling problem. The task scheduling problem is extensively studied in the literature and various heuristics have been proposed, both for homogenous and heterogeneous systems. Several static and dynamic task scheduling heuristics have been proposed in the last few decades. In static scheduling (e.g. list based algorithms), the decision is made at compile time assuming that all information needed for scheduling, such as the structure of the parallel application, the execution times of individual tasks and the communication costs between tasks is known at that time. On the other hand, in dynamic scheduling, the decision is made at runtime, which puts strict requirements on low scheduling overhead.

In this work, our focus is on dynamic implementation selection techniques used in practice for GPU-based systems. For that purpose, we would consider the Heterogeneous Earliest Finish Time scheduling (HEFT) technique which is a static scheduling heuristic, proposed by Topcuoglu et al. [17] for heterogeneous systems. It has been implemented as a dynamic scheduling/selection mechanism for GPU-based systems by many runtime systems (e.g. StarPU [12], UnMP [10], Nanos++ [11]). In this paper, we discuss this heuristic and practical implications of this greedy heuristic. We also discuss application scenarios where this heuristic might give poor performance and

\(^2\) We consider shortest execution time as the optimization objective for discussion in this paper.
propose some practical global heuristic(s) to achieve better performance on GPU-based systems.

This paper is organized as follows: Section 2 discusses the implementation of HEFT local greedy heuristic for GPU-based systems. Section 3 briefly describes our global composition framework whereas Section 4 describes global heuristics that we have devised. Related work is presented in Section 5 and Section 6 concludes and presents future work possibilities.

2 Greedy local implementation selection

Before describing the heuristic implementation, we describe how the runtime systems actually work for GPU-based systems. Figure 1 describes an application execution using a runtime system. The runtime system serves as a middleware between application and the hardware. The application exposes tasks to the runtime system which can detect data dependencies between submitted tasks based on their operand data. Only tasks whose dependencies are resolved (i.e. ready tasks) are considered for scheduling. Each runtime system provides different scheduling policies (HEFT, work-stealing etc.) and a chosen policy is used to schedule the tasks on the underlying hardware. The scheduling engine considers only the ready tasks and often schedules them considering one ready task at a time (greedy scheduling).

The greedy HEFT scheduler considers one component call at a time when making implementation selection. The selection is normally done by estimating the execution time with the help of a performance model for each component implementation. The performance model could be trained empirically either offline [9] or generated online [12] by the runtime system; it could also be an analytical model supplied by the implementation writer [13]. There exist two variations of this greedy local selection/scheduling technique:

- **Synchronous component calls**: As shown in Figure 2(a), with synchronous component executions, the selection decision is made by considering which component implementation has the expected shortest execution time for the current component implementation.
call. All compute devices are assumed to be available for execution when making the decision. It is quite simple to implement however it cannot exploit parallel execution across different compute units in the system.

Asynchronous component calls: With a runtime approach allowing asynchronous component execution, the expected duration for each task (component call) scheduled on different workers is tracked throughout the program execution. As shown in Figure 2(b), for a new component call, the component implementation on a worker which results in shortest overall expected timespan is selected. This is a greedy heuristic that tries to reduce overall timespan of the program execution and is employed by the runtime systems [10–12].

As GPU-based systems have separate memory address spaces, the operand data locality must be considered when making the selection decision and potential cost of data transfers needs to be included in case operand data is not already placed in the right memory unit. This requires runtime handling of operand data to track, transfer and optimize its usage across different memory address spaces. The data transfer cost is estimated by running bandwidth tests between different memory units on a given architecture. Based on these measurements, a simple model is built using latency ($\text{latency}$) and average cost of transferring one byte ($\text{costPerByte}$) which can later be used to predict data transfer cost between two memory units $a,b$ for $B$ bytes in direction $a \rightarrow b$:

$$\text{time} = \text{latency}_{[a][b]} + \text{costPerByte}_{[a][b]} \times B.$$  

Some approaches (e.g. [8, 9]) avoid runtime data handling for operand data by considering data transfers as part of the GPU component implementations. At component invocation, the operand data is placed inside the main memory no matter which component implementation is invoked and the implementation is internally responsible for transferring data back and forth if necessary. Although simple, it can have serious performance implications considering that no data locality can be exploited across different component calls for GPU execution.
2.1 Performance remarks

Although a greedy heuristic, the HEFT technique works quite well in practice. The main reason for its success is that this heuristic is practical and efficient. It is generic on the performance model and is normally realized using empirically tuned performance models. These models provide accurate performance estimation for computations with deterministic execution times. Results are reported in the literature for both small and medium size applications from different domains [7–12]. One reason for its success is that many computational applications used for evaluation with GPU-based systems (e.g. many applications in RODINIA benchmarks) internally have one computational kernel (component) normally executed multiple times. For such applications, this heuristic works quite well as reported by many. The main limitation of this heuristic can come for applications with multiple (types of) component calls. Even for many of these applications, this heuristic can still work reasonably well as we will discuss it later.

Also, the runtime overhead of this heuristic is negligible, usually up to a few microseconds, as reported in many earlier works [7, 9, 12]. Even performance models can be calibrated online considering low overhead of performance feedback which allows effective online learning [9, 10].

2.2 Scheduling multiple component calls

The HEFT heuristic considering one component call at a time can result in sub-optimal decisions in presence of multiple component calls. Here we discuss two possible scenarios for GPU-based systems where this heuristic can give a sub-optimal global decision.

Independent component calls (farm) : Two or more component calls that operate on different data can be executed concurrently. Runtime approaches supporting asynchronous component execution such as StarPU can schedule the component calls one after the other and can execute them concurrently. Figure 3a shows one such scenario.

---

Fig. 3: Two execution scenarios showing sub-optimal execution of HEFT greedy scheduler in case of two (a) independent and (b) data-dependent component calls respectively.
We suppose two independent component calls, each having an OpenMP and a CUDA implementation to choose from with the corresponding predicted execution time listed for each. For simplicity of demonstration, we assume no overhead of selection/scheduling in this case. The scheduler in runtime systems (e.g. StarPU) would first schedule the call1 call on the OpenMP worker by making a greedy decision. The second call (call2) would consequently be scheduled on the CUDA worker as the GPU is idle. In this case, this would result in a sub-optimal global decision.

Data dependent computations (sequence) : This scenario can happen with both synchronous and asynchronous component execution in case runtime data handling of operand data is in place. This is quite relevant to GPU-based system that expose different memory address spaces with PCIe interconnect. Exploiting data locality and minimizing PCIe communication is important in these architectures to get good performance [15]. Figure 3b shows a simple scenario with two component calls constrained in a data dependency chain (i.e. no parallelism). We separately present the communication cost in this case to make the effect of data transfers clear. The corresponding predicted execution time is listed for different implementations and data transfer. Assuming operand data is initially placed in the main memory and more component calls using this operand data can follow, we just consider data transfer cost from main memory to GPU worker device memory at the start of execution. The greedy scheduler would schedule both calls on the OpenMP worker one after the other whereas scheduling both calls on the CUDA worker would give better results in the end. We will later demonstrate this scenario with a real application along with a bulk heuristic to make better decisions in such situations.

3 Global composition framework

In this section, we briefly describe our global composition framework (GCF) that enables implementation selection for one or more component calls. Our framework uses ROSE source-to-source compiler [16] to analyse the set of component calls along with program control and operand data flow and can inject as well as modify existing source code. A component model that allow specifying extra information for a component supports the framework. The component model has the following three component-related concepts where extra information can be specified using C++ pragmas:

1. An interface that describes the functionality, using a C++ function declaration. Extra information include data accesses patterns (read, write or both) as well as relationship between different parameters (e.g., one parameter may describe the size of another parameter).

2. One or more implementations of each interface that implement the functionality described by the interface. For each component implementation, the following extra information can be specified: the execution platform (e.g. CPU, GPU) it targets, the programming environment it uses (e.g. C++, OpenMP, CUDA, OpenCL), a performance model for this implementation variant, and specific requirements about

---

4 The details of this framework are described elsewhere [18].
the target platform and execution context that must be satisfied for its successful execution (selectability conditions).

3. A component call that actually invokes the component functionality by calling the interface. Extra information can include constraints on the execution (e.g., execution must be carried out on a GPU) and/or on the call properties (e.g., limit on problem size). Besides specifying information at a single component call site, in Section 4, we will discuss coordination constructs that can be used to specify information about the relationship of multiple component calls.

Besides the component model, 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, 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. Moreover, extra data-handling code can be injected for operand data in the program control flow as well as existing code can be altered if necessary. 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 [12].

4. A performance modeling API that specifies a generic and extensible interaction interface between the performance models and the runtime library. Under-
neath, it supports specification of different kinds of performance models including empirical and user-supplied (e.g., analytical).

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. It currently supports both HEFT-based composition and a global composition heuristic (bulk scheduling).

As shown in Figure 4, our framework can generate the composition and data management code for performance-aware composition with our GCF runtime library as well as with the StarPU runtime system [12].

4 Global Heuristics

In Section 2, we discussed limitations of HEFT greedy heuristic that considers one component call in isolation when doing selection/scheduling. To make better decisions, we need to consider all or a set of component calls together when making the decision. In the following, we describe a set of call group constructs that can represent coordination relationships between different component calls which can help in making better scheduling decisions across different component calls. Specifically, we describe four call group constructs:

1. farm: A set of component calls that are (data) independent of each other and can be executed in parallel if resources permit. We consider one component call as the base case of farm group.

2. sequence: Two or more component calls that are constrained inside a data dependency chain (i.e. no parallelism). This set of component calls must be executed in sequence.

3. loop: A construct that models repetitive execution of one or more component calls.

4. exclusive: A construct that models mutual exclusive execution of different sets of one or more component calls. A typical example is if-else conditional execution where component calls inside the if statement do not execute if the component calls inside the else statement execute and vice-versa.

All four constructs described above can be nested inside each other in arbitrary fashion which allow representing complex component calls interaction in a hierarchical manner. For example, the loop construct can model repetitive execution of a sequence of component calls. Moreover, these constructs with well-defined semantics, can help in reducing complexity of making more informed scheduling decisions and enable modular and hierarchical composition. Figure 5 shows usage of grouping constructs to represent a component call graph using the four described constructs in a hierarchical manner.

In many cases, these coordination groups can be detected by source code static analysis for control and data flow for component calls (e.g. using the ROSE compiler in our framework). However, certain programming constructs or #pragmas can also be provided for the user to specify information about each group. Listing 1 shows group annotations (using #pragmas) for the component call graph shown in Figure 5(right). Extra information can be specified using such annotations; for example, a static estimate
about the loop iterations count to make some decisions statically. Moreover, in some
cases, the user can complement the static analysis framework with information about
more complex program structures.

The four group constructs proposed above can model different scheduling scenar-
ios. As a first step, we consider two base scheduling scenarios that can be modeled by
farm and sequence/loop constructs. In future, we plan to consider more complex (hi-
erarchical) composition scenarios with two or more such constructs nested/composed
together.

4.1 Farm scheduling

The farm group represents independence between its members (component calls and/or
other groups constructs) that can execute concurrently. In many cases, the farm mem-

Fig. 5: A component call graph representation using the grouping constructs in five steps
(left) and corresponding group tree that is constructed (right).

Listing 1: The group annotations for the call graph shown in Figure 5.Pragma
annotations for component calls are omitted here for brevity.
bers would reside adjacent to each other in the source code as we have in Listing 1 and possibly all can start execution at the same time. To improve on local greedy scheduling, the scheduler would need to consider all component calls inside the farm group at once while making the scheduling decisions. Implementing a practical global heuristic for farm component calls could become difficult if component calls are apart and there exists some source code in between for which no reasoning about its execution time can be made. In some cases, the programmer can restructure the source code if the component calls are independent from that intermediate source code. In other more complex cases where other user code lies in between the component calls with no constraint on its timing, sticking with the local heuristic and scheduling each call as it comes might prove a better solution in practice.

A practical scenario for global composition with a farm group is that all farm calls are ready for execution at the same time. In such situations, with a single component implementation available for each worker, an optimal schedule would require considering $W^N$ scheduling possibilities where $W$ is the number of workers and $N$ is number of component calls in the farm group. For the simple and most typical case (e.g. the one shown in Figure 3a) with two workers (an OpenMP and a CUDA worker) and two component calls in the farm group, $4$ ($2^2$) combinations need to be considered. However, in situations with more component calls/workers or with multiple implementations available for each worker (e.g., multiple OpenMP implementations), the overhead of considering all possible combinations could supersede the potential gain offered by the improved decision. In such situations, global heuristics with low overhead can prove more useful. Designing such heuristics is considered part of the future work.

![Normalized exec. time](chart.png)

**Fig. 6:** Execution of 2 independent tasks (DAXPY and DGEMM) on 1 CPU and 1 CUDA worker with each task having one CPU and one CUDA implementation. CPU and CUDA execution serialize task execution while HEFT makes sub-optimal decisions in this case just like shown in Figure 3a.

**Example:** Figure 6 demonstrate the scenario presented in 3a on a modern GPU-based system. Two independent tasks (DAXPY and DGEMM) with different execution complexity, each having one CPU and one CUDA implementation, are executed with a runtime system having 1 CPU and 1 CUDA worker. Execution of both tasks on either CPU or GPU worker serialize their execution\(^5\) whereas HEFT and Optimal execution executes both task in parallel by running one task on CPU worker and one on CUDA

---

\(^5\) The runtime system used for the experiment does not support concurrent kernel execution on GPUs.
The optimal scheduling for this experiment is obtained by re-ordering both calls (see Figure 3a) and using HEFT afterwards.

```c
// 1D arrays: y, help1, yscal, temp, err_vector

#pragma gcf call interface(libsolve_copy)
libsolve_copy(y, help1, ode_size);
#pragma gcf call interface(absaxpy)
absaxpy(y, help1, yscal, h, ode_size);
#pragma gcf call interface(axpy)
axpy(help1, y, temp, h, ode_size);
#pragma gcf call interface(libsolve_copy)
libsolve_copy(temp, y, ode_size);
#pragma gcf call interface(scale)
scale(help1, err_vector, h, ode_size);
#pragma gcf call interface(absquotMax)
absquotMax(err_vector, yscal, &error_max, ode_size);
```

Listing 2: Libsolve source code with component calls.

Fig. 7: Comparison of tool-generated performance-aware (TGPA) composition using bulk scheduling heuristic with HEFT and direct (OpenMP, CUDA) execution for ODE solver component calls [14] on a platform containing NVIDIA C2050 GPU and 2 Intel Xeon E5520 CPUs.

4.2 Bulk scheduling

The `sequence` groups represents a set of component calls (or other groups constructs) constrained in a data dependency chain that shall be executed one after the other. As we have seen earlier, if the data operands reside in main memory, scheduling considering one component call at a time might prefer CPU implementations over GPU implementations considering extra communication cost for GPU data transfers. This could prove sub-optimal in case we have more (upcoming) component calls operating on the same operand data. By amortizing the communication cost over all component calls, going to GPU for the first component call (although locally sub-optimal) may prove an overall
better choice in the end. Listing 2 shows such a scenario with a code portion of a Runge-Kutta ODE Solver from the LibSolve library [14] containing multiple component calls with data dependency between them.

The scheduler needs to consider such a group of component calls together when dispatching the first call. When dispatching them together, one choice could be to consider all possible call combinations; however, this could result in big overhead especially considering that the decision is done at runtime. For example, in Listing 2, for seven component calls with each call having just two implementations (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 (or a loop with one or more component calls in 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 the 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.

Fig. 8: Comparison of tool-generated performance-aware (TGPA) composition using bulk scheduling heuristic with HEFT and direct (OpenMP, CUDA) execution for the syntactic application on a platform containing NVIDIA C2050 GPU and 2 Intel Xeon E5520 CPUs.

A similar case is for a loop group with a sequence of one or more component calls that are repeated multiple times. Amortizing communication overhead over multiple
loop iterations may change the selection of the best variant. Making a greedy choice considering just a single component call might result in overall bad performance. Listing 3 shows pseudo-code of a BLAS Level 1 DAXPY component call (with an OpenMP and a CUDA implementation) executing inside a loop over the same data. Considering one component call at a time with operand data initially placed in the main memory, the HEFT scheduler might prefer an OpenMP implementation over a CUDA implementation if the data transfer overhead to GPU memory supersedes potential computational advantage of GPU execution. The bulk scheduler can make better decision by considering data transfer cost amortized over multiple executions of the component call. As the scheduling decision is made at runtime, the loop iteration count needs not be necessarily known statically.

Figure 8 shows the execution of the code shown in Listing 3 with bulk scheduling, HEFT as well as direct execution using OpenMP and CUDA implementation. The bulk scheduler makes more informed decision by amortizing communication overhead for GPU execution over multiple loop iterations\(^6\). The overhead of making bulk scheduling decision at runtime is included in the measurements and it proves to be negligible even for smaller problem sizes.

5 Related Work

In this work, we study the HEFT scheduler which is implemented by many runtime systems (e.g., StarPU \cite{12}, UnMP \cite{10}, Nanos++ \cite{11}). To the best of our knowledge, this is the first work on evaluation of the HEFT scheduler in context of global component composition for GPU based systems.

Regarding implementation selection, there exists many approaches that use the idea of multiple implementations for a given functionality and provide some selection mechanism for modern multicore and heterogeneous systems. Some of them propose a unified programming environment/model abstraction such as PetaBricks \cite{2}, EXOCHI \cite{3}, Merge \cite{4}, IBM's Liquid Metal \cite{5} and Elastic Computing \cite{6}. Other works \cite{7–12} do not propose a new programming model/API but rather support established programming models/APIs (OpenMP, PThreads, CUDA, OpenCL etc.). Our global composition framework is similar to the latter works with respect to supporting established programming models/APIs; however, it differs in its ability to do composition considering more than one component call at a time as well as its ability to analyse program control and data flow for making effective composition decisions.

6 Conclusion and Future work

Although HEFT is a greedy heuristic considering one component call when making the decision, it is still considered the most widely used and effective dynamic implementation selection mechanism for GPU-based systems. However, as a heuristic it can yield sub-optimal results. We have discussed two such scenarios and proposed four composable group constructs to model relationship information about different component

\(^6\) The loop iteration count was set to 50 for this experiment.
calls. As a first step, we develop a global bulk heuristic for a group of component calls constrained in a data dependency chain (sequence group) and shows its effectiveness with two examples over the HEFT scheduler. With our global composition framework (GCF) supporting source-to-source code analysis and transformation, we can generate the composition code for both HEFT and the global bulk heuristic for a given application program.

In future, the possible composition decisions in context of GPU-based systems with the four group constructs need to be explored further with more complex composition scenarios and applications showing more complex organization of component calls. Furthermore, global heuristics for a group of independent component calls (farm group) need to designed and evaluated with other approaches.

References


