Mozart: Efficient Composition of Library Functions for Heterogeneous Execution

Rajkishore Barik(✉), Tatiana Shpeisman, Hongbo Rong, Chunling Hu, Victor W. Lee, Todd A. Anderson, Greg Henry, Hai Liu, Youfeng Wu, Paul Petersen, and Geoff Lowney

Intel Corporation, Santa Clara, CA, USA
rajbarik@uber.com

Abstract. Current processor trend is to couple a commodity processor with a GPU, a co-processor, or an accelerator. To unleash the full computational power of such heterogeneous systems is a daunting task: programmers often resort to heterogeneous scheduling runtime frameworks that use device specific library routines. However, highly-tuned libraries do not compose very well across heterogeneous architectures. That is, important performance-oriented optimizations such as data locality and reuse "across" library calls is not fully exploited. In this paper, we present a framework, called *Mozart*, to extend existing library frameworks to efficiently compose a sequence of library calls for heterogeneous execution. *Mozart* consists of two components: *library description* (LD) and *library composition runtime*. We advocate library writers to wrap existing libraries using LD in order to provide their performance parameters on heterogeneous cores, no programmer intervention is necessary. Our runtime performs composition of libraries via task-fission, load balances among heterogeneous cores using information from LD, and automatically adapts to runtime behavior of an application. We evaluate *Mozart* on a Xeon + 2 Xeon Phi system using the High Performance Linpack benchmark which is the most popular benchmark to rank supercomputers in TOP500 and show GFLOPS improvement of 31.7% over MKL with Automatic Offload and 6.7% over hand-optimized ninja code.

1 Introduction

The current processor trend is to couple a commodity processor with GPUs, co-processors, or accelerators. Such heterogeneous systems not only offer increased computational power but also deliver high energy efficiency. However, due to the architectural differences between cores of the host processor and device, it is increasingly difficult to extract every-bit of performance out of these platforms, leading to growing "ninja gap" [38] where only a small number of expert programmers are capable of harvesting the full potential of the system.

Although compilers have matured significantly over the years, most of the time, compiler generated code still can not compete with hand-optimized implementation even on a homogeneous architecture. An alternative approach

L. Rauchwerger (Ed.): LCPC 2017, LNCS 11403, pp. 182–202, 2019.
https://doi.org/10.1007/978-3-030-35225-7_13

commonly used in several application domains is to use highly-tuned high-performance libraries to close ninja-gap without placing unnecessary development burden on a programmer. For example, Intel's Math Kernel Library (MKL) and NVIDIA's cuBLAS are two widely used high performance linear algebra libraries for CPUs and GPUs, respectively. These libraries are developed by domain experts who fully exploit the underlying processor architecture.

As heterogeneous systems become ubiquitous, it is challenging to make the existing library frameworks heterogeneity-aware: the libraries have to be specialized for each particular processor or device, the workload has to be properly load-balanced between them, and communication between them must be overlapped with computation as much as possible to gain performance. In order to determine optimal work distribution between devices one must use device specific performance characteristics of libraries (e.g., throughput of a library on each device). More importantly, work distribution and communication generation must look beyond just a single library call to improve data locality and reduce communication overhead. Sometimes the communication latency hiding techniques can be complicated as optimal device offload granularity might depend on the input problem size. Thus, a generic easy-to-use framework is necessary that not only allows automatic learning of device specific performance characteristics of libraries but also efficiently composes multiple library calls without any intervention from the programmer.

Existing research in this area fall into two broad categories:

- *Leverage device specific libraries and use a heterogeneous scheduling runtime for work distribution and communication.* In this approach, the device-specific libraries are treated as black-box and thus, the runtime can not easily take advantage of the expert programmers' domain knowledge related to the properties of a library during scheduling. Moreover, the programmer is responsible for providing wrappers for device-specific library task implementations on her own. There has been a lot of research work in the context of dividing work between CPU-GPU using a scheduling runtime [5,14,23,26,31–33,35,40]. Although, their techniques can be adopted in the context of libraries, they are primarily restricted to the work distribution of a single library call. Important inter-library call optimizations are left unexplored.
- *Library frameworks perform work distribution between heterogeneous cores transparent to the programmer.* Although this approach achieves peak performance for a single library call by having expert programmers' knowledge embedded in it, it can not perform optimizations across library calls. Moreover, the programmer manually instructs the library to execute on a single device or on the heterogeneous system. This approach is recently adopted by MKL by adding a functionality called Automatic Offloading (AO), which can offload part of the library call workload from a Xeon CPU to a Xeon Phi co-processor [3]. However, this advanced feature is currently limited to "sufficiently large problems" exhibiting large computation to data access ratio. Thus, only a hand-full of Level-3 BLAS functions (GEMM, SYMM, TRMM, and TRSM) and three matrix factorization routines (LU, QR, and Cholesky)

have this feature today. Although this approach is best for programmers, our experiments show that it can leave 25% performance on the table when compared to a `Hand-tuned` version for the High Performance Linpack benchmark on average (details in Sect. 4).

To the best of our knowledge, none of the existing systems exploit inter-library call optimizations such as data locality and reuse. That is, if a series of library tasks are invoked one after another, it is possible to schedule them better by grouping them based on their data access patterns rather than naively executing them one after another. It might also be necessary to decompose a library call to finer granularity in order to improve its scheduling and reduce communication overhead. The goal of this paper is to devise a generic framework that can perform optimizations across library call boundaries in order to efficiently compose them in heterogeneous systems and further reduce the ninja-gap.

In this paper, we propose a framework for library composition, called *Mozart*. *Mozart* consists of two components: *library description* (LD) and *library composition runtime*. *Mozart* transparently composes and decomposes such library calls across heterogeneous processors delivering performance on par with that of expertly tuned hand-written code. LD expresses library routines as tasks and embeds library developer expertise via meta-information about every library routine. The meta-information initially comes from library developers, is subsequently augmented with install-time profiling on the target platform, and is finally used in guiding the scheduling runtime to automatically load balance between heterogeneous processors. The runtime dynamically builds a runtime task graph for an application from the numerous library calls of the application, dynamically decomposes tasks in the graph according to the granularity specified by LD and assigns them for execution to host processor and device as they become available. To facilitate efficient composition of library tasks, our runtime applies a novel optimization, *task-fission*, that pre-processes the task graph as it is being constructed and partitions the tasks into coarse-grain sub-tasks according to data flow between the tasks. We demonstrate that our approach can improve data-locality and reuse, resulting in improved performance compared to existing approaches. To the best of our knowledge, our work is the first attempt to seamlessly perform library call composition for heterogeneous architectures.

Compared to other task based approaches such as [5,9,14,23,26,31,32,35,40], *Mozart* has the following additional capabilities. First, *Mozart* transparently schedules a decomposable library task between host processor and device cores using library metadata (LD) provided by library writers. Unlike existing systems, programmer is not responsible for writing any wrappers for device specific library implementations. Second, *Mozart* efficiently composes a series of library tasks by performing *task-fission* dynamically on the runtime task graph resulting in improved data locality. Finally, *Mozart* profiles device data transfer overhead and measures host/device throughput at runtime. It then uses this data to adaptively control the number of iterations performed by the device including the offload

granularity of device. This results in both improved load-balance and better communication-computation overlap.

The key contributions of this paper include:

- a novel *Library Description* (LD) framework to describe meta-information about libraries. We expect library developers to use this framework to specify their domain knowledge about library routines. The parameters of this framework are either specified by the library developer or determined via install-time profiling of libraries on the target platform.
- a scheduling runtime that performs load balancing among heterogeneous cores using the LD information. The distinguishing features of our runtime include dynamic *task-fission* and adaptation to runtime behavior of an application. Compared to existing scheduling runtimes [6,14,26,31,32,34,40], our runtime enables cross library call scheduling optimizations such as data-locality and communication optimization as well as host and device work distribution.
- an experimental evaluation of *Mozart* in a heterogeneous system consisting of a Xeon CPU and 2 Xeon Phi co-processors using High Performance Linpack benchmark which is the most popular benchmark to rank supercomputers in TOP500. Our results show a GFLOPS improvement of 31.7% over MKL with Automatic Offloading and 6.7% over hand-optimized version of the application. *Please note that, although we perform our experimental evaluation on a Xeon+Xeon Phi system, our technique should be applicable to any host plus device based system including widely used CPU+GPU based systems.*

The rest of the paper is organized as follows. Section 2 describes the library description interfaces. Section 3 describes our heterogeneous scheduling runtime. Evaluations are presented in Sect. 4. We discuss related work in Sect. 5 and conclude in Sect. 6.

2 Library Description Language

In this section, we describe the library description (LD) framework that drives composition of library calls at runtime, so that the calls are effectively executed in a distributed fashion on a heterogeneous system – to take advantage of the rich hardware parallelism in such a system. LD expresses domain knowledge from library developers and uses install-time profiling to build platform-specific performance models. Such expertise, code, and models reflect important aspects of the dynamic behavior of a library function.

Library experts (typically library-writers) build a library description (LD) for each library function during or after library development. This LD is basic in that it might not contain platform-specific information. The expert also builds an extensive set of microbenchmarks to perform install-time profiling in order to fill in the LD parameters of a library. Any relevant performance characteristics of a library function can be put into the LD, but specifically, we propose a set of abstract interfaces as shown in Table 1: `Threads()` and `SubTaskSize()` are described, because the number of threads and the task granularity for offloading

Table 1. Library description APIs

API	Description
In, Out	Returns the inputs and outputs of the library function
`Threads(id)`	Returns the optimal number of threads to set for this library function for a given device *id*
`SubTaskSize(id)`	Returns the best sub-task size to set for this library function for a given device *id*
`Rate(id)`	Returns the computation throughput of the library function on a given device *id* Rate is defined as the ratio of number of iterations processed per unit time
`Affinity()`	Returns the device affinity of the library function
`Task_arch(range)`	Returns a task function operating on a sub-range of the original iteration space of the library function for a device, e.g., arch is either CPU or GPU
`InsertTask()`	Inserts task functions into the runtime's task graph and schedule it for execution when it is ready

are the two most important optimization parameters for most applications. Note that `Threads()` for memory-bound applications may be much smaller than the maximum available cores on the underlying platform. Similarly, `SubTaskSize()` determines the offload granularity to a device in order to optimally overlap it's computation with communication. When the library is installed, the microbenchmarks are used to profile the underlying platform and build platform-specific models for the functions in LD, including `Threads()`, `SubTaskSize()`, and `Rate()`. Many different combinations of the inputs of a library function can be used to run the function. With different input combinations and their corresponding execution time of the function, profiling can learn a model for each of `Threads()`, `SubTaskSize()`, and `Rate()`. The learned performance models replace the default `Threads()`, `SubTaskSize()`, and `Rate()` specified by the library-writer.

Figure 1 illustrates the implementation of LD class for a matrix-matrix multiplication library call, dgemm_LD. In this example, `Threads()` and `SubTaskSize()` are hard-coded numbers based on library-writer's experiences. `Rate()` is an auto-generated performance model built from install-time profiling (in particular, we perform a linear approximation of the profiling data after executing the microbenchmarks on each device). `Task_arch()` and `InsertTask()` are written manually by expert programmers. Here we have two `Task_arch()` functions: one for the host, the other for the device. We have used Intel Offload programming model [1] to demonstrate device offloading, but it is not a limitation.

In a specific implementation, the compiler may automatically redirect a library function call in a user program to its corresponding library description,

```
// Library description (LD) for matrix-matrix multiplication using dgemm call on  host and device
class dgemm_LD {
  int Threads(int tid) { return tid == 0 ? 40:240;}
  int SubTaskSize(int tid) { return tid == 0 ?  1500 : 3000; }
  bool Affinity() { return Device0; }
  int Rate(int tid) { // Auto-generated performance model via linear approximation
    if  (1.00000*M -20000.00000 <= 0) {
      dtmp=0.00087*M+0.00087*N +0.00143*K+157.65765;
    } else {
      if  (1.00000*M -25001.00000 <= 0)
        dtmp =0.00220*M + 0.00220*N +0.15566*K;
      else
        dtmp=-0.00008*M-0.00008*N -0.00813*K+360.9299;
    }
    return MAX(dtmp, 1);
  }
  void TaskHost(range r) {
    cblas_dgemm(...); // Host MKL call to matrix-matrix multiply
  }
  void TaskDevice(range r) {
    // Ninja written code
#pragma offload target(DEV0)
    cblas_dgemm(r, ...);  // Device library call to matrix-matrix multiply
  }
  void InsertTask() {
    // set arguments and argument metadata, call to scheduling runtime
    insert_divisible_task(...);
  }
}
```

Fig. 1. LD for matrix-matrix multiplication library, dgemm. The library developer initially writes the dgemm_LD function. During install time of the library, `Threads`, `SubTaskSize`, `Affinity`, and `Rate` are populated via install-time profiling. The `TaskHost` and `TaskDevice` functions operate on sub-ranges in order to let the runtime adaptively decide the work distribution between host and device. The `InsertTask` function calls into runtime (Sect. 3).

which enqueues a divisible task into the runtime system's task graph (described in Sect. 3). For example, for a library function `f()` with LD as `f_LD()`, the library wrapper or the compiler performs the following two operations:
$LD*\quad ld\ =\ f\_LD(/*originalparameters*/);$
$ld->InsertTask();$

When the user program runs, the above two statements create an LD object, and invoke our runtime through that object's `InsertTask()` function, which inserts the `Task_arch()` function(s) into the runtime system's task graph, and invokes the runtime system. The runtime is described in Sect. 3.

In another implementation, the library writer may hide LD details from the user by wrapping the library functions in new interfaces and expose these interfaces to the programmer. Either way, the newly created runtime task is semantically equivalent to the original library function. The runtime system executes this task in a parallel and perhaps in a distributed fashion on the heterogeneous system.

Figure 2 depicts the high-level flow of LD. At library installation time, a performance model (via linear approximation) is built for how each library function performs on the entire system with such factors as number of threads, sub-task size and the relative performance of each processor in the system. The compiler pattern matches library function call names and replaces a regular function call with a call to the runtime providing access to the model for the given function. The runtime is then responsible for determining how best to execute the function with the current inputs within the system. It will often do so by splitting up the

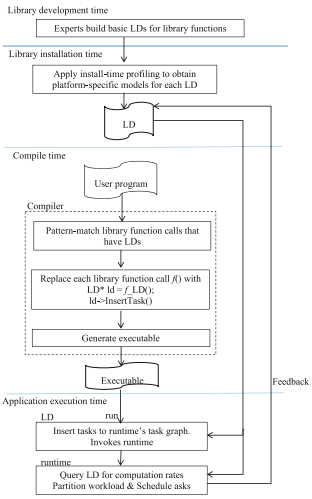

Fig. 2. Detailed work-flow of library description framework.

work into smaller granularity and giving that work to different processors in the heterogeneous system.

One thing to keep in mind is that LD interfaces are developed by library writers, programmer is not involved. Additionally, the LD information for each library routine is written exactly once and thus we believe the complexity is manageable. Augmenting auto-tuning with install-time profiling is a subject for future work.

3 Library Composition Runtime

In this section, we describe our heterogeneous runtime system that performs dynamic task-fission, overlaps communication and computation to hide communication latency of heterogeneous processor, and finally, adapts to runtime behavior of an application. Our runtime system takes advantage of the performance profile information from LD described in Sect. 2. For simplicity of presentation, we treat each task as a library task created from a particular library routine.

Library Task Representation: We introduce the notions of *simple* and *divisible* library tasks in our runtime. A *divisible* task can be further decomposed into sub-tasks. Typical example of a divisible task is a data-parallel library routine which can be decomposed in many different ways including simply varying the number of loop iterations that are grouped into a single task. For example, the classical matrix-matrix multiplication library task is a divisible task since the 2-d iteration range of the output matrix can be blocked into sub-ranges and each sub-range can be processed independently either on host or device.

Runtime Interfaces: Our runtime exposes two key APIs: *insert_{T}_task* where T is either divisible or simple and *task_wait*. *insert_{T}_task()* communicates all information regarding a task to the runtime. *task_wait()* function waits for the previously issued tasks to complete and their output data become available. Some of the important information passed to an *insert_{T}_task*$(T, \ldots)$ include iteration range and dimensionality (for divisible tasks), function pointers to device task implementations that operate on sub-tasks, arguments and their corresponding metadata information (using array access descriptors as described later in this section), device affinity if known, and performance profile information for the corresponding library function from LD (described in Sect. 2). Each argument to a task is marked as one of the following: `IN`, `OUT`, `INOUT`, `VALUE` (passed-by-value), `ACCUMULATOR` (passed-by-reference, typically reduction variables are passed as accumulator variables). Please note that the library developer wraps *insert_T_task* in the LD method `InsertTask` for a library task and *task_wait* is inferred by our runtime based on input and output dependences.

Array Access Descriptors: Affine array accesses are typical in scientific and HPC applications, therefore are used heavily in library routines. The metadata information for an array argument of a task in our runtime also carries a *descriptor* in order for the runtime to be able to determine the sub-region of the array being accessed by a sub-task, which is crucial to generate the data movements between host and devices. The following array access descriptor captures affine array accesses of the form $a * i + b$, where a and b are compile-time constants and i denotes the loop induction variable:

```
struct array_access_desc_Nd_s {
  unsigned int dim; /* number of array dimensions */
  int64_t *a1, *a2; /* "a" coefficients of ai+b in each dimension */
  int64_t *l_b, *u_b; /* "b" (lower and upper bound) coefficients of ai+b in each dimension */
};
```

The array region accessed by a sub-task can be 1-d linear, 2-d rectangular, or 3-d rectangular prism (our implementation currently does not support beyond 3-d). At runtime, each array region is associated with a location information indicating the device that holds that array region. Runtime uses this information to decide device affinity of a sub-task. Additionally, efficient implementations of standard set operations such as union, intersection, and difference are also provided for array regions in order to reduce the runtime overheads of task graph construction. Note that since we are composing libraries, the array access descriptors are written by expert library developers as part of LD, programmer who uses these libraries is not involved.

Work-Sharing Runtime: The runtime determines task dependencies from the array access descriptor metadata of task arguments, builds a *task dependence graph*, generates necessary data transfers and schedules tasks for execution possibly choosing the device. Once all the predecessors of a task in the task graph have completed execution and all the input data has been transferred to the target device, the task is executed. This might result in the output data transfer to a different device, as well as a trigger for the other task execution (Fig. 3).

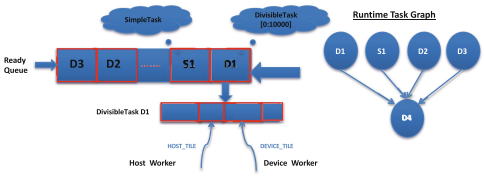

Fig. 3. Work-sharing runtime and runtime task graph

Typically a heterogeneous system node has a few devices leading to fewer contention among devices. Moreover, the optimal task granularity for offload may differ from device to device, e.g., a discrete GPU may choose a sub-task tile that completely hides the data communication latency of an application. With these design choices in mind, we implement a *work-sharing runtime* where an idle device grabs it's own sub-task tile of iterations for a ready divisible task from a common shared queue of the parallel iterations. Note that sub-task tile size for each device is bootstrapped using `SubTaskSize()` in LD and is adaptively adjusted in the runtime (described later in this section). A proxy host thread (i.e., *worker*) is assigned to each device that offloads work to that device.

When a task becomes ready, i.e., all its predecessors in the task dependence graph finished their execution, it is added to the ready queue. When a worker becomes idle, it tries to retrieve a task from this ready queue and executes it. There could be more than one task available in the ready queue, in which case one of the two following strategies could be used: (1) the idle worker first tries to pick a task that no other worker is working on (that is, *breadth-first* approach); (2) the idle worker picks the first ready task (that is, *depth-first* approach). This is currently implemented via a runtime flag. By default, a simple task has an affinity to the host CPU in our runtime.

Task Fission: Choosing sub-task size is critical to application performance on heterogeneous systems as it affects both scheduling granularity, data locality, and communication latency. It can also artificially limit parallelism and flexibility of scheduling. In general, the sub-task granularity should be large enough to occupy at least a single core of the heterogeneous system, yet small enough to support efficient load balancing and communication/computation overlap. Consider an example program shown in Fig. 4. `task1` invoked with parallel iterations [1 . . . 10000] writes 10000 elements of array b, while `task2` invoked with parallel iterations [1...500] reads only the first 500 elements of b. Consider the following scheduling constraint on a discrete CPU+GPU heterogeneous system: `task2` executes significantly better on CPU than GPU while `task1` can be executed on either CPU or GPU. An optimal scheduling solution is to execute first 500 iterations of `task1`, as well as, all of `task2` on CPU and the last 9500 iterations of `task1` on the discrete GPU until CPU completes execution, at which point both CPU and GPU can execute the remaining iterations of `task1`. This enables overlapping of `task2` and `task1` execution resulting in improved data-locality (that is, `task2` might reuse data from the cache prefetches of `task1`) and reduced

communication too (that is, there is no data transfer cost to/from GPU for the first 500 iterations of `task1`. However, if tasks have to be scheduled as indivisible units, a scheduler has to assign all iterations of `task1` to either CPU (leaving GPU unused) or GPU (resulting in data transfer cost) and wait for all of `task1` to complete before starting `task2`. Choosing fine-grain tasks resolves this problem but leads to higher overhead of maintaining task graph and scheduling tasks, potentially defeating benefits from exploiting a higher degree of parallelism.

```
task1(range r, in a, out b) {
  for (i = r.start  : r.end)
    b[i] = f(a[i]) // f is a library call
}

main() {
  insert_task([1...10000], task1); // CPU or GPU
  insert_task([1...500], task2); // CPU
  task_wait(); // wait for task completion
}
```

```
task2(range r, in b, out c) {
  for (i = r.start : r.end)
    c[i] = g(b[i]); // g is a library call
}

main() {
  insert_task([1...500], task1); // CPU
  insert_task([501...10000], task1); // CPU or GPU
  insert_task([1...500], task2); // CPU
  task_wait(); // wait for task completion
}
```

Fig. 4. Optimal task granularity example with dynamic task-fission; `task1` with range $[1 \ldots 10000]$ was split into two tasks with ranges $[1 \ldots 500]$ and $[501 \ldots 10000]$ at runtime based on consumer task `task2`; Now `task2` can start executing immediately after `task1` with range $[1 \ldots 500]$ completes on CPU. This improves locality and reduces communication.

We propose *task-fission* at runtime that automatically adjusts task granularity to discover additional available parallelism while keeping the cost of task graph maintenance and scheduling under control. Tasks are split to achieve exact match between one task output and another task input. Such tasks are further combined into task chains that can be scheduled to a single device to reduce communication cost.

We support two kinds of task-fission. The first one splits an existing task A when a new task B is inserted into task graph whose input is a subset of the output of the task A. In this case, A is split into A1 and A2, such that output of A1 is the same as input of B. This enables execution of B as soon as A1 finishes without waiting for completion of A2. It also allows the runtime to schedule A1 and B to the same device while, in parallel, executing A2 on a different device. The second one splits a new task B when its input is a super-set of the output of an existing task A. In this case, B is split into B1 and B2, such that output of A is the same as input of B1. This enables execution of B2 without waiting of completion of A and allows the runtime to schedule A and B1 to the same device, while scheduling B2 to a different one.

When a task is inserted to our runtime, the argument metadata information (described using array access descriptor) is used to derive task dependencies and to construct the runtime task dependency graph. During this addition of the newly arrived task to the runtime task graph, our runtime checks if it is feasible to perform task-fission with the immediate predecessor task or with the immediate successor task (as described in the previous paragraph). If the immediate predecessor task is not already executing and offers opportunities for

task-fission, we perform task fission and update the runtime task graph. We enqueue the newly created tasks from task-fission to our runtime when they are ready. In most cases, we expect that the cost of task-fission be mitigated by the benefits we get from task-fission. Our runtime augments online profiling to decide whether to perform task-fission or not.

Task-fission results in the following benefits: (1) increased task parallelism, due to precise matching between task dependencies and task granularity; (2) reduced communication cost, as sub-tasks with the same input/outputs can be scheduled to the same device without affecting scheduling of the whole task; (3) improved data locality and reuse, as sub-tasks with the same input/output can reuse data from caches.

Overlap Communication and Computation: Accelerators including GPUs and Xeon Phi are typically connected to the host processor via PCIe interconnect. Thus, communication overhead is one of the dominant factors in obtaining performance of these systems. There are many existing approaches for hiding communication latency [2]. We use the *double buffering* technique transparently in our runtime to overlap communication to device with computation on host CPU. This transparency is feasible in *Mozart*, since tasks can be divided into sub-tasks and the argument array regions accesses by sub-regions can be computed from the array access descriptors. Our runtime creates two temporary buffers for every argument array region corresponding to sub-tasks and while the current buffer holds the array region for computation on the device of sub-task A, the next buffer holds the array region that is used for transferring data from the device to the host (for output transfer) for sub-task B. Sub-tasks A and B originate from the same divisible task. Similarly, we can also overlap input data transfer of one sub-task with computation of another sub-task. The granularity of the sub-tasks are chosen such that the two temporary buffers fit on the device memory and more importantly, the ratio of communication to computation time of the sub-tasks must be close to 1. This results in optimal performance as it hides the communication latency completely. Our runtime initially uses LD information to choose this sub-task granularity (via `SubTaskSize()`), but later on adjusts adaptively at runtime.

Strided Data Access: Several scientific and HPC applications access strided data. Strided data can be tricky to transfer to devices using pragma based compilers such as Intel Offload compiler [1] as they are limited by their expressibility resulting in unnecessary data transfer. Consider the shaded regions to the left of Fig. 5. In order to transfer only the shaded regions to Xeon Phi using current Intel Offload compiler, entire rows and columns corresponding to the shared regions need to be transferred. This incurs runtime overhead and can be significant if the matrix is large and the shaded region is very small. We mitigate this by transparently copying data in runtime to contiguous memory locations and then remapping the index space of this data in the library kernels for both Xeon and Xeon Phi. The code snippet to the right of Fig. 5 depicts the argument offset data structure used in our runtime and its use in a matrix multiplication Xeon Phi library kernel. This data structure stores dimensionality information (dim),

row/column size (depending on row-major or column-major) using max_size, and the original indices using index_offset for each dimension. Each library kernel now takes an additional argument for this offset data structure offset and replaces each access to $A[i]$ by $A[i - \text{offset} - > \text{index\_offset}[0]]$. The offset data structure is transparently populated by runtime and passed to the library kernel before executing it on the device. This approach avoids unnecessary data transfer and is likely to improve performance for strided data access applications. Based on the above design, the library developer needs to write `Task_arch(range)` functions for libraries dealing with non-contiguous data.

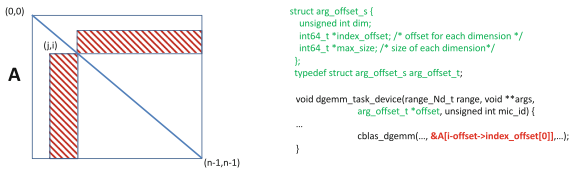

Fig. 5. Strided and non-contiguous data transfer

Runtime Adaptation: Even though LD provides platform-specific performance sketch of libraries to runtime, it is also possible to adaptively learn and improve these library parameters such as `Affinity`, `Threads`, `Rate`, and `SubTaskSize`. For instance, if an application repeatedly invokes the same library task (perhaps with a different range), we can estimate its optimal tile size for overlapping computation and communication from invocation to invocation even before executing it. We can also predict if task-fission is beneficial across library tasks (due to the overhead involved in splitting a task). Our runtime maintains an *online profiling database* in order to track performance profile of library tasks as they execute on devices. Following data structure depicts our online profiling database:

```
typedef struct task_profile_s {
  unsigned int task_type; // 0 for simple_task, 1 for divisible_task
  /* host execution profile */
  int64_t host_num_iters; /* Number of iterations executed by host */
  double host_time; /* Time taken by host */

  /*device execution profile */
  int64_t device_num_iters; /* Number of iterations executed by devices */
  int64_t num_bytes_xfered; /* Bytes of output transferred from devices */
  double xfer_time; /* Time taken for output data transfer */
  double compute_time; /*Device computation time*/

  /* task fission*/
  double task_split_time;
} task_profile_t;

/* Map from phase signature (id) to task name (string) to profiling database*/
map< int64_t, map<string, task_profile_t> > profile_db;
```

We divide a program execution into phases. Each phase consists of all the tasks being executed in between two consecutive wait_task(). We profile each phase and accumulate their information in profiling data-base. Each worker locally gathers profiling information for each task executing in a profiling phase. After the phase completes, wait_task() accumulates the per-worker profiles into

the data structure above. Typical information we collect include data transfer time, computation time on host and device, and number of parallel iterations performed by host and device. These information can be used to improve LD's `Affinity()` and `Rate()` information. Additionally, during runtime task graph construction, we use the profiling database to estimate execution time of each new invocation of a task and determine if it is beneficial to perform task-fission in the current phase or not. That is, the total estimated time with and without task-fission is computed using the throughput and data transfer overhead of each device (computed from prior iterations). Since we estimate execution time and data transfer time on device for each task, we also adjust the tile size (i.e., `SubTaskSize()`) in order to reduce the communication to computation gap.

4 Evaluation

In this section, we investigate the performance of *Mozart* in High Performance Linpack (HPL) benchmark. This program is typically written using several library calls and hence, offering opportunities to our runtime to efficiently compose them. We compare *Mozart* with `MKL Offload` and hand-tuned optimized implementations of the same computation. The details of the hand-tuned HPL implementation is described in [19]. With better load balancing, efficient library composition, and adaptation to runtime behavior, *Mozart* is able to comprehensively outperform `MKL Offload` and is also able to beat the hand-tuned ninja version.

Implementation: Figure 6 depicts our implementation framework. We have implemented *Mozart* on top of the Julia compiler infrastructure [8]. The standard Julia compiler converts a Julia program into the Julia AST, then transforms it into LLVM IR, and finally generates native assembly code. We intercept the Julia compiler at the Julia AST level, recognize the library calls via AST pattern matching and rewrote them to create the corresponding LD object and invoke the *InsertTask()* function on it (as described in Sect. 2). Finally, we generate C++ code from Julia AST. Please note that the techniques described in this paper are not tied to our choice of implementation language and can be applied to other languages as well.

Platform: We use a host server with two Intel® Xeon® E5-2690v2 processors and 128 GB RAM. The processors are code-named Ivy Bridge and manufactured using 22 nm technology. Each processor has 10 cores (20 cores total) with base frequency of 3.00 GHz. The cache sizes are 32 KB for L1I, 32 KB for L1D, 256 KB for L2, and 25 MB for the L3 cache. It runs CentOS v6.6 distribution of Linux. This host server is connected via PCIe (with bandwidth 6 GB/s) to two massively parallel Xeon Phi co-processor with 61 in-order Pentium cores with each core being 4-way hyper-threaded. Applications can use up to 240 threads simultaneously (one core is reserved for the Linux OS running on it). Each core of Xeon Phi is embedded with a 512-bit vector unit for increased SIMD parallelism. Furthermore, each core has 32 KB L1 data cache, 32 KB L1 instruction cache and

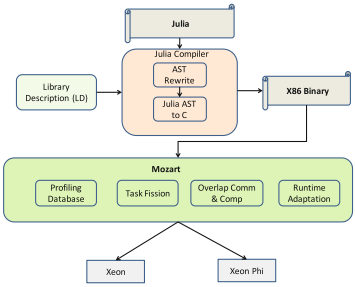

Fig. 6. Overall implementation framework of *Mozart*

512 KB partition of globally coherent L2 cache. Each Xeon phi co-processor has 8 GB of GDDR memory.

We use the Intel®C++ Compiler (ICC) v15.0.2 with "-O3" flag for compilation of the *Mozart* runtime and the C++ code generated from the Julia AST. We report execution times using the average of five runs.

High Performance Linpack (HPL) Performance. HPL is the most popular benchmark to rank supercomputers in TOP500, and it spends majority of time in numeric libraries. The key routine in HPL is LU factorization. LU factorization decomposes a matrix A into a lower-triangular matrix L and an upper triangular matrix U. We use a blocked LU factorization version that demonstrates the benefits of library composition using *Mozart*. The high-level blocked LU formulation code is shown in Fig. 7.[1] This algorithm proceeds from left to right of matrix A in blocks of nb until entire A is factorized. In each iteration of the loop, nb column panels are first factorized using dgetrf library call, nb block of matrix rows are swapped based on the pivot vector ipiv from panel factorization using two dlaswp library calls (denoted as dlaswpL and dlaswpR), and a portion of row panel is updated using dtrsm forward solver library call. The trailing submatrix of A is then updated using dgemm library call.

```
for( j = 0; j < n; j += nb ) {
  dgetrf(A[j:n-1][j:j+nb-1], ipiv[j:j+nb-1]);
  dlaswpL(A[j:j+nb-1][0:j], ipiv[j:j+nb-1]);
  dlaswpR(A[j:j+nb-1][j+nb:n-1], ipiv[j:j+nb-1]);
  dtrsm(A[j:j+nb-1][j:j+nb-1], A[j:j+nb-1][j+nb:n-1]);
  dgemm(A[j+nb:n-1][j:j+nb-1], A[j:j+nb][j+nb:n-1], A[j+nb:n-1][j+nb:n-1]);
}
```

Fig. 7. HPL blocked version demonstrating library composition.

We wrote the LD specifications for the library tasks from Fig. 7. The performance models for `Threads()`, `SubTaskSize()` and `Rate()` were built using

[1] Note that we specifically choose the blocked version of HPL to highlight the contribution of this paper, which is library composition. We do not directly use the LU factorization algorithm provided by MKL.

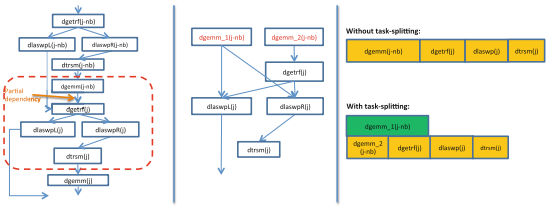

Fig. 8. HPL: task-fission performed via *Mozart*

simple curve-fitting from an extensive set of micro-benchmarking executions on our platform. The library tasks were then inserted inserted into our runtime using `InsertTask()` LD interfaces. Our runtime automatically identifies the task dependencies between them based on their input-output dependencies on matrix A and builds the task graph. The runtime task graph for two consecutive iterations $j-nb$ and j is shown to the left of Fig. 8. Our runtime identifies an opportunity to detect partial loop-carried dependency between dgemm$(j-nb)$ task and dgetrf(j) task, i.e., dependency between previous iteration $j-nb$ dgemm call and next iteration j dgetrf call. It then splits dgemm$(j-nb)$ into dgemm_1$(j-nb)$ and dgemm_2$(j-nb)$ tasks using task fission (as shown in the middle of Fig. 8). This allows dgemm_1$(j-nb)$ task to execute concurrently with both dgemm_2$(j-nb)$ and dgetrf(j) tasks (as shown to the right of Fig. 8). Both dgemm_2$(j-nb)$ and dgetrf(j) are executed on Xeon based on the affinity of dgetrf(j) task specified in LD. On the other hand, dgemm_1$(j-nb)$ starts by executing on Xeon Phi but is subsequently executed on both Xeon and Xeon Phi when dgemm_2$(j-nb)$ and dgetrf(j) tasks are completed.

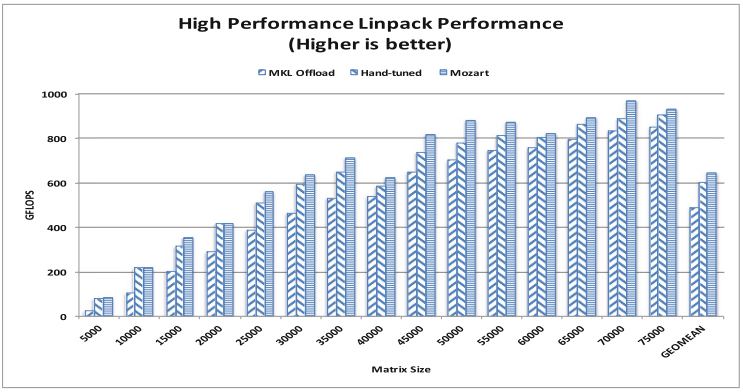

Fig. 9. Performance of High Performance Linpack running on a Xeon + 2 Phi Ivy Bridge system (higher is better). On average, `MKL Offload` approach achieves 489 GFLOPS, `Hand-tuned` achieves 604 GFLOPS, and *Mozart* achieves 644 GFLOPS for matrix sizes 5K–75K.

Figure 9 presents our experimental evaluation comparing *Mozart* with `MKL Offload` and `Hand-tuned` ninja version [19]. We vary the matrix size from 5K until 75K in steps of 5K. We observe an average GFLOPS improvement of 31.7% for *Mozart* vs. `MKL Offload`, primarily due to the fact that `MKL Offload` does not perform any cross library optimization for the blocked version of HPL, although it is able to execute the dgemm and dtrsm library functions across both Xeon and Xeon Phi and achieves peak performance for them individually. When compared to hand-tuned ninja version, *Mozart* yields an average GFLOPS improvement of 6.7%. Although hand-tuned ninja version performs library composition using a manual implementation of task fission, it does not perform the following tasks effectively: (1) the ninja version is unable to load balance effectively – it does not use a performance model like ours (via LD) to divide work between Xeon and Xeon Phi instead uses a platform-specific hand-tuned step function to determine the number of dgemm_1($j - nb$) iterations to be performed on Xeon[2]; (2) the ninja version does not perform runtime adaptation like ours as described in Sect. 3.

Performance Breakdown: The GFLOPS improvement of 31.7% for *Mozart* vs. `MKL Offload` can be explained as follows: we observe 10–15% benefit for small size matrices (5K–25K) and close to 20% benefit for larger matrices (>=30K) from our task-fission optimization, which MKL-Offload can not perform in the blocked version of HPL. Remaining benefits of close to 10–15% is obtained from runtime adaptation via sub-task granularity determination (i.e., `SubTaskSize()`) and cost-benefit analysis of task-fission (that is whether to perform task fission or not) as described in Sect. 3.

Discussion: The HPL application has the following properties that *Mozart* exploits:

- *dgemm* matrix-matrix multiplication task is computationally expensive and can be decomposed into smaller granularity which can distributed to host and device cores;
- *dgemm* can be split via task-fission into two sub-tasks where one of them can execute in parallel with *dgetrf* of the next iteration in order to reduce the critical path length.

One of the key ideas of the paper is to dynamically perform task-fission optimization across library calls. Task-fission opportunities are prevalent in many data parallel application domains including emerging AI and machine-learning that compose many data parallel libraries, e.g., Intel's DAAL and CUDA-DNN within a network model. Existing HPC applications such as LU, Cholesky and QR factorization already exhibit task-fission patterns. Thus, we believe efficient library composition via task fission will play an important role in optimizing future applications. Although it is possible to implement task-fission statically, it may not be straightforward in the presence of complicated control-flow such as the one present in HPL. In summary, the techniques described in *Mozart*

[2] Hand-tuned implementations are rarely performance portable.

including library description and library composition runtime are general and can be applied to other heterogeneous architectures and applications.

5 Related Work

Heterogeneous Execution: There have been several efforts [5,7,10–12,14,18,20,22,25,26,29,31,32,32,34–36,39,40,44] to make heterogeneous execution of applications more efficient. [4] is most closely related to our work as it analyzes a series of GPU library calls in order to minimize the CPU-GPU communication overheads by caching the reused data. Our work is different from the above body of works in two key aspects: *Mozart* performs task-fission at runtime and dynamically adapts device tile sizes. It augments library description metadata framework to decompose libraries for efficient heterogeneous execution.

Dynamic Task Graph: Dynamic task graph is widely used in parallel systems. StarPU [5], OMPSs [9,13,33], and BDDT [43] build dynamic task graphs, but they do not dynamically split tasks into subtasks. There is previous work [15] on dividing/consolidating tasks to make better use of resource or to achieve better load balance, but it considers only independent tasks. [17,19] manually splits tasks in application level to enable dynamic load-balancing on a Xeon Phi system. To the best of our knowledge, no known work splits tasks in a dynamic task graph based on the dependencies between tasks.

Overlapped Communication and Computation: [5,9,28] present runtimes that overlap communication with computation. [37] describes a hybrid threading approach where one thread handles all MPI-related operations and all other threads handle computation. [24] provides an OpenCL communication library and programming interface to optimize communication between host and accelerators. [40] uses static inter-node data and task distribution in large-scale GPU-based clusters, and dynamic task scheduling within a node to overlap communication with computation. [41] uses source-to-source compiler optimization to enable streaming for offloaded computations for Xeon Phi. In our work, we use performance models from LD and dynamically adjust tile size to completely overlap communication and computation for heterogeneous systems.

Telescopic Languages: The idea of annotating libraries in order to generate fast specialized code at the translation of scripting languages has been explored in [21]. Our library description metadata is inspired by this work, but extends it to enable cross-library heterogeneous execution via task-fission. Library annotation has also been explored in [16].

Skeleton Composition: Efficient composition of algorithmic skeletons such as map, reduce, and zip for shared-memory systems and clusters has been explored in the STAPL Skeleton Framework [42,45,46].

6 Conclusion

In this paper, we show that library composition plays a crucial role in achieving peak performance in heterogeneous architectures. We propose a framework, *Mozart*, consisting of two components: a *library description* (LD) interface for library writers and a generic *library composition runtime*. The runtime performs task-fission on-the-fly in order to improve data locality and data reuse across library calls using the performance parameters from LD. *Mozart* transparently composes library calls across heterogeneous cores and delivers close-to expertly tuned performance. Our experimental evaluation on a heterogeneous system consisting of a Xeon CPU and 2 Xeon Phi co-processors executing High Performance Linpack benchmarks shows that *Mozart* achieves an average GFLOPS improvement of 31.7% over MKL+AO and 6.7% over hand-optimized code. In future, we would like to augment auto-tuning within our framework to further improve our performance results. We would also like to extend *Mozart* to handle non-affine array accesses. A tool that can automatically generate LD specifications for library routines is also a subject for future work.

References

1. Effective Use of the Intel Compiler's Offload Features. https://software.intel.com/en-us/articles/effective-use-of-the-intel-compilers-offload-features
2. How to Overlap Data Transfers in CUDA C/C++. https://devblogs.nvidia.com/parallelforall/how-overlap-data-transfers-cuda-cc/
3. Intel Math Kernel Library Automatic Offload for Intel Xeon Phi Coprocessor. https://software.intel.com/en-us/articles/math-kernel-library-automatic-offload-for-intel-xeon-phi-coprocessor
4. AlSaber, N., Kulkarni, M.: Semcache: semantics-aware caching for efficient GPU offloading. In: Proceedings of the 27th International ACM Conference on International Conference on Supercomputing, ICS 2013, pp. 421–432. ACM, New York (2013)
5. Augonnet, C., Thibault, S., Namyst, R., Wacrenier, P.-A.: StarPU: a unified platform for task scheduling on heterogeneous multicore architectures. Concurr. Comput.: Pract. Exp. **23**(2), 187–198 (2011)
6. Barik, R., et al.: Efficient mapping of irregular C++ applications to integrated GPUs. In: IEEE/ACM International Symposium on Code Generation and Optimization (CGO) (2014)
7. Belviranli, M.E., Bhuyan, L.N., Gupta, R.: A dynamic self-scheduling scheme for heterogeneous multiprocessor architectures. ACM Trans. Archit. Code Optim. **9**(4), 57:1–57:20 (2013)
8. Bezanson, J., Karpinski, S., Shah, V.B., Edelman, A.: Julia: a fast dynamic language for technical computing. CoRR, abs/1209.5145 (2012)
9. Bueno, J., Martorell, X., Badia, R.M., Ayguadé, E., Labarta, J.: Implementing OmpSs support for regions of data in architectures with multiple address spaces. In: Proceedings of the 27th International ACM Conference on International Conference on Supercomputing, ICS 2013, pp. 359–368. ACM, New York (2013)
10. Cederman, D., Tsigas, P.: On dynamic load balancing on graphics processors. In: Proceedings of the 23rd ACM SIGGRAPH/EUROGRAPHICS Symposium on Graphics Hardware, GH 2008, Aire-la-Ville, Switzerland, pp. 57–64 (2008)

11. Chatterjee, S., Grossman, M., Sbîrlea, A., Sarkar, V.: Dynamic task parallelism with a GPU work-stealing runtime system. In: Rajopadhye, S., Mills Strout, M. (eds.) LCPC 2011. LNCS, vol. 7146, pp. 203–217. Springer, Heidelberg (2013). https://doi.org/10.1007/978-3-642-36036-7_14
12. Chen, L., Villa, O., Krishnamoorthy, S., Gao, G.R.: Dynamic load balancing on single- and multi-GPU systems. In: IEEE International Symposium on Parallel and Distributed Processing (IPDPS), pp. 1–12 (2010)
13. Chronaki, K., Rico, A., Badia, R.M., Ayguadé, E., Labarta, J., Valero, M.: Criticality-aware dynamic task scheduling for heterogeneous architectures. In: Proceedings of the 29th ACM on International Conference on Supercomputing, ICS 2015, New York, NY, USA, pp. 329–338 (2015)
14. Grewe, D., Wang, Z., O'Boyle, M.F.P.: OpenCL task partitioning in the presence of GPU contention. In: Caşcaval, C., Montesinos, P. (eds.) LCPC 2013. LNCS, vol. 8664, pp. 87–101. Springer, Cham (2014). https://doi.org/10.1007/978-3-319-09967-5_5
15. Guo, Z., Pierce, M., Fox, G., Zhou, M.: Automatic task re-organization in MapReduce. In: 2011 IEEE International Conference on Cluster Computing (CLUSTER), pp. 335–343 (2011)
16. Guyer, S.Z., Lin, C.: Broadway: a software architecture for scientific computing. In: IFIPS Working Group 2.5: Software Architecture for Scientific Computing (2000)
17. Haidar, A., Tomov, S., Arturov, K., Guney, M., Story, S., Dongarra, J.: LU, QR, and Cholesky factorizations: programming model, performance analysis and optimization techniques for the Intel Knights Landing Xeon Phi. In: 2016 IEEE High Performance Extreme Computing Conference (HPEC), pp. 1–7, September 2016
18. Harris, T., Maas, M., Marathe, V.J.: Callisto: co-scheduling parallel runtime systems. In: Proceedings of the Ninth European Conference on Computer Systems, EuroSys 2014, pp. 24:1–24:14. ACM, New York (2014)
19. Heinecke, A., et al.: Design and implementation of the Linpack benchmark for single and multi-node systems based on Intel Xeon Phi coprocessor. In: 2013 IEEE 27th International Symposium on Parallel and Distributed Processing (IPDPS), Washington, DC, USA, pp. 126–137 (2013)
20. Hugo, A.-E., Guermouche, A., Wacrenier, P.-A., Namyst, R.: Composing multiple StarPU applications over heterogeneous machines: a supervised approach. IJHPCA **28**(3), 285–300 (2014)
21. Kennedy, K., et al.: Telescoping languages: a strategy for automatic generation of scientific problem-solving systems from annotated libraries. J. Parallel Distrib. Comput. **61**(12), 1803–1826 (2001)
22. Kim, J., Kim, H., Lee, J.H., Lee, J.: Achieving a single compute device image in OpenCL for multiple GPUs. In: Proceedings of the 16th ACM Symposium on Principles and Practice of Parallel Programming, PPoPP 2011, NY, USA, pp. 277–288 (2011)
23. Kim, J., Seo, S., Lee, J., Nah, J., Jo, G., Lee, J.: SnuCL: an OpenCL framework for heterogeneous CPU/GPU clusters. In: Proceedings of the 26th ACM International Conference on Supercomputing, ICS 2012, pp. 341–352 (2012)
24. Komoda, T., Miwa, S., Nakamura, H.: Communication library to overlap computation and communication for OpenCL application. In: Proceedings of the 2012 IEEE 26th International Parallel and Distributed Processing Symposium Workshops and Ph.D. Forum, IPDPSW 2012, Washington, DC, USA, pp. 567–573 (2012)

25. Lee, J., Samadi, M., Park, Y., Mahlke, S.: Transparent CPU-GPU collaboration for data-parallel kernels on heterogeneous systems. In: Proceedings of the 22nd International Conference on Parallel Architectures and Compilation Techniques, PACT (2013)
26. Luk, C.-K., Hong, S., Kim, H.: Qilin: exploiting parallelism on heterogeneous multiprocessors with adaptive mapping. In: Proceedings of the 42nd Annual IEEE/ACM International Symposium on Microarchitecture, MICRO, NY, USA, pp. 45–55 (2009)
27. Majo, Z., Gross, T.R.: A library for portable and composable data locality optimizations for NUMA systems. ACM Trans. Parallel Comput. **3**(4), 20:1–20:32 (2017)
28. Marjanović, V., Labarta, J., Ayguadé, E., Valero, M.: Overlapping communication and computation by using a hybrid MPI/SMPSs approach. In: Proceedings of the 24th ACM International Conference on Supercomputing, ICS 2010, New York, NY, USA, pp. 5–16 (2010)
29. Ogata, Y., Endo, T., Maruyama, N., Matsuoka, S.: An efficient, model-based CPU-GPU heterogeneous FFT library. In: IEEE International Symposium on Parallel and Distributed Processing, IPDPS, pp. 1–10 (2008)
30. Pan, H., Hindman, B., Asanović, K.: Lithe: enabling efficient composition of parallel libraries. In: Proceedings of the First USENIX Conference on Hot Topics in Parallelism, HotPar 2009, p. 11. USENIX Association, Berkeley (2009)
31. Pandit, P., Govindarajan, R.: Fluidic kernels: cooperative execution of OpenCL programs on multiple heterogeneous devices. In: Proceedings of Annual IEEE/ACM International Symposium on Code Generation and Optimization, CGO 2014, NY, USA, pp. 273:273–273:283 (2014)
32. Phothilimthana, P.M., Ansel, J., Ragan-Kelley, J., Amarasinghe, S.: Portable performance on heterogeneous architectures. In: Proceedings of the Eighteenth International Conference on Architectural Support for Programming Languages and Operating Systems, ASPLOS 2013, NY, USA, pp. 431–444 (2013)
33. Planas, J., Badia, R.M., Ayguadé, E., Labarta, J.: SSMART: smart scheduling of multi-architecture tasks on heterogeneous systems. In: Proceedings of the Second Workshop on Accelerator Programming Using Directives, WACCPD 2015, pp. 1:1–1:11 (2015)
34. Ravi, V.T., Ma, W., Chiu, D., Agrawal, G.: Compiler and runtime support for enabling generalized reduction computations on heterogeneous parallel configurations. In: Proceedings of the 24th ACM International Conference on Supercomputing, ICS 2010, NY, USA, pp. 137–146 (2010)
35. Ravi, V.T., Agrawal, G.: A dynamic scheduling framework for emerging heterogeneous systems. In: 2011 18th International Conference on High Performance Computing (HiPC), pp. 1–10, December 2011
36. Rey, A., Igual, F.D., Prieto-Matías, M.: HeSP: a simulation framework for solving the task scheduling-partitioning problem on heterogeneous architectures. In: Dutot, P.-F., Trystram, D. (eds.) Euro-Par 2016. LNCS, vol. 9833, pp. 183–195. Springer, Cham (2016). https://doi.org/10.1007/978-3-319-43659-3_14
37. Satish, N., Kim, C., Chhugani, J., Dubey, P.: Large-scale energy-efficient graph traversal: a path to efficient data-intensive supercomputing. In: Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis, SC 2012, Los Alamitos, CA, USA, pp. 14:1–14:11 (2012)

38. Satish, N.: Can traditional programming bridge the ninja performance gap for parallel computing applications? In: Proceedings of the 39th Annual International Symposium on Computer Architecture, ISCA 2012, Washington, DC, USA, pp. 440–451 (2012)
39. Schaa, D., Kaeli, D.: Exploring the multiple-GPU design space. In: IEEE International Symposium on Parallel Distributed Processing, IPDPS, pp. 1–12 (2009)
40. Song, F., Dongarra, J.: A scalable framework for heterogeneous GPU-based clusters. In: Proceedings of the 24th ACM Symposium on Parallelism in Algorithms and Architectures, SPAA 2012, NY, USA, pp. 91–100 (2012)
41. Song, L., Feng, M., Ravi, N., Yang, Y., Chakradhar, S.: COMP: compiler optimizations for manycore processors. In: Proceedings of the 47th Annual IEEE/ACM International Symposium on Microarchitecture, MICRO-47, Washington, DC, USA, pp. 659–671 (2014)
42. Thomas, N., Tanase, G., Tkachyshyn, O., Perdue, J., Amato, N.M., Rauchwerger, L.: A framework for adaptive algorithm selection in STAPL. In: Proceedings of the Tenth ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, PPoPP 2005, pp. 277–288. ACM, New York (2005)
43. Tzenakis, G., Papatriantafyllou, A., Vandierendonck, H., Pratikakis, P., Nikolopoulos, D.S.: BDDT: block-level dynamic dependence analysis for task-based parallelism. In: Wu, C., Cohen, A. (eds.) APPT 2013. LNCS, vol. 8299, pp. 17–31. Springer, Heidelberg (2013). https://doi.org/10.1007/978-3-642-45293-2_2
44. Wu, W., Bouteiller, A., Bosilca, G., Faverge, M., Dongarra, J.: Hierarchical DAG scheduling for hybrid distributed systems. In: 29th IEEE International Parallel and Distributed Processing Symposium (IPDPS), Hyderabad, India, May 2015
45. Yu, H., Rauchwerger, L.: An adaptive algorithm selection framework for reduction parallelization. IEEE Trans. Parallel Distrib. Syst. **17**(10), 1084–1096 (2006)
46. Zandifar, M., Jabbar, M.A., Majidi, A., Keyes, D., Amato, N.M., Rauchwerger, L.: Composing algorithmic skeletons to express high-performance scientific applications. In: Proceedings of the 29th ACM on International Conference on Supercomputing, ICS 2015, pp. 415–424. ACM, New York (2015)