Loughborough University

Loughborough University
Institutional Repository

Impact of the scheduling strategy in heterogeneous systems that provide co-scheduling [extended version]

This item was submitted to Loughborough University's Institutional Repository by the/an author.

Citation: SUSS, T. ... et al., 2017. Impact of the scheduling strategy in heterogeneous systems that provide co-scheduling. IN: Trinitis, C. and Weidendorfer , J. (eds.) Co-Scheduling of HPC Applications. Amsterdam: IOS Press, pp. 142-162.

Additional Information:

- This paper was initially presented at the 1st Workshop on Co-Scheduling of HPC Applications, COSH@HiPEAC 2016, Prague, Czech Republic, Jan 19th 2016. This paper is published with kind permission of IOS Press. The publication is available at IOS Press through http://doi.org/10.3233/978-1-61499-730-6-142

Metadata Record: https://dspace.lboro.ac.uk/2134/37334

Version: Accepted for publication

Publisher: © The Authors and IOS Press

Rights: This work is made available according to the conditions of the Creative Commons Attribution-NonCommercial-NoDerivatives 4.0 International (CC BY-NC-ND 4.0) licence. Full details of this licence are available at: https://creativecommons.org/licenses/by-nc-nd/4.0/

Please cite the published version.

Impact of the Scheduling Strategy in Heterogeneous Systems That Provide Co-Scheduling

Tim Süß, Nils Döring, Ramy Gad, Lars Nagel, André Brinkmann [a,1] and
Dustin Feld, Eric Schricker, Thomas Soddemann [b]
[a] *Zentrum für Datenverarbeitung, Johannes Gutenberg University Mainz, Germany*
[b] *Fraunhofer SCAI, Schloss Birlinghoven, Sankt Augustin, Germany*

Abstract. In recent years, the number of processing units per compute node has been increasing. In order to utilize all or most of the available resources of a high-performance computing cluster, at least some of its nodes will have to be shared by several applications at the same time. Yet, even if jobs are co-scheduled on a node, it can happen that high performance resources remain idle, although there are jobs that could make use of them (e. g., if the resource was temporarily blocked when the job was started). Heterogeneous schedulers, which schedule tasks for different devices, can bind jobs to resources in a way that can significantly reduce the idle time. Typically, such schedulers make their decisions based on a static strategy.

We investigate the impact of allowing a heterogeneous scheduler to modify its strategy at runtime. For a set of applications, we determine the makespan and show how it is influenced by four different scheduling strategies. A strategy tailored to one use case can be disastrous in another one and can consequently even result in a slowdown - in our experiments of up to factor 2.5.

Keywords. scheduling, scheduling strategies, heterogeneous systems

1. Introduction

For several years now, multi-core processors equipped with powerful vector units are the standard in almost all parts of the computing world. They are available in cell phones, notebooks, desktop computers, servers and supercomputers. Additionally, GPUs and other architectures (Xeon Phi, FPGA, digital signal processors) are used in combination with normal processors to speed up suitable parts of an application. These *accelerators* mostly operate on separate memory spaces which requires time-consuming copy operations when the architecture is changed during a program run. At the moment, it seems as if this will not change in the foreseeable future. All these hardware architectures have in common that they only offer their performance benefits if developers write specific code for them and if they are able to exploit their inherent parallelism. Code for accelerators can, e. g., be created using OpenCL and domain-specific languages (DSLs).

[1] Corresponding Author: Tim Süß, Anselm-Franz-von-Benzel-Weg 12, 55128 Mainz, Germany; E-mail: suesst (at) uni-mainz.de.

In almost all systems, a large fraction of the accelerator hardware will be frequently idle and not optimally used. This happens when

1. all concurrently executed programs on a computer cannot make use of an accelerator due to the type of algorithm.
2. programs do not provide codes for the available accelerators.
3. a program cannot use its preferred resource because it is temporarily blocked by another application. The application may then be started on a less favorable resource. However, once a better resource becomes free, the program cannot be moved to this resource.

When the first situation occurs in a cluster environment, it can be solved by moving jobs between nodes or by already taking resource requirements into consideration during scheduling. If a resource is oversubscribed by multiple jobs on one node while the same resource is undersubscribed on another node, jobs can be migrated to balance the utilization. The second situation can obviously be avoided by providing codes for all concerned resources. Typically, a separate version of the program is needed for each resource. If multiple codes are available, the most suitable free resource can be chosen during runtime.

To tackle the third situation, it must be possible for a program to start its computation on one resource and move to another one later. Also a scheduler is required which manages the resources, assigns tasks to resources and migrates tasks. This way it can also prevent the oversubscription of resources.

However, if the *scheduling strategy* (the algorithm which decides when a computation is started or migrated) is static, it cannot exploit information provided by the program developers about the program's behavior at runtime.

VarySched is a scheduler that allows the scheduling of computations (denoted as *tasks*) on heterogeneous resources [26]. An application must register itself at the scheduler by implementing an interface. The interface requires only the set of codes for the different resources (denoted as *kernels*) and a ranking of these kernels. The ranking can correspond to performance, accuracy, energy consumption etc. We denote such a set of kernels as a *kernel collection*. The scheduler receives collections, chooses one of their kernels and schedules it to an available resource. This is similar to the behavior of the *Grand Central Dispatch* resource scheduler [4]. In contrast to the Grand Central Dispatch, VarySched allows to change the scheduling strategy. It even allows that an application provides its own strategy in form of a simple Lua script.

In this work, we evaluate the impact of different scheduling strategies on the makespan of programs. Four different types of strategies are tested:

long-term scheduler: aims to place all tasks on the resources they prefer most.

short-term scheduler: aims for using all resources permanently.

banking-based scheduler: is an extension of the short-term scheduler with an additional resource budget to control the availibility of resources.

constraint-based scheduler: similar to the banking-based scheduler but with a different computation for the limited resource budget.

In addition, we evaluate the positive and negative impact of the VarySched framework. In particular, we determine the costs caused by the scheduling strategies and by changing in between them. It turns out that the newly introduced overhead (for example,

due to scheduling decisions) is low compared to the gain, i.e., the increased throughput and the energy efficiency of systems. In the experiments, we use micro benchmarks, but also real-world applications and libraries.

2. Related Work

For unsigned a computers' full potential, jobs must be distributed among all available resources and they must be used in parallel, but not necessarily parallel within a single application.

Approaches as DAGuE [8], Elastic Computing [29] or StarPU [5] support multi-threaded applications. Others address multi-application thread scheduling like ADAPT [17], but they are limited to the CPU side of the problem. Other approaches' ideas are similar to ours. DAGuE and StarPU rely on Directed Acyclic Graphs (DAGs) to determine an optimal schedule, e. g. by utilizing task-graphs. All approaches have in common that code changes may be necessary.

Sun et al. have introduces a task queuing extension for OpenCL that improves the performance within a heterogeneous environment [25]. Anyway, this approach does not allow the use of external code generators or other ways of utilizing its scheduling system.

Menychtas et al. suggest a fair scheduling scheme for accelerators that monitors resources and that intercede when the usage of resources is unbalanced [19].

The Grand Central Dispatch (GCD) [4] is able to schedule sub-tasks (like functions) individually. However, its scheduling has no global view which limits the schedule's quality.

Beisel et al. [6] introduce a resource-aware scheduler, capable to distribute tasks among different hardware resources as VarySched. In contrast to VarySched the scheduler uses always the same, static scheduling function.

Many accelerators require hardware-specific codes. Aside from hardware-specific programming languages like VHDL or CUDA, techniques like OpenCL or OpenACC create codes for accelerators [20,1]. To exploit the resources' full potential, developers have to optimize the code taking hardware-related properties into account. OpenCL, e. g., has been developed to unify programming for different devices, but even if this framework is used, hardware-specific code might be needed. Additionally, OpenCL can only be used if the programming interfaces for the devices are provided.

Another approach was introduced with *Heterogeneous System Architecture (HSA)* [21]. Here CPUs and GPUs are coupled on a single chip to accelerate processes. This fusion allows to program the GPU with C++ or OpenCL. AMD provides a HSA SDK that will contain a abstraction layer the *HSA Intermediate Language (HSAIL)* which allows C++ optimization for GPU computing.

Our framework includes tools to perform automatic code transformation, optimization, and parallelization of C source code through the polyhedral model. This model has been well studied and numerous source-to-source compilation tools evolved from it, e.g., PluTo [7], PPCG [28], Par4ALL [24] (based on PIPS [2] allowing polyhedral and non-polyhedral transformations), PluTo with the SICA extension [10,11] or the ROSE compiler infrastructure [23] with the PolyOpt/C optimizer. Even though these frameworks traditionally aim for an automatic OpenMP and SIMD parallelization of sequential CPU codes, several are capable of CUDA code generation for NVIDIA GPUs.

Recently, novel developments were published in the field of polyhedral code optimization that are related to improved GPU code generation [15,14] and to the aim of extending the range of applicable codes in the polyhedral model [27].

Using different hardware components is also an issue in distributed computing. Desktop grids a BOINC allow to provide different codes for different environments [3]. Another systems that allows the utilization of different hardware components is Ibis/-Constellation [18]. The application is divided in so-called *activities* capable to run in different environments and (if necessary) to communicate with each other. The presented systems distribute one application among different compute nodes (maybe with different architectures) and perform the computations there. However, they do not take other applications into account as VarySched does. Thus, resources which have been unavailable at the beginning of computation cannot be utilized if they get free during the calculation.

3. The VarySched Framework

The VarySched framework consists of a task scheduler (the *VarySched daemon*), a programming interface (*VaryLib*), and a scheduling strategy interface. The heart of the VarySched framework is the task scheduler which schedules tasks according to the (current) scheduling strategy. Applications register their tasks with the daemon via VaryLib in the form of so-called kernel collections. A *kernel* is the program code of a task for a particular resource. A *kernel collection* is the set of all kernels belonging to a task and also includes a set of data mover routines which copy the task-related data when a task changes the resource. For the registration and all further communication messages are placed in *message boxes* provided by the daemon.

The strategy which provides the scheduling algorithm decides the spatial and temporal placement of the tasks (i.e., the resource and the start time) and triggers the execution of the respective kernels. The scheduling strategy / algorithm is given as a Lua script implementing the scheduling strategy interface. It is called by the daemon to create a new schedule or update the current one whenever, for example, a task finishes, a new task arrives, or the strategy is changed. The latter is done by modifying the Lua script which is even allowed at runtime. The registration, scheduling, and execution of a task is schematically shown in Figure 1. Tasks of different applications can be handled in parallel.

The developer / user influences the process by defining application- or device-specific strategies which, for example, aim to reduce the overall energy consumption or the make-span of a set of programs / tasks. For making decisions, these strategies can take any parameters into account. Examples are the queue lengths of the resources (i.e. the number of tasks waiting to be processed), the current and predicted energy usage, or the suitability of a resource. For the latter each task can be assigned *resource affinities* which rank the resources based on the (expected) performance that the task would achieve on them.

In the case of a long-lasting task (for instance, the multiplication of large matrices), it might be advisable for the programmer to make the task interruptible. The respective kernel then interrupts its execution from time to time to communicate with the VarySched daemon. This makes it possible to migrate a task to a more suitable resource or to assign resources more fairly.

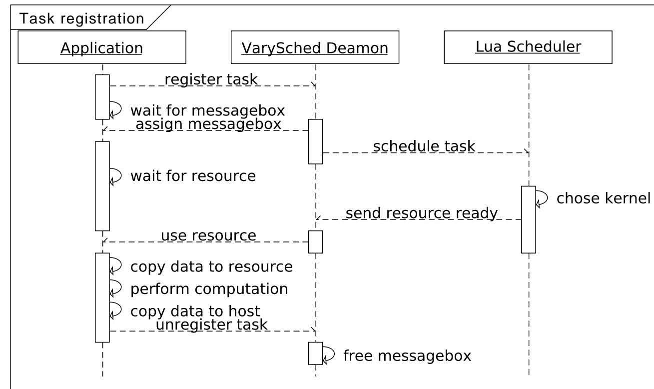

Figure 1. Registration, scheduling, and execution of a task using VarySched.

In the following we will provide more details about the inner workings of the framework, the scheduling process and the execution of the kernels. Note that beyond the registration of tasks (and possibly the implementation of scheduling strategies), scheduling and execution details are completely hidden from the programmer.

3.1. Scheduling of tasks

While applications register their kernel collections with the VarySched daemon, the kernels themselves are executed in the original memory spaces of the applications unless the kernels are moved to other nodes. The exchange of control messages between scheduler and applications is performed via a shared memory segment. This segment consists of two parts: a *registration slot* and a *message box array* (see Figure 2). During registration, an application provides its kernel collection to VarySched by placing a request in the registration slot. It is then assigned a message box if one is available; otherwise the application waits (synchronously/blocking) for one. All further communication between client and VarySched is exchanged via that message box.

When a kernel collection is registered, the schedule is reevaluated and a best matching resource is picked. For flexibility, the scheduling strategy must provide a capacity-limited queue (denoted as *resource queue*) for every compute resource representing the execution order. These task queues contain all tasks (task: kernel + data handling) which are scheduled for an associated resource. The result of the scheduling decision is communicated to all applications with imminent or paused kernel executions which, in turn, start their selected kernels on the (re-)assigned resources.

If necessary, the kernel copies all required data to the resource's memory, informs VarySched that it has accessed the resource, and starts the computation. In the case that another task is scheduled to a occupied resource, VarySched sends a rescheduling request to the occupying task. The occupying task receives this request during an interrupt, copies all its data to the host and frees the resource.

3.2. Implementation of scheduling strategy

The scheduling strategy can be considered a target function of an optimization problem. It must be provided in form of a Lua script which, if necessary, can access relevant

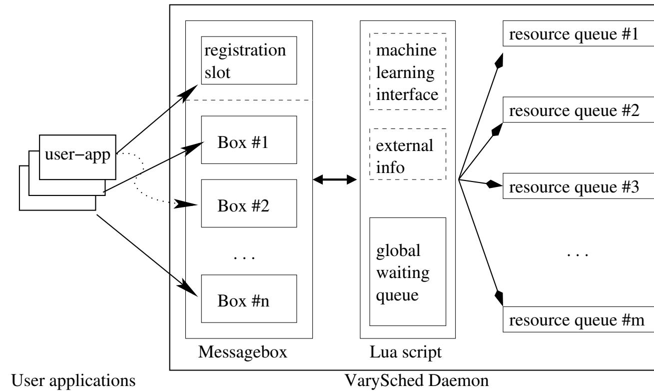

Figure 2. Architecture of VarySched. Applications register, the scheduler takes suitable kernels and schedules them.

performance parameters and any parameters defined by the developer. The strategy can be modified during runtime by changing / replacing the Lua script. Although Lua is an interpreted language, the time for creating a new schedule is usually negligible (see Section 6.1.1).

Furthermore, it is possible to express task dependencies in the Lua script. The scheduler can build a dependency graph and create a schedule on top of it. Such a scheduler is informed about dependencies by messages and must therefore implement the Lua function `schedule` that receives all incoming messages. The `schedule` function is called whenever a new task is registered or a running task finishes.

As mentioned, every resource has its own *resource queue*, a task queue that contains all tasks assigned to the resource. Additionally there is a global waiting queue for tasks which have not yet been assigned to a resource. Since Lua stores queues in a state object in C++ code, the queues are persistent between different calls of the script. The lengths of the resource queues can be modified at runtime to change the scheduler's behavior.

New components to support the scheduling process can be added to VarySched and dynamically activated at any time. This includes event sources (like sensors), but also interfaces to machine learning applications or statistical evaluations.

3.3. Programming Interface

As described, the scheduler operates with kernel collections consisting of kernels that are tuned for different resources. For optimal operation, our scheduler ideally has kernels available for all present resources.

An example code for the registration of a kernel collection is shown in Figure 3. It includes a pointer to an object containing the data required by the task. At the end of the computation, this object also contains the result.

For moving data, each kernel must provide two functions (parameters 3 and 2 of `Kernel`):

- one function to copy data from the host to the device / resource (e. g. a GPU) before the kernel is executed, and
- one function to copy the results back to the host.

```
void regFunc(int* _data, size_t _size) {
/* Kernelcall insertion begin*/
  KernelData data = KernelData();
  data.data = (void*)_data;
  data.size = _size;
  data.sizeInBytes = _size * sizeof(int);
/* The first parameter is the kernel, the second is the
copy-to-host function, the third is the copy-to-device
function, the fourth specifies the resource, and the last
is the resource affinity */
  Kernel k1(cpuKern, sCTH, sCTD, CPU, HIGH);
  Kernel k2(gpuKern, gCTH, gCTD, GPU, LOW);
  Kernel k3(ompKern, mCTH, mCTD, MPU, MED);
  Kernel k4(netKern, nCTH, nCTD, NET, NEVER);

  KernelCollection kc(&data);
  kc.registerKernel(&k1);
  kc.registerKernel(&k2);
  kc.registerKernel(&k3);
  kc.registerKernel(&k4);

  TaskRegistration tm(&kc, 100);
  tm.run(&data);
/* Kernelcall insertion end*/
}
```

Figure 3. The original function is extracted and replaced by code to register and call the different kernels. In this way, the scheduler is responsible for its execution.

Depending on the hardware architecture, they can be empty (e. g. in case of a zero-copy). Additionally, these functions can also be used to reorganize the data for the specific needs of a resource and thereby improve its performance.

If data must be copied from one memory space to another, data structures may have to be serialized before their transfer. Codes for these operation must be inserted, too, if necessary. One can use serialization tools or libraries like *boost*.

No code will perform equally well on all resources in terms of speed and/or energy consumption. Hence, usually there is a preference (affinity) as to where a specific kernel should ideally be run. As mentioned, these affinities are also communicated to the scheduler (parameter 5 in Figure 3).

3.3.1. Life cycle of a task

After registration, a task waits until a resource becomes available. Once it is assigned to a device, the task proceeds and informs the scheduler that it accesses the resource. If necessary, all required data is copied to the device and the task's computation starts by executing the appropriate kernel. At the end of the execution, all resulting data is returned to the host and the task deregisters itself.

VarySched allows to pause computations and proceed them later, maybe on another resource. Instead of deregistering itself, a task can pause and asks the scheduler if the

```
/* Extracted kernel */
int kernel(KernelData* _kd) {
  ...
  return 0;
}

int main(int argc, char** argv) {
/* Before program is started, required data are transferred.
They are loaded when generated program is started. */
  KernelData kd = readDataFromFile();
/* Extracted kernel executed in loop until it finishes. */
  while(!kernel(&kd));
/* Results are stored on disk and sent back after termination
of program. */
  storeResultToFile(&kd);
  return 0;
}
```

Figure 4. The kernel call is surrounded by two functions, one that loads the input data and one that stores the results. The kernel is executed until it has finished its computation.

resource is claimed by another task. If no other task is waiting for the resource, it simply proceeds. Otherwise the task's data is returned to the host's memory, the scheduler is informed that the resource is available, and the task is rescheduled. This feature allows fair scheduling strategies.

3.3.2. Automatic code generation

Instead of using resource-specific libraries or writing all functions for the different architectures manually, domain-specific source-to-source compilers like Pluto [7] or PPCG [28] can be used to generate the program codes for the different devices. The generated program parts can be integrated into the target code and prepared for registration at the scheduler by automatically injecting the required program codes.

3.4. External nodes as resources

Next to node-internal hardware components, other computers can be resources, too (if they are connected via network). VarySched provides the ability to move tasks to other nodes. We have developed a compiler preprocessor that generates a program which packs the required data and executes the kernel on a remote node. This requires to enclose a kernel with `#pragma BEG_VARYSCHED` and `#pragma END_VARYSCHED` (similar to the markings in polyhedral transformators as pluto or ppcg). The preprocessor extracts the marked kernel and injects it into a new program. The new program's main function first loads the input data, then executes the kernel function using this data, and finally, when finished, stores the results before they are sent to the original host node (see Figure 4).

To execute the program remotely, the program and its data are (possibly) serialized and then copied to another host using *scp* (see Figure 5). After the program's execution, the results are serialized, copied back to the first host and deserialized.

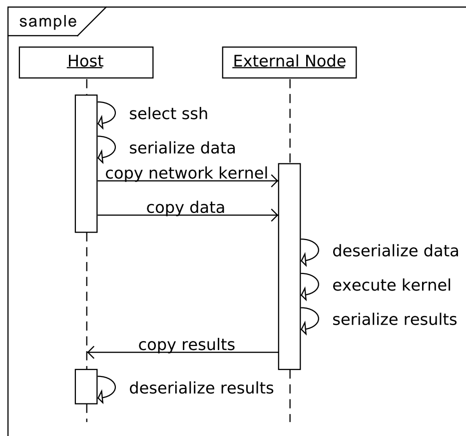

Figure 5. Tasks can be migrated to other nodes via ssh. Data and kernel are copied to external resources, the kernel is executed, and the results are copied back.

The mechanism for copying the data could be replaced by a more advanced and better performing solution in the future (for example, a parallel file system).

At its current state, VarySched does not coordinate the usage of external nodes. This is planned for later versions, and our evaluation does not cover this feature.

4. Scheduling Strategies

The VarySched scheduler is used to evaluate the impact of different scheduling strategies on the performance. VarySched is a newly developed task scheduler which will be published in the near future. In this section, we shortly describe the scheduling strategies and applications that we use in our tests. We use four scheduling strategies with different aims. We define two different governors (named low and high) which determine the type and the amount of resources that can be used.

4.1. Short-term Scheduler

The short-term scheduler aims for using all resources permanently. While it focuses on keeping all resources busy, the selection of a good kernel is secondary. It does not use the resource governor's state for the scheduling decisions.

At first, incoming tasks are placed in the global queue which is not associated to any resource. All resource queues contain only a single task that is processed instantly when it arrives. If a resource ρ becomes free, the scheduler traverses the global queue and searches for the first task that has the best performance on resource ρ (with respect to the strategy). This task is then scheduled on ρ and the current scheduling phase terminates. If there is no such task, the scheduler traverses the global queue again, searching for a task whose second preference is ρ, and so on.

4.2. *Long-term Scheduler*

The long-term scheduler aims to place all tasks on the resources they prefer most. Additionally, it tries to fill the queues such that the queues' work off requires similar time. Depending on the resource governor's state, the long-term scheduler masks different resources to stay unused. In our tests, if the governor's level is high, jobs can be scheduled on all resources; if the level is low, only the CPU cores can be used (e. g. for energy reasons).

The global queue contains only a single task t while the different resource queues can contain an arbitrary amount of task. The scheduler determines the length of the resource queue l_1 that t prefers the most. Then it determines the length of the resource queue l_2 that t prefers the second most. If $l_1 \leq \delta \cdot l_2$ (whereby δ is the performance factor between the resource that t prefers the most and the resource that t prefers the second most) t is schedule on its first choice. Otherwise the procedure is repeated with t's second and third preferences and so on until it reaches the least preferred resource.

4.3. *Banking-based Scheduler*

The banking-based scheduler assumes that each available resource has a limited budget of credits. Running a kernel on a resource costs a certain amount of credits. If a resource's budget suffices to bear the costs of a kernel, the respective amount of credits is removed from the budget and the task is scheduled to that resource. The scheduler starts with the most preferred resource and proceeds successively with the following resources. If no resource has a sufficient budget to take the task, the task stays in the global queue. The budget is refilled over time. After a certain amount of time, credits are added to the budget until the maximal budget limit is reached.

In our tests we start with a full budget of one hundred credits. Every five seconds 15 credits are added to the budget if the resource governor is set to high, and ten credits are added if the governor is set to low. Running a task on the GPU costs ten credits; five credits are needed for all CPU cores, one credit for a single CPU core.

4.4. *Constraint-based Scheduler*

The constraint based scheduler assumes that every incoming job consumes resources denoted as credits and has an overall credit limit. The credit sum of concurrently running jobs must not exceed this predefined limit. The aim of this schedule is to provide a constant upper boundary for currently used resources. The resource queue of every device can hold only a single job. Incoming jobs are scheduled until all resources are used or the overall credit limit is reached. If one of these conditions is met, incoming jobs will be enqueued in the global queue until a resource has been freed and the free credits are sufficient.

In our tests, a task on the GPU costs nine credits, on all CPU cores six credits, and three credits on a single core. The credit limit is set to 18 if the governor is set to high and nine if it is set to low.

4.5. Test Environment and Applications

We tested two applications on a NVIDIA Jetson-TK1 system. The tests have been performed with two different resource governor states as described before. In our tests performing a matrix-matrix multiplication, we scheduled one hundred instances of the same application. For the LAMA application, we scheduled 25 instances.

Matrix-matrix multiplication Our first test application performs a matrix-matrix multiplication. The matrices are quadratic and contain 1024×1024 single-precision floating point values. The performed algorithms consists of three nested loops iterating over the two matrices. To generate parallelized versions this code automatically we used Pluto-SICA [10,11] and PPCG [28]. These tools generated the resource specific kernels which are encapsulated in a kernel collection.

This scenario shows how VarySched can be used in collaboration with automatic code generators. Therefore, a system's resources can be exhausted without the need of programming the different code versions for multi-core CPUs and GPUs manually. This is possible for this matrix-matrix multiplication example as the code sufficiently simple and, hence, manageable by the aforementioned tools.

LAMA application LAMA [16] is an open source C++ library for building efficient, extensible and flexible solvers for sparse linear systems. Once a LAMA solver is written, it can be executed on various compute architectures without the need of rewriting the actual solver. LAMA supports shared and distributed memory architectures, including multi-core processors and GPUs.

For our tests, we use a conjugate gradient solver to solve an equation system resulting from discretizing Poisson's equation with a 3-dimensional 27-points (and therefore very sparse) matrix. The number of unknowns is $50 \cdot 50 \cdot 50 = 125000$. The CG algorithm is one of the best known iterative techniques for solving such sparse symmetric positive (semi-)definite linear systems [22]. It is therefore used in a wide range of applications (e.g. Computational Fluid Dynamics (CFD) or oil and gas simulations). The used kernel collection contains of three different kernels: one on a single CPU core, one with OpenMP, and one on the GPU and a single CPU core.

This scenario shows, in contrast to the previous one, how VarySched can be used to schedule instances of one application that with available implementations for the different resources, in this case provided by a library.

Scheduling strategy tests The Jetson-TK1 is an ARM-based (Cortex-A15, four 32-bit cores, 2.3 GHz) system equipped with a 192-core Kepler GPU (GK20A). Additionally, the board provides 2 GiB main memory, shared and accessible by CPU and GPU. We use two different Linux operating systems with different CUDA versions. For the matrix-matrix multiplications we use Ubuntu 14.04 and CUDA-6.5 and for the LAMA tests we use Gentoo and CUDA-6.0.

VarySched impact Tests Most of the VarySched impact tests are executed on computers equipped with an Intel Core i7-2860QM CPU (4 cores, 8 threads, 2.5 GHz), 8 GiB RAM, and an NVIDIA Quadro 2000M. The operating system was Linux Mint 16, 64-bit, kernel version 3.11.0-12. The bitcoin miner test was run on an Intel Sandybridge i7-2600 CPU (4 cores, 8 threads, 3.40GHz), 8 GiB RAM, and an NVIDIA GeForce GTX 750. The operating system was Ubuntu Linux 14.04.2 LTS (Trusty Tahr), 64-bit, kernel version

3.16.0-52. The compilers and libraries were gcc-4.8.1, llvm-3.4, clang-3.4, pluto 0.9.0, sica-0.9.0 and ppcg-0.01-53 as well as CUDA-5.0 and libssh2. If not differently stated, we use the short-term scheduler in our tests.

5. Evaluation of the Strategies Impact

In this section we evaluate the impact of the scheduling strategies (Section 4) on the applications' (Section 4.5) execution by running respectively 100 or 24 instances on one node in parallel. The experiments are performed for both governors and the quality of the schedules is measured by the makespan which is the time necessary to process all jobs.

The experiments are conducted as follows: All instances are started at approximately the same time in the beginning. One after another, the jobs register at the VarySched daemon and the scheduling strategy determines for each job which of their kernels is to be executed.

5.1. Matrix-Matrix Multiplication

The total execution time for all matrix-matrix multiplications are displayed in Figure 6 for the different governors. An important observation is that the makespan of all scheduling strategies is almost the same when the governor is set to high and consequently all resources can be used. Additionally, the makespan increases when the resource governor's setting is lowered from high to low. Only for the short-term scheduler the makespan stays almost constant independent of the governor's state. This can be explained by the way this scheduler works. As it always tries to utilize all available resources, the governor's setting has no influence on the schedule.

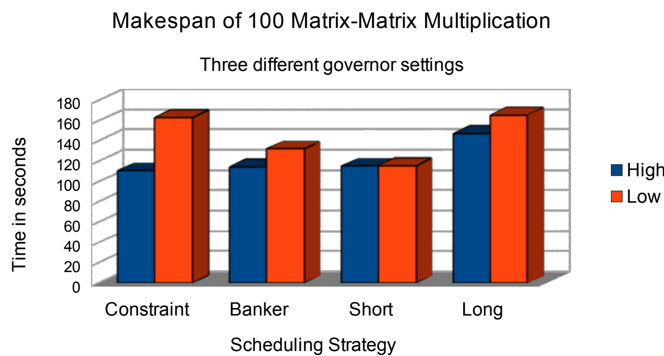

Figure 6. Makespan of one hundred instances of the matrix-matrix multiplication application with different governor states.

The decreasing performance of the other schedulers in the low governor state can be explained with the credit base working approach of the credit-based strategies and by the governor's setting respecting behavior of the long term scheduler.

The available budget of the constrained-based scheduler and the banking-based scheduler is decreased when the governor's state is lowered. While the constraint-based scheduler's credit limit is 18 if the state of the governor is high, it is nine in the low

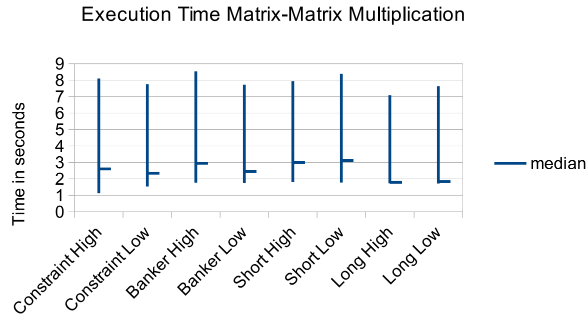

Figure 7. The runtime of the governor considering schedulers decreases in the governor's state is lowered.

state. Using this strategy, the high state allows to utilize all available resources while the low state allows only the utilization of either nine times a single core or all cores in an multi-threaded kernel and an additional single core kernel.

In this respect, the banking-based scheduler behaves differently as its performance is relatively smaller if the resource governor's state is lowered. This can be explained by the way how credits are added to the budget and when a task is scheduled. In both states, every five seconds a constant number of credits is added to the budget. In the high state 15 are added and in the low state ten. A task is scheduled when a sufficient amount of credits for a resource is available and, if the credits are only sufficient for the slowest resource, the kernel for this resource is chosen. Thus, 15 credits are sufficient for a GPU and multi-threaded kernel, ten credits are enough for one multi-threaded and one single-threaded kernel, and five credits are still sufficient for one multi-threaded or one single-threaded kernel. If the resource governor is set to low, the constraint based scheduler's is about 1.4 times faster than program execution using the long-term scheduler.

The long-term scheduler achieves the worst results of all scheduling strategies when the resource governor is set to high. This can be explained by the fact that the last jobs to run are scheduled and executed on the slowest resource because there are some inaccuracies in the performance factors between the different resources. Nevertheless, the result with low governor setting is similar to the result of the constraint-based scheduler.

But the increase of the makespan alone with different governors does not show the positive impact of co-scheduling. Since in most scheduling strategies the number of utilizable resources is reduced if the governor is lowered, the total execution time (the makespan) increases. However, this is even the case if the median of the applications' execution time decreases. Figure 7 shows that the runtime of a single matrix-matrix multiplication decreases if less resources can be used.

The increase of the single application's performance has two reasons: 1) The size of the matrices is small so that much of the time during the matrix-matrix multiplication is spent on copying data in the case of executing on the GPU. 2) The GPU versions also use CPUs a little bit and thereby influence CPU kernels. If there are no GPU kernels, then the cores are not shared.

When the governors are set to high, all resources are used and applications share and compete for these resources so that applications might block each other; while in the

case, where the governors are set to low, less resources are used and applications execute more sequentially.

5.2. LAMA Application

As in the previous section we first analyze the makespan of the LAMA application. When scheduling this application, the results for the short-term scheduler are similar to ones of the matrix-matrix multiplication (see Figure 8). The short-term scheduler does not consider the governor's state and the application's preferences, thus all test runs have similar makespans.

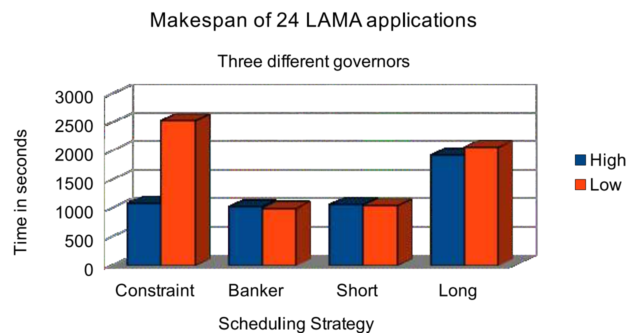

Figure 8. Makespan of one hundred instances of the LAMA application with different governor states.

The applications suffer the most if the constraint-based scheduler is applied and the governor is lowered. The total execution times more than double if the usage of the GPU is prohibited.

The banking-based scheduler behaves differently. While the makespan for the matrix-matrix multiplication increases when setting the governors state to low, it is almost constant in case of the LAMA application. This can be explained by the required runtime for one application which is significantly higher than for the matrix-matrix multiplication. Due to the high runtime, the scheduler's budget can be refilled sufficiently fast, for which reason all resources can be used.

In case of the long-term scheduler, there is the same issues as for the matrix-matrix multiplication. The performance factors between the different resources are not well-adjusted and, thus, this scheduler achieves the worst results.

The median of the runtimes varies for three of the schedulers when changing the governors state (see Figure 9). While for the matrix-matrix multiplication the median only stays constant for the short-term scheduler, for the LAMA application it stays constant for the banking-based scheduler, too. This was expected from the previous results of the makespan. At maximum, using the banking-based scheduler is 2.5 times faster than using the constraint-based scheduler.

The median of the application runtime increases if the long-term scheduler is used. In comparison to the previous tests, the time required to copy the necessary data to the GPU can be compensated by the accelerated computation.

Findings: Co-scheduling can reduce the makespan of parallel executed applications. It has a positive impact on the systems performance, even in the case when

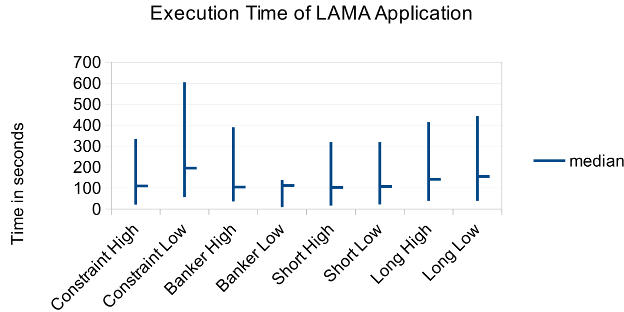

Figure 9. The runtime of the LAMA applications with different governors.

the median runtime of a single application slightly decreases if number of used resources is increased.

6. Evaluation of VarySched

In this section, different aspects of our framework are evaluated. First, we examine the overhead inferred by VarySched. Then, we show by comparison with other systems that the gain from using VarySched is nevertheless considerable. Finally, we analyze the influence of different scheduling strategies and show that choosing/designing the right strategy makes a difference. In order to quantify the impact, we measure the make-span, the energy consumption, and the overhead introduced by the scheduler.

We used manually written and automatically generated codes (by PluTo-SICA [10, 11] and PPCG) in Section 6.1.1-6.2.2, library-based codes in Section's 6.2.2 second part, and a program created from two hardware-specific versions of a bitcoin miner [13,9] in Section 6.2.3.

6.1. Overhead Analysis

6.1.1. Scheduling time

Preparing data required for the computation and registering kernels for different resources require time in addition to the net computation. Furthermore, VarySched's Lua scheduler introduces another overhead to assign the kernels to the different resources. During the phases in which the scheduler organizes the system, the actual applications are paused. We can distinguish two types of events during which an application is idle, namely

- preparation of data and kernel, and
- waiting for an available resource.

We cannot avoid the waiting times, but we can make sure that the overhead introduced by the preparations is as small as possible. Gad et al. showed in [12] that the serialization of the required data can be done efficiently. This is why we only look at the overhead of the task registration. For this, we start multiple jobs in parallel and measure

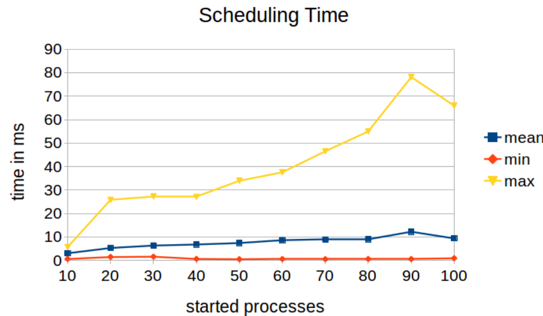

Figure 10. When many tasks want to get registered simultaneously, the first tasks are handled fast while later ones have to wait longer.

the registration time. We start with ten parallel applications and increase the number in every step by ten (see Figure 10).

The time required to register a kernel at the scheduler increases with the number of tasks running in parallel. This can be explained by the fact that each task has to first access one of the 100 registration slots. Thus, if the number of tasks is increased, they have to wait longer on average until a mailbox is assigned to them.

In order to fulfill the defined requirements, the schedule also needs to be recomputed every time there is a change in the environment such as the possible violation of a predefined energy envelope constraint. In the trivial case where only one task is scheduled per resource, no rescheduling is required because the resource queues are empty. In our tests, the rescheduling of ten jobs took about $20\,\mu s$. After increasing the number of jobs to one hundred, the rescheduling required about $30\,\mu s$. The number of jobs which are handled by the scheduler is sufficiently low, thus, the schedules can be reorganized sufficiently fast.

Findings: Tasks might wait a while until they are finally scheduled. However, scheduling tasks for resource utilization is fast as well as the rescheduling of them when the scheduling strategy is changed.

6.1.2. Kernel interruption for rescheduling

VarySched allows to interrupt kernels and to reschedule them. However, moving a task from one resource to another requires some time due to the evaluation of the target function and possible data movement to other memory spaces.

To measure the introduced overhead, we used a testing kernel performing simple vector-vector additions. The computation was interrupted after a defined amount of elements had been summed up while vectors consisted of 30 k single-precision floating point elements in all tests. The kernel is resumed directly after the interruption.

First, the kernel was interrupted each time when 128 additions were performed (234 interrupts in total). We compared the interrupted execution of the vector-vector addition with the uninterrupted one.

Since the calculation of this highly parallel kernel finishes relatively fast on the GPU compared to the necessary additional copy operations, the overhead should become apparent. To emphasize that, we performed an additional test where we interrupted the kernel after 3 k additions (10 interrupts in total). We did not move the computation to

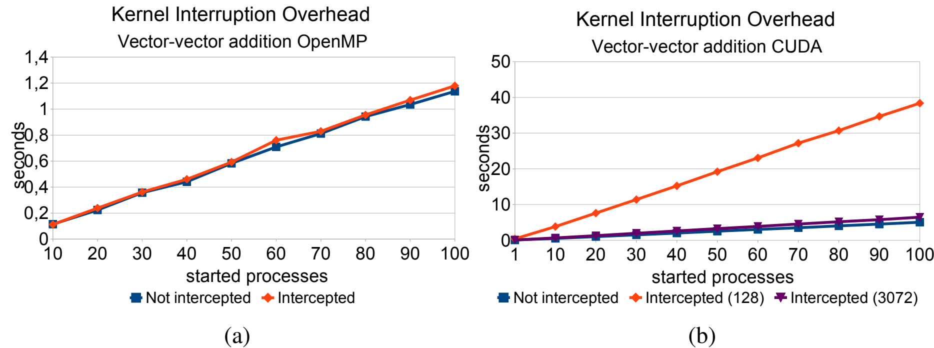

Figure 11. (a) If a task is interrupted while it is executed on the CPU, it is not significantly slowed down. (b) Interrupting GPU tasks produces a significant overhead.

another resource during the execution in both scenarios because we are only interested in the interruption overhead.

When performing the computation on many CPU cores, the overhead is very small (see Figure 11a).

If only the CPU is used, the loop can be interrupted, the message box can be checked, and the kernel can proceed (a single interruption lasts about 0.0004 seconds). In contrast, if the GPU is used (see Figure 11b), this interruption causes a significant overhead (one interruption takes about 0.014 seconds). Besides the initial and final copy operations, it is additionally necessary to stop the kernel and to relaunch it. If this is done too often, the impact on the overall runtime is massive.

If a GPU kernel should be interrupted in such a way, it is therefore necessary to carefully adjust the frequency of interrupts. We set the number of the interrupts during a task execution to ten for our final benchmarks. This produced a relatively small and acceptable overhead compared to the case that the task is processed continuously (compare Figure 11b).

Findings: Kernel interruptions cause low overheads on the CPU. On GPUs the interval between interruption must be chosen wisely to reduce their impact on the execution time.

6.2. Analysis of Advantages

6.2.1. Execution time

We use three basic applications to test how much time is saved in total when VarySched is applied. The first application multiplies two matrices, the second one adds two matrices, and the third one computes the heat distribution of a continuously heated plate over time. The plate is represented by a regular 1024^2 grid. The temperature change of a point is iteratively computed as the average temperature of its direct neighbors; 200 iterations were performed.

In those tests, we compared the duration of starting many instances of the applications in parallel with our scheduler to the time required if the applications are started sequentially, each simply using the fastest available resource. In other words, we test how the make-span can be reduced if some instances of the applications are computed on slower resources, but in parallel to the others. Since the scheduler and the OS run beside

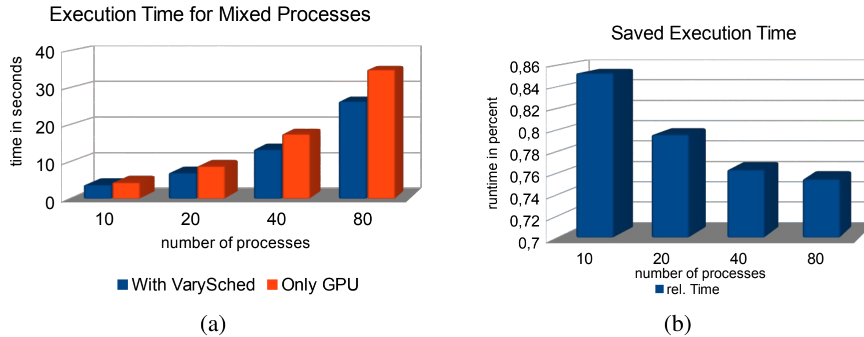

Figure 12. The more jobs, the more time is saved by VarySched.

our applications, we allow the OpenMP-processes to use only six threads. We assign the task on the different resources with the following distribution: 70% of the jobs use the GPU, 25% of the jobs use OpenMP, and the remaining jobs use only a single thread (with SICA optimized SIMD- and Cache-usage). We choose this configuration to saturate all resources within the node and to demonstrate the potential performance improvements caused by VarySched.

In Figure 12a we can see that the overall execution time increases if more jobs are started. However, we can also see that time is saved every time the scheduler is used.

Findings: VarySched allows to increase the throughput of a system.

6.2.2. Power consumption

In the next test, we evaluated the power and energy consumption. One has to take into account that we used a low power graphics adapter in our evaluation. To achieve higher GPU performance, faster adapters could be used which would, in turn, consume more power. In our test system, we measured about 44 W in idle state without the GPU, and the GPU itself consumed up to 55 W.

Test using the LAMA library LAMA [16] is an open source C++ library for building efficient, extensible and flexible solvers for sparse linear systems. LAMA supports shared and distributed memory architectures, including multi-core processors and GPUs.

For our tests, we used a conjugate gradient solver to solve an equation system resulting from discretizing Poisson's equation with a 3-dimensional 7-points (and therefore very sparse) matrix. It is used in a wide range of applications (e. g. Computational Fluid Dynamics (CFD) or oil and gas simulations). We performed the computation in five different configurations: on a single CPU core, with OpenMP on six cores, on the GPU and a single CPU core, on the GPU and two CPU cores, and finally with VarySched.

Figure 13 and Table 1 show that VarySched achieves the best results: it is not only the fastest configuration, but also the one with the lowest energy consumption. Using VarySched, 25.4 kJ are consumed while a single CPU core consumes 57.8 kJ.

The curves' integrals in Figure 13 represent the energy (in $joule = watts \cdot seconds$) consumed for ten program runs. When using VarySched, all provided resources (6 threads on the CPU as well as the GPU) can be used in parallel, at each point in time depending on how good the current code fits to the different available hardware devices. Thus, the peak power intake of VarySched is the highest of all the configurations whereas

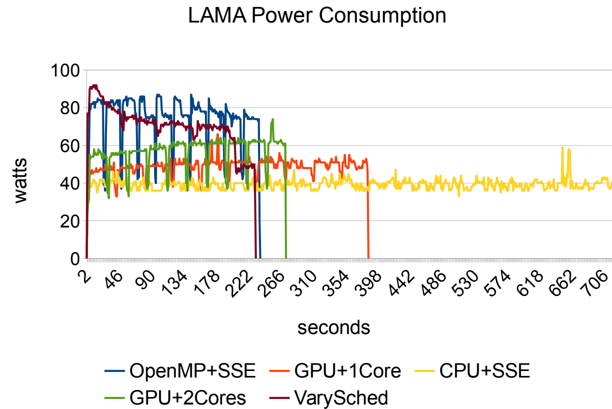

Figure 13. Power consumed by LAMA application with and without VarySched.

Table 1. Energy consumption of LAMA application using different configurations.

	CPU + SSE	OMP + SSE	GPU + 1 core	GPU + 2 cores	VarySched
Sys. total	57,843 J	27,244 J	34,347 J	26,492 J	25,486 J
App. total	28,272 J	17,527 J	18,644 J	15,340 J	16,097 J
Sys. avg. watt/sec.	80.00	114.95	89.68	97.40	109.85

the overall consumed energy of the system is with 25.4 kJ the lowest of all with VarySched (see Table 1).

Findings: VarySched allows to reduce a systems power consumption for the same work.

6.2.3. Merging two programs for flexibility

Mining bitcoins is a number crunching task with implementations for many hardware platforms. The program codes can be easily integrated into our framework because the internal computations are independent of each other.

We compare the performances of a multi-core CPU, a CUDA-capable GPU, and VarySched having access to both. The used bitcoin miner for GPUs is a fork of the used CPU miner [13]. The programs' main functions were renamed to `runCPUMiner` and `runGPUMiner` which are called by the CPU and GPU kernel, respectively. We used the internal counters provided by the original bitcoin miner programs.

The measurements in Figure 14 show that the number of hashes/second using the VarySched framework roughly equals the sum of hashes/second of the other two. This implies that the make-span of the VarySched system is also shorter.

Findings: Different resource specific versions of an application can be combined easily with VarySched. Thus, a computers full potential can be exploited more easily.

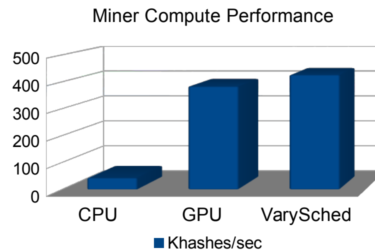

Figure 14. Throughput of bitcoin miner on a CPU, on a GTX 750 GPU, and on the combination of both using VarySched.

7. Conclusion

We have shown that the scheduling strategy has a high impact on the makespan of co-scheduled applications when they are run on nodes with heterogeneous resources. In our experiments, we used VarySched, a resource scheduler that is specialized for such heterogeneous environments and that allows dynamic modifications of the scheduling strategy. We evaluated four different strategies using two applications and two resource governor settings. The results show that the application can be accelerated by a factor of 2.5 if the scheduler is chosen wisely.

Furthermore, the evaluation of the VarySched framework has shown that the throughput can be increased and the energy consumption reduced at the cost of a small overhead. The experiments were performed using real world applications. Thus, using VarySched the system resources can be exploited far more efficiently.

References

[1] The OpenACC Application Program Interface. Version 2.5, Oct. 2015. http://www.openacc-standard.org/.

[2] M. Amini, C. Ancourt, F. Coelho, B. Creusillet, S. Guelton, F. Irigoin, P. Jouvelot, R. Keryell, and P. Villalon. PIPS Is not (just) Polyhedral Software Adding GPU Code Generation in PIPS. In *First International Workshop on Polyhedral Compilation Techniques (IMPACT 2011) in conjonction with CGO 2011*, Chamonix, France, Apr. 2011. 6 pages.

[3] D. P. Anderson. BOINC: A System for Public-Resource Computing and Storage. In *Proceedings of the 5th IEEE/ACM International Workshop on Grid Computing*, GRID '04, pages 4–10, Washington, DC, USA, 2004. IEEE Computer Society.

[4] Apple. Grand Central Dispatch - A better way to do multicore. Technology Brief, 2009. http://opensource.mlba-team.de/xdispatch/GrandCentral_TB_brief_20090608.pdf.

[5] C. Augonnet, S. Thibault, R. Namyst, and P.-A. Wacrenier. StarPU: A Unified Platform for Task Scheduling on Heterogeneous Multicore Architectures. *Concurrency Computation: Practice & Experience - Euro-Par 2009*, 23:187–198, 2011.

[6] T. Beisel, T. Wiersema, C. Plessl, and A. Brinkmann. Cooperative Multitasking for Heterogeneous Accelerators in the Linux Completely Fair Scheduler. In *Proceedings of the International Conference on Application-Specific Systems, Architectures, and Processors*, pages 223–226, Piscataway, NJ, USA, Sept. 2011.

[7] U. Bondhugula, A. Hartono, J. Ramanujam, and P. Sadayappan. A Practical Automatic Polyhedral Parallelizer and Locality Optimizer. In *Proceedings of the ACM SIGPLAN 2008 Conference on Programming Language Design and Implementation*, pages 101–113, 2008.

[8] G. Bosilca, A. Bouteiller, A. Danalis, T. Herault, P. Lemarinier, and J. Dongarra. DAGuE: A Generic Distributed DAG Engine for High Performance Computing. *Parallel Computing*, 38(1-2, SI):37–51, 2012.

[9] C. Buchner. cudaminer 2013.11.20, April 2015. https://github.com/cbuchner1/ccminer.

[10] D. Feld, T. Soddemann, M. Jünger, and S. Mallach. Facilitate SIMD-Code-Generation in the Polyhedral Model by Hardware-aware Automatic Code-Trans- formation. In A. Größliger and L.-N. Pouchet, editors, *Proceedings of the 3rd International Workshop on Polyhedral Compilation Techniques*, pages 45–54, 2013.

[11] D. Feld, T. Soddemann, M. Jünger, and S. Mallach. Hardware-Aware Automatic Code-Transformation to Support Compilers in Exploiting the Multi-Level Parallel Potential of Modern CPUs. In *Proceedings of the 2015 International Workshop on Code Optimisation for Multi and Many Cores*, COSMIC '15, pages 2:1–2:10, 2015.

[12] R. Gad, T. Süß, and A. Brinkmann. Compiler Driven Automatic Kernel Context Migration for Heterogeneous Computing. In *Proceedings of the 34th International Conference on Distributed Computing Systems*, pages 389–398, 2014.

[13] J. Garzik. cpuminer 2.4.1, April 2015. https://github.com/pooler/cpuminer.

[14] T. Grosser, A. Cohen, J. Holewinski, P. Sadayappan, and S. Verdoolaege. Hybrid Hexagonal/Classical Tiling for GPUs. In *International Symposium on Code Generation and Optimization*, 2014.

[15] C. Juega, J. I. G. Pérez, C. Tenllado, and F. C. Catthoor. Adaptive Mapping and Parameter Selection Scheme to Improve Automatic Code Generation for GPUs. In *International Symposium on Code Generation and Optimization*, 2014.

[16] J. Kraus, M. Förster, T. Brandes, and T. Soddemann. Using LAMA for Efficient AMG on Hybrid Clusters. *Computer Science - R&D*, 28(2-3):211–220, 2013.

[17] K. Kumar Pusukuri, R. Gupta, and L. N. Bhuyan. ADAPT: A Framework for Coscheduling Multithreaded Programs. *ACM Transactions on Architecture and Code Optimization*, 9(4):45:1–45:24, 2013.

[18] J. Maassen, N. Drost, H. E. Bal, and F. J. Seinstra. Towards Jungle Computing with Ibis/Constellation. In *Proceedings of the 2011 Workshop on Dynamic Distributed Data-intensive Applications, Programming Abstractions, and Systems*, 3DAPAS '11, pages 7–18, New York, NY, USA, 2011. ACM.

[19] K. Menychtas, K. Shen, and M. L. Scott. Disengaged Scheduling for Fair, Protected Access to Fast Computational Accelerators. In *Proceedings of the 19th International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '14, pages 301–316, 2014.

[20] A. Munshi, B. Gaster, T. G. Mattson, J. Fung, and D. Ginsburg. *OpenCL Programming Guide*. Addison-Wesley Professional, 1st edition, 2011.

[21] P. Rogers. Heterogeneous System Architecture Overview. HOT CHIPS Tutorial - August 2013, 2013.

[22] Y. Saad. *Iterative Methods for Sparse Linear Systems*. Society for Industrial and Applied Mathematics, Philadelphia, PA, USA, 2nd edition, 2003.

[23] M. Schordan and D. J. Quinlan. A Source-to-Source Architecture for User-Defined Optimizations. In *Modular Programming Languages*, pages 214–223, 2003.

[24] SILKAN. HPC project. Par4All Automatic Parallelization. *http://www.par4all.org*.

[25] E. Sun, D. Schaa, R. Bagley, N. Rubin, and D. R. Kaeli. Enabling Task-Level Scheduling on Heterogeneous Platforms. In *GPGPU@ASPLOS*, pages 84–93, 2012.

[26] T. Süß, N. Döring, R. Gad, L. Nagel, A. Brinkmann, D. Feld, T. Soddemann, and S. Lankes. VarySched: A Framework for Variable Scheduling in Heterogeneous Environments. In *Proceedings of the IEEE CLUSTER 2016*, 2016.

[27] A. Venkat, M. Shantharam, M. Hall, and M. Strout. Non-affine Extensions to Polyhedral Code Generation. In *Int. Symposium on Code Generation and Optimization*, 2014.

[28] S. Verdoolaege, J. C. Juega, A. Cohen, J. I. Gómez, C. Tenllado, and F. Catthoor. Polyhedral Parallel Code Generation for CUDA. *ACM Transactions on Architecture and Code Optimization*, 9(4), 2013.

[29] J. R. Wernsing and G. Stitt. Elastic Computing: A Portable Optimization Framework for Hybrid Computers. *Parallel Computing*, 38(8, SI):438–464, 2012.