Device-Hopping: Transparent Mid-Kernel Runtime Switching for Heterogeneous Systems

Device-Hopping: Transparent Mid-Kernel Runtime Switching for Heterogeneous Systems

PAUL METZGER, University of Edinburgh, United Kingdom
VOLKER SEEKER, University of Edinburgh, United Kingdom
CHRISTIAN FENSCH, University of Edinburgh, United Kingdom
MURRAY COLE, University of Edinburgh, United Kingdom

Existing OS techniques for homogeneous many-core systems make it simple for single and multithreaded applications to migrate between cores. Heterogeneous systems do not benefit so fully from this flexibility, and applications that cannot migrate in mid-execution may lose potential performance. The situation is particularly challenging when a switch of language runtime would be desirable in conjunction with a migration. We present a case study in making heterogeneous CPU + GPU systems more flexible in this respect. Our technique for fine-grained application migration, allows switches between OpenMP, OpenCL, and CUDA execution, in conjunction with migrations from GPU to CPU, and CPU to GPU. To achieve this, we subdivide iteration spaces into slices, and consider migration on a slice-by-slice basis. We show that slice sizes can be learned offline by machine learning models. To further improve performance memory transfers are made migration-aware. The complexity of the migration capability is hidden from programmers behind a high-level programming model based on parallel_for. We present a detailed evaluation of our mid-kernel migration mechanism with the First-Come, First-Served scheduling policy. We compare our technique in a focused evaluation scenario against idealized kernel-by-kernel scheduling, which is typical for current systems, and makes perfect kernel to device scheduling decisions, but cannot migrate kernels mid-execution. Models show that up to 1.33x speedup can be achieved over these systems by adding fine-grained migration. Our experimental results on CPU + GPU systems with all nine applicable SHOC and Rodinia benchmarks achieve speedups of up to 1.30x (1.08x on average) over an implementation of a perfect but kernel-migration incapable scheduler when migrated to a faster device. Our mechanism and predicted slice sizes introduce an average slowdown of only 2.44% if kernels never migrate. Lastly, our parallel_for reduces the code size by at least 88% if compared to manual implementations of migratable kernels.

CCS Concepts: • **Software and its engineering → Runtime environments**; *Scheduling*; • **Computer systems organization → Heterogeneous (hybrid) systems**; • **Computing methodologies** → *Parallel programming languages.*

ACM Reference Format:
Paul Metzger, Volker Seeker, Christian Fensch, and Murray Cole. 2021. Device-Hopping: Transparent Mid-Kernel Runtime Switching for Heterogeneous Systems. *ACM Trans. Arch. Code Optim.* 18, 4, Article 57 (September 2021), 25 pages. https://doi.org/10.1145/3471909

1 INTRODUCTION

[0]New Paper, Not an Extension of a Conference Paper

Authors' addresses: Paul Metzger, Institute for Computing Systems Architecture, University of Edinburgh, 10 Crichton Street, Edinburgh, EH8 9AB, United Kingdom, paul.metzger@ed.ac.uk; Volker Seeker, Institute for Computing Systems Architecture, University of Edinburgh, 10 Crichton Street, Edinburgh, EH8 9AB, United Kingdom, volker.seeker@ed.ac.uk; Christian Fensch, Institute for Computing Systems Architecture, University of Edinburgh, 10 Crichton Street, Edinburgh, EH8 9AB, United Kingdom, c.fensch@ed.ac.uk; Murray Cole, Institute for Computing Systems Architecture, University of Edinburgh, 10 Crichton Street, Edinburgh, EH8 9AB, United Kingdom, mic@inf.ed.ac.uk.

The resources available to an application can vary unpredictably over its execution. For example, multiprogramming is used across the computing spectrum to improve utilization: high-end data center machines are multiprogrammed with demand-driven services and other jobs that reuse the machines' resources at times of low activity [32, 38, 50], while mixed criticality embedded systems mix timing critical workloads with less critical ones, again to improve utilization and energy consumption [4, 19, 51]. Energy and thermal constraints exacerbate the problem. An evolving workload mix can cause changes to the best allocation of the available power budget to silicon, or the appropriate exploitation of hardware characteristics, as in Big-Little systems [12]. In response, on CPU only systems, the OS is able to migrate multithreaded applications between cores. For example, dynamic scheduling strategies redeploy cores as they become available [16, 44, 47], and dynamically review the application to device mapping [15]. Power capped systems control the active core count and so their power consumption with thread migration [13].

It would be natural and desirable for a similarly flexible migration capability to be available on CPU + GPU platforms. However, migrating running applications between devices in such systems is challenging. Current runtime systems for GPUs make kernel-to-device scheduling decisions only at coarse-grained kernel launches and cannot perform *mid-kernel migration* [42, 45]. Because of this, perfect scheduling decisions on whether to launch on an earlier available slow device or wait for a fast device require unattainable knowledge of the future. If the best performance after migration also requires a change of language runtime (e.g. switching from OpenMP on the CPU to CUDA on the GPU, or vice-versa) the challenge is even greater.

In this paper we investigate a mechanism which enables these flexibilities for CPU + GPU systems, including the ability to switch the underlying language runtime. Specifically, we investigate a mechanism which allows schedulers to migrate applications in *mid-kernel execution* from CPU to GPU, and in the other direction, from GPU to CPU. In principle, to exploit mid-kernel migration, each application could be written with multiple embedded variants, data transfer code to switch between these, and heuristics to decide under which dynamic circumstances to do so. However, this would be very challenging to application developers, and in fact, could reduce maintainability by solidifying these decisions amongst true application level code. In this paper, we address this problem by drawing upon the concept of programming with parallel patterns [14, 20, 43]. Parallel patterns present APIs which abstract away implementation details, while simultaneously providing the system implementer with the contextual information to make good optimization and tuning decisions. Specifically, we show that mid-kernel migration can be hidden behind the parallel_for pattern, and that decisions on scheduling granularities, migration strategies, and device utilization can be handled efficiently and transparently, without burdening the application programmer. In more detail, our migration mechanism subdivides iteration spaces into slices as prior work has done [10, 11, 41, 53], and considers migration on a slice-by-slice basis. Slices provide stable states at which points the context of an application can switch devices and runtimes. To choose slice sizes we use off-line trained predictive models. Data transfers in systems with distinct per-device memory can have a significant cost, and so we also transfer data in a slicing aware way, to avoid unnecessary transfers if a kernel migrates. Our mechanism provides the key technological basis for transparent migration and runtime adaption of applications in CPU + GPU systems.

We evaluate mid-kernel migration with the First Come, First Served (FCFS) scheduling policy, using a simple scenario to allow us to focus on the cost and contribution of the migration mechanism. We show analytically that mid-kernel migration removes the need for unattainable knowledge of the future for scheduling decisions. In more detail, we show that FCFS with mid-kernel migration can achieve a theoretical maximum speedup of 1.33x over a perfect knowledge FCFS schedule without mid-kernel migration, and never performs worse than it. We confirm these results experimentally with nine benchmarks on a CPU + GPU system and show that mid-kernel migration with our

simple policy achieves speedups of up to 1.30x over kernel-by-kernel scheduled systems even if they benefit from a perfect schedule. We also demonstrate that if a kernel never migrates the overheads of slicing are negligible, and lastly, that our parallel_for reduces the code complexity significantly if compared to a manual implementation of the migration capability. In summary, this paper makes the following contributions:

(1) We describe a mechanism that enables mid-kernel execution migration between CPU and GPU in both directions, including the possibility of switching language runtimes dynamically. In contrast, current systems cannot migrate kernels between CPUs and GPUs once they have been launched [42, 45].
(2) We show that the complexity of the mechanism and its efficient implementation can be hidden *from programmers* behind a high-level programming model based on the widely known parallel_for construct.
(3) We show that slice size choices that introduce acceptable overheads can be learned by predictive models.
(4) We present an analytically derived maximum for the speedups with migration over current kernel-by-kernel scheduled systems in a simple deployment scenario.
(5) We provide a detailed evaluation that demonstrates, among other things, the general performance benefits of mid-kernel migration and exposes the performance behaviour of the mechanism in edge cases.

The remainder of this paper is structured as follows: Section 2 provides a motivating example. Section 3 introduces the migration mechanism and techniques for an efficient implementation. Section 4 discusses the predictive slice size models. Section 5 discusses a high-level programming model based on parallel_for that hides the complexity of the migration mechanism and the slice size predictors. Section 6 discusses the comparator for our analytical models and experimental evaluation, and the maximum theoretical speedup over it that can be achieved with mid-kernel migration for our deployment scenario. Section 7 presents experimental results. Section 8 discusses related work. Section 9 concludes and discusses future work.

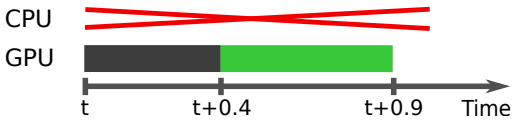
2 MOTIVATING EXAMPLE

This section illustrates the performance benefits of mid-kernel migration, and how a scheduler can take advantage of it in a simple scenario. We make mid-kernel migration and its implementation programmer transparent with our parallel_for, which is introduced in Section 5.

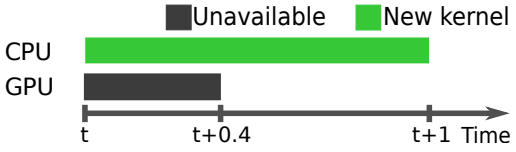
2.1 Better Performance with Mid-Kernel Migration

In current systems, kernels are only scheduled at launch time as discussed in Section 1 and illustrated with an example in Figure 1 [42, 45] (see Section 6.1 for a more detailed discussion of current systems). Opportunities for performance improvements are therefore missed if devices are temporary unavailable. On the one hand, kernels cannot make progress while they wait for their fast device[1]. On the other hand, if launched on an alternative slow device, kernels cannot migrate when faster devices become available. We fix this with mid-kernel migration. As shown in Figure 1c, kernels make progress on earlier available devices instead of waiting, and schedulers can migrate them to faster ones when they become available.

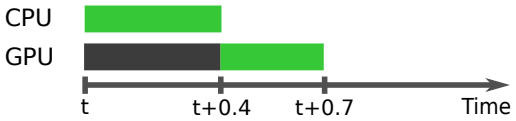
[1]In this paper we distinguish between the *fast* and *slow device* from the perspective of a kernel. In the absence of interference and contention kernels have a lower execution time on their *fast device* than on their *slow device*. Whether the CPU or the GPU is the slow device depends on the kernel and the target system.



(a) Execution without mid-kernel migration. The *new kernel* waits for the faster GPU and does not use the slower but immediately available CPU.



(b) Alternative execution strategy without mid-kernel migration. The *new kernel* starts immediately on the slower CPU but cannot use the faster GPU when it becomes available.



(c) Execution with mid-kernel migration. The *new kernel* starts on the CPU and finishes execution on the faster GPU.

Fig. 1. Illustration of the typical performance improvement opportunities missed without mid-kernel migration. (1a) and (1b) illustrate two possible execution strategies of a *new kernel* without migration, and (1c) its execution with migration. When the new kernel arrives at time t the GPU, on which it would execute twice as fast as on the CPU, is unavailable for 0.4 time units. Without mid-kernel migration the *new kernel* either waits for the GPU (1a) or starts immediately on the slower CPU (1b). In the former case, the new kernel finishes after 0.9 time units, after it waited for 0.4 time units for the GPU and then executed for 0.5 time units on it. In the latter case, the kernel launches immediately on the CPU and executes for one time unit. With migration the kernel executes for 0.4 time units on the CPU and migrates to the GPU when it becomes available to finish the remaining work in 0.3 time units (1c). With migration the *new kernel* finishes after 0.7 time units, and, therefore, is 1.29x faster than the schedule that results in the shortest combined waiting and execution time without migration, which is illustrated in (1a).

2.2 Simplified Scheduling Decisions

Without mid-kernel migration, perfect decisions on whether to launch kernels on their fast or slow device require generally unattainable knowledge of the future. To determine which kernel launch decision leads to the shortest combination of waiting and execution time, schedulers of such systems need to know when the fast device of a kernel will become available, and how long a new kernel would take to complete on each device. We simplify this decision with mid-kernel migration so that schedulers do not require such knowledge about the future. With mid-kernel migration schedulers can launch kernels on the earliest available device and migrate them if a faster device becomes available. This enables kernels which would not otherwise have been migrated to make progress on another device in the meantime. Similarly, kernels can utilize faster devices when they become available, which is advantageous when the best decision for current systems and schedulers is to execute on their slow device.

Section 6 builds idealized models to show that this policy never performs worse than the current kernel-by-kernel scheduled systems. However, in practice, management code, interference during the migration, and on-the-fly device setup cause slowdowns in some cases. We develop techniques, and strategies which address these problems in order to leave an overall performance win.

3 MID-KERNEL MIGRATION

This section presents a migration mechanism for systems with a CPU and a dedicated GPU, and techniques required for its efficient implementation. We implement the mechanism on top of

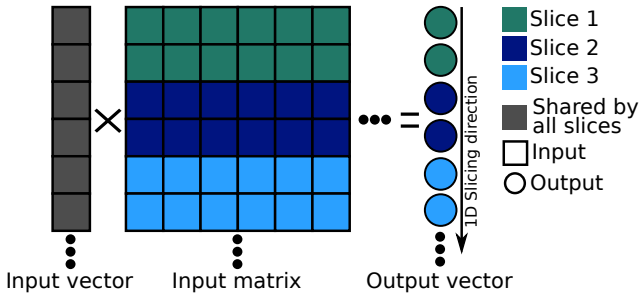


Fig. 2. Slicing and slicing aware data transfers with matrix-vector multiplication. The output vector is computed slice-by-slice. Only the matrix rows needed for a slice are transferred. The input vector is shared by all slices and so not affected by slicing. In this example the iteration space is one-dimensional, and its size is equal to the length of the output vector. Additionally, each output element could be computed in multiple slices with a 2D iteration space.

```
...
// Execute the kernel in slices
while not all slices have been executed:
  offset_into_iteration_space ← get iteration space offset
  slice_size                  ← get slice size
  target_device               ← get device
  // Launch slice kernel
  launch_slice_kernel(target_device, offset_into_iteration_space, slice_size)
...
```

(a) A pseudo code based illustration of slicing.

```
...
launch_application_kernel(target_device, iteration_space_size)
...
```

(b) The equivalent unsliced pseudo code.

Fig. 3. High-level illustration of how slicing is implemented internally with pseudo code. We do not require programmers to hand implement sliced kernels but automate this with our parallel_for instead (see Section 5). Implementation details are omitted for clarity. For example, we have omitted code for chunked data transfers, code for the abortion of slices (see Sections 3.2 and 3.3), and code for edge cases such as iteration space sizes that are not a multiple of the slice size. Additionally, our actual CUDA implementation executes data transfers and manipulates pointers into the in- and output buffers inside the loop before and after the kernel launch.

OpenCL, CUDA, and OpenMP. Specifically, in our implementation, kernels switch between CUDA on GPUs and OpenCL or OpenMP on CPUs. The mechanism is enabled through our parallel_for based programming model by which the programmer guarantees that iteration space points can be processed independent of each other (see Section 5). Moreover, the programming model makes the mechanism and its optimizations programmer transparent by virtue of its high-level nature.

3.1 Iteration Space Slicing, Runtime Switching, and Slicing Aware Data Transfers

To enable migration, kernel iteration spaces are processed in slices as illustrated in Figure 2 and migration is considered on a slice-by-slice basis. For clarity, in the remainder of the paper we

distinguish between *application level kernels* and *slice kernels*. The former are the simple intuitive CUDA and OpenCL kernels or OpenMP loop nests which would have been written in a conventional coding of the application and which are now generated implicitly by our system. With our mechanism, these are each sliced into a sequence of finer grained kernels, as illustrated in Figure 3, and for which we now reserve the term *slice kernel*. We slice in all dimensions of the iteration space with the same slice size. More complex policies can be developed. For example, slices in 2D iteration spaces are squares except for the slices at the end of rows or columns, which might be rectangular. Optionally, kernels can execute without slicing on their fast device if preemption on this device never occurs. In some cases, offsets need to be added to the thread and block IDs in the kernel code (not shown).

It might be necessary or desirable to change the underlying language runtime when an application kernel migrates. For example, CUDA kernels cannot execute on CPUs and so the runtime system must be switched when an application kernel migrates from a GPU where it used CUDA to a CPU. Between slices the state of a kernel, including which iteration space portion has already been processed, is stable and known. This way, after migration the application kernel execution can be resumed on the target device with a different implementation and runtime. This does not require any changes to the underlying CUDA, OpenCL, and OpenMP runtimes because slicing can be implemented as a layer on top of them. After migration, execution is simply resumed by calling the kernel launch functions and possibly data transfer calls of the new target device to execute the next slice kernel[2].

Data is transferred to dedicated GPUs in a slicing aware way to avoid unnecessary transfers which might negate the benefits of migration. Without this, the entire input data set needs to be transferred before the computation of the first slice kernel. If an application kernel migrates later on, some of the input data would have been transferred unnecessarily. Therefore, only the input data required for the current slice kernel must be transferred. In Figure 2 slices and the parts of the input matrix they require are color-coded. In some cases, such as sparse matrix multiplication, this is non-trivial because some of the data required by a slice kernel is dependent on other input data, but this is addressed in our implementation.

3.2 Migration Strategies

The migration strategy depends on the device in use, whether the kernel is idempotent[3], and the current execution step of the slice in execution. In addition, partial results computed on different devices must be merged. In some situations, slice kernels on the GPU are *aborted*. To abort a slice kernel, the execution jumps out of the current sequence of data transfers and kernel launches and restarts the slice kernel on the CPU.

Migration from GPU to CPU. In this migration scenario, the current slice kernel is in most cases aborted on the GPU and restarted on the CPU. GPU slice kernels are composed of multiple data transfers and a CUDA kernel launch, and can be aborted between these substeps, even though the individual steps cannot be aborted once launched. Therefore, if the application kernel migrates from a GPU to a CPU the current slice kernel is restarted on the CPU and aborted at the end of the current substep on the GPU. Non-idempotent slice kernels are not aborted on the GPU if any results of the slice kernel have been already transferred back from the GPU to the host. Unlike

[2]In the case of OpenMP a loop nest that implements the slice kernels is executed.

[3]Our parallel_for allows programmers to set whether the user code, that is passed to it, is idempotent [18]. The user code is idempotent if it can be re-executed multiple times for an iteration point and produces the same correct outputs with each re-execution. The results of the parallel_for are still correct if this optional tuning parameter is not set despite the user code being idempotent, but potential performance might be lost.

our implementation, migration could be implemented without aborts but would be less efficient because data transfers of the non-aborted slice kernel on the GPU would interfere with execution on the CPU.

Migration from CPU to GPU. In this scenario, either the *current* slice kernel on the CPU is restarted on the GPU or the *next* slice kernel is started on the GPU and the *current* one finishes on the CPU. On CPUs slice kernels correspond to either a single OpenCL kernel without data transfers, which cannot be safely aborted, or to an OpenMP loop nest, which will not be aborted in our implementation. We choose not to abort OpenMP loops in order to avoid the performance penalty that the corresponding code would introduce even if the abortion is not activated. Therefore, if the application kernel migrates from a CPU to a GPU and the kernel is idempotent the *current* slice on the CPU is started again on the GPU. In this case we do not wait for the old instance of the slice on the CPU to finish if the application kernel finishes earlier on the GPU. If the kernel is not idempotent the *next* slice kernel is started immediately on the GPU and executes in parallel with the *current* already launched slice, which runs on the CPU.

Merging results. Intermediate results successively computed on more than one device must be merged. We use two strategies depending on the access patterns in the output buffers. Both, access pattern and whether a buffer is an in- or output buffer are specified through our programming model (see Section 5 for a discussion of the access pattern attributes).

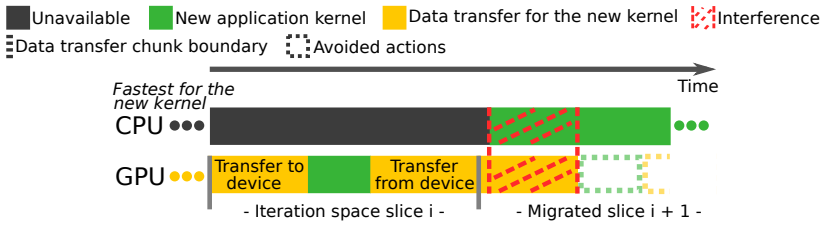
Merging Strategy 1: If the slice kernels write to distinct subsections of an output buffer then results computed on the GPU for such a buffer are transferred into their subsection in host main memory after each slice kernel.

Merging Strategy 2: If a buffer is not accessed in this way then incremental buffer updates in host main memory, as described above, are not possible. Instead, intermediate results computed on the CPU are transferred to the GPU before the first GPU slice kernel executes. With this strategy, GPU results are only transferred to host main memory when the application kernel migrates to the CPU or once the application kernel has completed on the GPU.

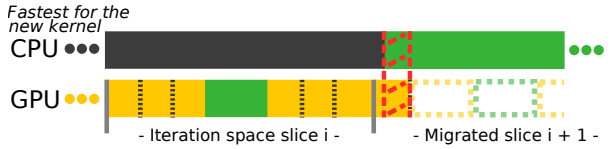
3.3 Interference Reduction and Earlier Aborts

Migration from GPU to CPU involves the abortion of the current slice kernel on the GPU (see Section 3.2). To be able to abort GPU slices earlier, data transfers are broken down into chunks as shown in Figure 4. Each data transfer chunk corresponds to a new internal API call and so transfers can be aborted between them. Application kernels cannot be aborted at arbitrary points because, as above, OpenCL and CUDA do not allow already issued data transfers and kernels to be aborted. Because of this, execution on the CPU and GPU are overlapped until the application kernel can be aborted, as indicated in Figure 4 by the red dashed lines. This overlap must be minimized for two reasons. Firstly, data transfers interfere with the execution on the CPU and therefore degrade performance. Secondly, reaching the next point at which the slice on the GPU can be aborted can take longer than the remaining application kernel execution time on the CPU.

We use a one-off brute force search to determine chunk sizes that introduce no more than an implementation-set maximum slowdown over execution without chunking. In a full deployment, this search would take place only once, transparently to programmers, "at the factory" or when the run-time system is installed. The search is not repeated for each application kernel instance because the chunking overheads are application kernel independent. This is the case because per buffer data transfer code is the same across kernels. In our implementation, instead of a single call to `cudaMemcpy`, `cudaMemcpy` is called once for each chunk in a loop, and the overheads are caused by the loop, additional function calls, and pointer arithmetic.

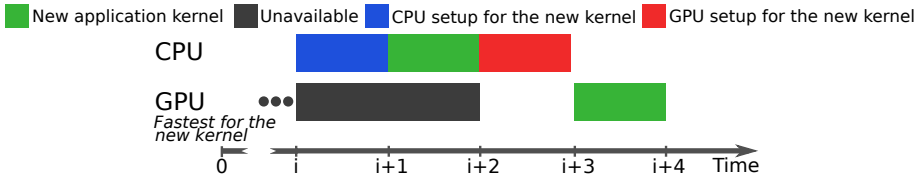


(a) Migration without data transfer chunking. The slice kernel on the GPU cannot be aborted until the data transfer is finished. The execution on the CPU is slowed down by interference caused by the data transfer.

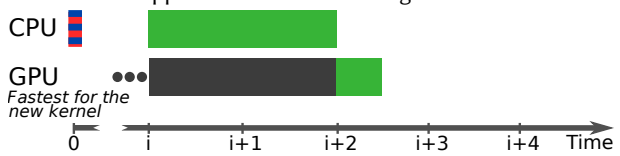


(b) Migration with data transfer chunking. The slice kernel on the GPU is aborted between data transfer chunks. This reduces the interference with the execution on the CPU.

Fig. 4. Migration without (4a) and with (4b) chunked data transfers.



(a) Execution of the new application kernel with migration but without the daemon.



(b) Execution of the new application kernel with migration and with the daemon.

Fig. 5. Execution without and with a daemon that sets up OpenCL and/or CUDA in advance. With the daemon the new application kernel does not spend time in setup code when it migrates. The daemon performs the setup once when it starts as indicated by the striped bar at time zero in (5b). Data transfers are not shown for simplicity.

3.4 Device Setup Cost Reduction

OpenCL and CUDA must setup devices before their use. Applications that execute without migration can hide this cost when an application kernel is waiting for a device other than the CPU. With migration this is not possible if waiting for the GPU, because the CPU, where the setup must take place, is already occupied by the new application kernel, as shown in Figure 5a. To fix this we introduce a daemon that performs the setup once when it starts, on behalf of any applications that run afterwards, as illustrated in Figure 5b. In this way, applications do not have to spend time in setup code during migrations. To implement this, applications are executed by the daemon to give them access to the preinitialized OpenCL and CUDA handles.

Table 1. Features used to predict slice sizes. "Device" indicates whether the features are used to predict slice sizes for the CPU or GPU. All static features are extracted at compile time and are adjusted at runtime. Dynamic features are extracted at runtime.

Feature	Device	Extraction Type
Bytes transferred to the GPU per CUDA thread	GPU	Dynamic
Comparison operations	CPU	Static (adjusted with loop bounds)
Floating point and integer compute operations	CPU & GPU	Static (adjusted with loop bounds)
Memory accesses	CPU	Static (adjusted with loop bounds)

4 CHOOSING SLICE SIZES

This section discusses our machine learning models that predict kernel instance-specific slice sizes. Since slicing is hidden entirely behind our parallel_for, slice size choices are also handled transparently to the programmer. As discussed in Section 3.1 if the slow device of an application kernel is available earlier than its fast device, then the kernel executes on its slow device in slices to allow for mid-kernel migration once the fast device becomes available. The models presented in this paper are a function $f(v) = s$, that based on a vector of application kernel features v predicts a slice size s. We build separate models for the CPU and GPU because of their strong microarchitectural differences. We also exploit prior work on slicing in the context of caching (see Section 4.5) [10, 35].

4.1 Target Slice Sizes

The target slice sizes are a compromise between slicing overheads and resource wastage. On the one hand, *the larger* the slice sizes the smaller the overheads because fewer slices are executed, which in turn means code that implements slicing and introduces overheads is executed less often (see Section 3.1). These overheads cannot be amortized if a kernel never migrates, and lead to slowdowns in these situations. On the other hand, *the smaller* the slice sizes, the quicker an application kernel migrates once a faster device becomes available because migration is considered more frequently. Additionally, less interference is caused on the fast device by the residual execution on the slow device after migration (see Section 3.2 and Section 3.3). This is so because the remaining slice on the slow device finishes earlier with smaller slice sizes, or because a point at which the slice can be aborted is reached earlier, because the data transfers between two such points are smaller. Similarly, the smaller the slice sizes, the less work is thrown away when a slice is aborted. Therefore, the target slice sizes are the smallest slice sizes that introduce an acceptable slowdown if an application kernel never migrates.

4.2 Application Kernel Features for the Slice Size Predictors

Table 1 lists the features used by the slice size predictors. To extract static source code features, we build a feature extractor with Clang that traverses the abstract syntax tree of the kernel code. Static features are adjusted at run-time once loop bounds are known by multiplying counts for operations inside the loops with the iteration counts of the corresponding loops. Loop bounds are determined based on the parameter values that are passed to the kernel if the parameter values can be directly inserted into the bounds, or the kernel source code if the bounds are hard-coded. As heuristics, terms in the loop condition that cannot be evaluated before the kernel execution are ignored and loops whose bounds cannot be determined are set to an iteration count of one. Operations in if-branches without a corresponding else branch in loops are not counted as they are typically not taken, like for example a branch that is only taken when a sought value has been found. To generate training data, we determine loop bounds manually if they cannot be determined

automatically. The only dynamic feature is *"Bytes transferred to the GPU per thread"*. This feature is computed based on the amount of data to be transferred to the GPU and the number of CUDA threads with which an application kernel implemented in CUDA would be launched.

The chosen features are indicative of the work required for each iteration space point. This enables the models to predict slice sizes that introduce acceptably low overheads if an application kernel never migrates. This is so because the target slice size for a set overhead and the average execution time per iteration space point correlate. The less work is required per iteration space point, the larger the slice sizes that introduce only a set overhead. The reason is that the per slice execution time of the slicing code that causes the overheads is independent of the slice size (see Section 3.1). For example, for an average execution time per iteration space point of 1ms and 1ms per slice for the slicing code, the smallest slice size that introduces a slowdown of no more than 1.01x is 100.

4.3 Training the Slice Size Predictors

We use linear regression for the CPU model, and a random forest regressor with 50 decision trees with a depth of two for the GPU model [6]. All hyperparameters except the tree depth and the tree count are the defaults of the popular Scikit-Learn[4] machine learning library. We reduce the tree count to reduce the runtime costs of predictions with the random forest. For slice sizes on the CPU we use linear regression because of its low runtime overheads and the strong linear correlations between CPU target slice sizes and input features. For GPU slice sizes we use random forests because they performed best of all models explored.

To train the models we extract features from the kernels in the training set, as described in Section 4.2, and determine target slice sizes through brute force search. The training and the required brute force search need to be done only once *"at the factory"* before the system is deployed, as in previous work [26, 52]. If the workload type in a real deployment changes the models can be retrained with applications representative of the new workload and updated. For practical reasons we do not test each point in the slice size parameter space of each kernel, but step through the parameter space with a step size that we set by hand for each kernel instance. During the brute force search, we increase the slice size until the overheads are between 2% and 4% (see Section 4.1). We use a maximum allowed overhead of 2% for slicing on the GPU except for a small number of benchmark instances for which slice sizes with only 2% overhead cannot be found. We increase the maximum allowed overhead to up to 2.75% in these cases. For the same reason we always use a maximum of 4% for slicing on the CPU. We take the logs of the features for both models and the log of the training slice sizes for the CPU model to strengthen the linear correlations between features and target slice sizes. Otherwise, the correlations are weakened by heavy tails. Finally, for the CPU model we standardize the features and slice sizes to place their means at zero and normalize them to their standard deviation.

4.4 Deploying the Slice Size Predictors

Figure 6 illustrates when features are extracted, and predictions are made. Static features are extracted at compile time (1). At run-time (2) the size of the buffers allocated on the GPU, the iteration space size, and application kernel parameters that determine loop bounds are recorded. Next, the loop bounds are used to adjust the static features (3) as described in Section 4.2. Finally, the predictive models predict a slice size (4). Because we use the log of the slice sizes and standardize the slice sizes afterwards to train the CPU model (see Section 4.3), we apply the inverse of both to the raw predictions to compute the final CPU slice sizes.

[4]https://scikit-learn.org (version 0.23.2),

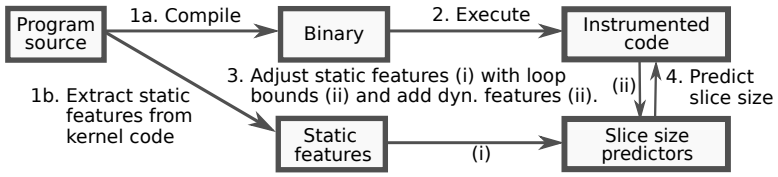


Fig. 6. Overview over the steps required to predict slice sizes (see Section 4.4 for a detailed discussion). The working steps are performed in order of the numbering.

```
1  ...
2  // Create parallel_for instance. Code similar to this is required by existing parallel_for implementations.
3  parallel_for pf(0, output_vector_size, [=]DEVICE_HOPPER_FUNCTION_PARAMETER() {
4   int iteration = get_iteration();
5   int result = 0;
6   for (unsigned int col = 0; col < input_vector_size; ++col)
7    result += in_vector[col] * in_matrix[iteration * input_vector_size + col];
8   out_vector[iteration] = result;
9  });
10 // Specify memory accesses. Our programming model requires these in addition to the function parameter.
11 int results_per_batch       = pf.batch_size;
12 int matrix_inputs_per_batch = pf.batch_size * MATRIX_ROW_SIZE;
13 pf.add_buffer_access_patterns(
14  buf(in_vector,  direction::in,  pattern::all_or_any),
15  buf(in_matrix,  direction::in,  pattern::successive_subsections(matrix_inputs_per_batch)),
16  buf(out_vector, direction::out, pattern::successive_subsections(results_per_batch)));
17 // Set optional tuning parameters and run.
18 pf.opt_set_simple_indices(true).opt_set_is_idempotent(true).run();
19 ...
```

Fig. 7. Illustration of the parallel_for based programming model with a simple implementation of matrix vector multiplication (see Section 5 for a detailed explanation). Existing parallel_for implementations require code similar to lines three to nine. The only additional code that our model requires are access pattern attributes for each in- and output buffer in lines 11 to 16. The method calls in 18 are optional tuning parameters except for run(), which executes the parallel_for.

4.5 Choosing Slice Sizes for Sparse Matrix Vector Multiplication

The slice sizes for Sparse Matrix Vector Multiplication (SPMV) are handled differently, because SPMV is usually implemented as a specialized library, and so we assume that the run-time system knows when SPMV is executed. We use this knowledge to repurpose previous work that uses slicing without migration [10, 35]. This work exploits caching effects in SPMV of which our predictors are not aware with smaller and so for our purposes better slice sizes (see Section 4.1). Chen et al. report that SPMV benefits from these caching effects [10]. With this knowledge we use a static analysis based slice size heuristic that is heavily inspired by the heuristic presented by Kim et al. [35]. In contrast to Kim et al., we consider the L2 cache instead of the L1 because on our GPU regular memory accesses do not go through the L1 cache. Besides that, we do not change the block size but use the original block size of the benchmark.

5 OUR HIGH-LEVEL PROGRAMMING MODEL

The complexity of mid-kernel migration and the slice size prediction is hidden behind a high-level programming model based on parallel_for in the style of OpenMP, Kokkos, Raja, and SYCL [3, 5, 23, 27]. In comparison to existing parallel_for implementations, we only require programmers to provide additional memory access pattern attributes for each buffer. In return we hide the complexity of an efficient implementation of slicing and the slice size prediction. In more detail, these attributes are required to generate code for slicing aware data transfers (see Section 3.1) and to merge partial results (see Section 3.2). Our parallel_for executes in parallel multiple instances of a function parameter that each correspond to an iteration space point. Programmers must ensure that the function parameter does not assume any order in which the iteration space points are executed. Our API is implemented with a library and C++ source-to-source translator that generates CUDA, OpenCL, and OpenMP kernels, as well as code that implements slicing and chunked data transfers (see Section 3).

Figure 7 uses an example user implementation of matrix vector multiplication to demonstrate the parallel_for. The first two arguments of the parallel_for constructor are the start and end of the iteration space, and the third is the function parameter (lines 3 to 9). Iteration points are grouped into successive *batches*. The current iteration and the batch size can be retrieved (line 4, and lines 11 and 12) with respective functions. All buffers that are used by the function parameter are registered with attributes that describe the access direction and access pattern (lines 14 to 16). The access pattern attribute is `all_or_any` if each batch accesses either all elements of a buffer or the access pattern does not fit the attributes discussed below (line 14). The attribute is `successive_subsections` (lines 15 and 16) if successive batches access only successive contiguous subsections of a buffer. With this attribute programmers can simply specify how many buffer elements each batch accesses. Optional tuning parameters indicate how indices are used (first function call in line 18), or if the function parameter is idempotent, which informs the migration strategy (see Section 3.2). The `opt_set_simple_indices(true)` function call indicates that the `get_iteration()` and `get_batch_iteration()` indices are only used for memory accesses. `get_batch_iteration()` returns the current batch ID. As an optimisation buffers allocated on the GPU are only large enough to contain data for a single slice if this tuning parameter is set, and if the buffers have the access pattern attributes `successive_subsections` or `continuous_subsections` (see below for an explanation of the latter).

We offer further API calls for more complex applications. For example, for applications with indirect memory access patterns or applications in which batches access overlapping buffer subsections a `continuous_subsections` attribute can be parametrized with two function parameters that compute the start and end indices of the subsections based on the batch IDs. These function parameters can access other buffers for indirect memory accesses. To optimize execution on the GPU, address space qualifiers can be added to variables and buffers. Finally, the parallel_for can be specialized to a reduction with a method call.

One typical use-mode for our API is to code kernels from scratch. However, pre-existing OpenCL and CUDA kernels can be ported to it in a simple process. Original kernel code can be used as the function parameter for the parallel_for with minor modifications, like replacing CUDA barrier operations with our barrier function. Code for manual management of device buffers and data transfers is replaced with the memory access attributes for each buffer.

6 AN IDEALIZED PERFORMANCE MODEL

This section shows analytically that mid-kernel migration outperforms kernel-by-kernel scheduling, which is typical for current systems. For this, we create idealized models of both and derive a maximum of 1.33x for the speedups that can be achieved by adding migration in our deployment scenario (see below), irrespective of the kernels and devices involved. Finally, insights into how

these speedups change with system characteristics and migration time points are provided. In Section 7.3, application specific speedups based on this model serve as essentially unattainable idealized upper bounds, allowing us to evaluate the quality of our practical slicing implementation and its overheads. Our modelled scenario is composed of a new application kernel, and its *fast* and *slow* devices (see Section 2.1 for an explanation of the terms *fast* and *slow device*). The new kernel arrives while its fast device is temporary unavailable. This simplified scenario allows us to focus on the evaluation of the migration mechanism. The execution times on the fast and slow devices, and the waiting time for the fast device are normalized to the execution time on the fast device.

6.1 Our Baseline Comparator System

In this section and our experimental evaluation in Section 7 mid-kernel migration is compared against the best possible implementation of the non-migrating kernel-by-kernel scheduling, which is typical of current systems [42, 45]. These systems require unattainable knowledge for perfect scheduling decisions (see Section 2 for an example). When a kernel arrives, and its *fast device* is occupied by another kernel or is otherwise not available, current systems have two options: (1) wait for an unpredictable amount of time for its *fast device* or (2) launch earlier on an alternative but *slower device* without being able to migrate when a better one becomes available. The wrong decision can lead to serious slowdowns. A perfect scheduler for such systems would need to know how long the new kernel will run on its slow and fast devices, and when in the future its fast device will become available, information which is not always known.

As a comparator, we define a theoretical perfect scheduler which has this practically unattainable knowledge and, therefore, call it the *Perfect Non-Migrating Scheduler* (PNS). However, the PNS is incapable of mid-kernel migration and, therefore, limited to kernel-by-kernel scheduling decision. Because the PNS has knowledge about the future, actual systems presented in previous work can only be approximations of it [1, 42, 45]. Therefore, it is a harder reference point than any of these systems.

In comparison to the PNS our scheduler does not rely on unattainable knowledge. Kernels are simply started on their slow device if the fast device is not available and are migrated to the fast device as soon as it becomes available (see Section 2.2).

6.2 The Scheduler

We schedule kernels with the First Come, First Served (FCFS) policy, as in previous work [42, 45]. In more detail, in this section and the evaluation we compare *FCFS with mid-kernel migration* with *FCFS without mid-kernel migration* but with perfect knowledge about the future, the latter is implemented by the PNS.

6.3 Components of the Model

We will model idealized implementations of the PNS and a system with migration in order to derive the maximum speedup of the latter over the former. The models are idealized because they assume the absence of these practical issues: interference during migration, device setup costs, slicing overheads, and the fact that kernels cannot be migrated instantaneously (see Section 3 and Section 4.1 for a discussion of all of these). The models use the following components:

- k denotes the ratio of how much faster the new application kernel executes on its fast device than on its slow device.
- The normalized execution time on a kernel's slow device is also k (now as a number of time units) because it is, as above, the execution time on the slow device normalized to the execution time on the fast device. Because of this equality we use k for both in the rest of the

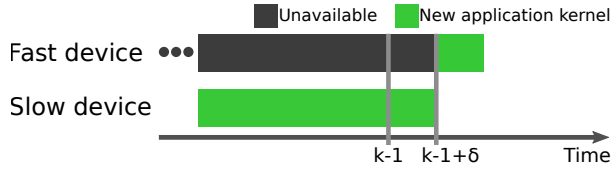


Fig. 8. Execution with migration at time $k-1+\delta$. In contrast, the execution with the PNS is as shown in Figure 1a for negative δ and as in Figure 1b for positive delta.

paper. In fact, the formula to compute the theoretical maximum speedup of a kernel through migration exploits this equality.

- Time $k-1$ marks a crucial transition point for the PNS. Recall that (because of normalization) the new application will execute in *one* time unit on its fast device. Therefore, when the new kernel arrives, if PNS knows that the fast device will become available before time $k-1$, it is preferable to wait and execute it there. This will cause the new kernel to finish earlier than executing it immediately on the slow device. In contrast, if PNS knows that the fast device will only become available after time $k-1$, then it is preferable to execute it immediately on the slow device. At $k-1$ either decision results in the same execution time.
- δ is the difference between $k-1$ and the point in time at which the fast device becomes available.

Additionally, the components C_{fast}, C_{slow}, C_{PNS}, and C_{mig} denote the execution time of the new kernel on its fast device, its slow device, with the PNS, and with our migration mechanism respectively, and are used to derive the models. W_{fast} (waiting time) is the time starting from the arrival of the new kernel after which its fast device becomes available and is also referred to as waiting time for the fast device.

6.4 Speedup with Migration over the Perfect Non-Migrating Scheduler

The maximum speedups for the two choices available to the PNS are modelled separately. Our first model is a specialization of Amdahl's Law [30] and our second model is related to it. Figures 1a and 1b depict the choices of the PNS and Figure 8 execution with migration.

PNS Choice 1) Immediately launch the new application kernel on its slow device. We distinguish two cases for the value of δ.

a) $\delta < 0$: This case is not possible. The choice made by the Perfect Scheduler implies that the execution time of the new application kernel on its slow device C_{slow} is lower than the combined waiting and execution time with its fast device:

$$C_{slow} < W_{fast} + C_{fast} \tag{1}$$

However, if δ is negative then scheduling the application kernel on its fast device would result in a shorter combined execution and waiting time which stands in contradiction to the decision made by the PNS:

$$C_{slow} > W_{fast} + C_{fast} \tag{2}$$

$$\Leftrightarrow \quad k > k - 1 + \delta + 1 \qquad , k > 1 \text{ and } \delta < 0 \tag{3}$$

$$\Leftrightarrow \quad k > k + \delta \tag{4}$$

b) $\delta \geq 0$: In this case the execution time C_{PNS} with the PNS and C_{mig} with migration are

$$C_{PNS} = k \qquad , k > 1 \tag{5}$$

$$C_{mig} = k - 1 + \delta + \frac{1-\delta}{k} \qquad , k > 1 \text{ and } \delta \geq 0. \tag{6}$$

The speedup $S_1(k, \delta)$ of migration over the PNS derived from this is

$$S_1(k, \delta) = \frac{C_{PNS}}{C_{mig}} \tag{7}$$

$$\Leftrightarrow \quad S_1(k, \delta) = \frac{k}{k - 1 + \delta + \frac{1-\delta}{k}}. \tag{8}$$

PNS Choice 2) Wait for the fast device. Again, we distinguish two cases for the value of δ.

a) $\delta > 0$: This case is not possible. The choice made by the PNS implies

$$W_{fast} + C_{fast} < C_{slow}. \tag{9}$$

However, if δ is positive then scheduling the new application kernel on its slow device would have resulted in a shorter execution time which stands in contradiction to the decision made by the PNS:

$$C_{slow} < W_{fast} + C_{fast} \tag{10}$$

$$\Leftrightarrow \quad k < k - 1 + \delta + 1 \qquad , k > 1 \text{ and } \delta > 0 \tag{11}$$

$$\Leftrightarrow \quad k < k + \delta \tag{12}$$

b) $\delta \leq 0$: The execution times with the PNS and migration are

$$C_{PNS} = k + \delta \qquad , k > 1 \text{ and } \delta \leq 0 \tag{13}$$

$$C_{mig} = k - 1 + \delta + \frac{1-\delta}{k} \qquad , k > 1 \text{ and } \delta \leq 0. \tag{14}$$

The speedup $S_2(k, \delta)$ derived from this is

$$S_2(k, \delta) = \frac{k + \delta}{k - 1 + \delta + \frac{1-\delta}{k}}. \tag{15}$$

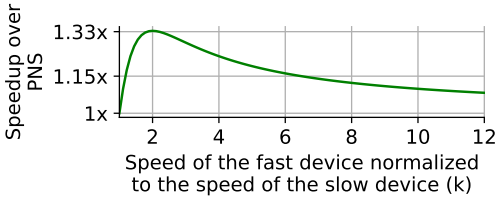
6.5 Application Kernel and Device Independent Maximum Speedup

The maximum speedup over the PNS is 1.33x. Both speedups derived in the previous subsection are maximal at $k = 2$ and $\delta = 0$.

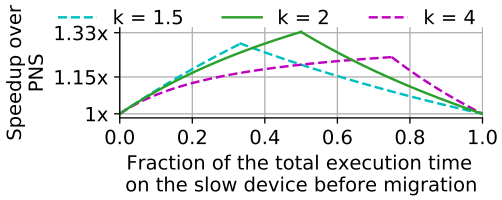
$$S_1(2, 0) = S_2(2, 0) = 1^1/_3 \tag{16}$$

6.6 Speedups with Different k

The maximum speedup for a particular application kernel depends on the performance difference between both devices, which is k. Figure 9a shows the maximum speedup over the PNS for different k. The speedup decreases for $k > 2$ because the larger the performance difference between the devices, the less progress can be made on an alternative slow device before the migration. For $k < 2$ the speedup decreases because the closer the performance of both devices, the less advantage can be gained through migration compared to full execution on the slow device.



(a) Maximum speedup with migration over the PNS for different speedups of the application kernel on its fast device over execution on its slow device. This is a limit study of the speedup and so the best migration time point is used, which means δ is set to 0.



(b) Speedups for different migration time points, i.e. for different δ. See Section 6.3 for an explanation of k.

Fig. 9. Speedups over the PNS with different relative device speeds (a) and migration time points (b).

Table 2. Details of the evaluation machine. The GPU uses the Kepler microarchitecture. DVFS and Turbo-Boost are deactivated.

CPU Model	Cores	Sockets	Hyperthreading	GPU Model
Intel Core i7-4770	4	1	Off	nVidia GTX Titan

6.7 Speedups with Different δ

The amount of work done on the slow device before the migration determines the speedup over the PNS. For example, if an application kernel migrates right after it launched or with virtually no work left no significant benefit can be gained with migration over just waiting for its fast device or finishing the execution on its slow device. Figure 9b shows speedups over the PNS for different migration time points expressed as the fraction of the total execution time on the slow device. For k not equal to 2 the maximum is lower. The maximum moves to the right for $k > 2$ and to the left for $k < 2$.

7 EVALUATION

7.1 Experimental Setup

Table 2 lists details of the evaluation platform. We use the Intel C++ Compiler (ICC) 19.0.5.281 with -O3, NVCC 10.2, version 455.23.05 of the nVidia GPU driver, Linux Kernel 5.3.18, and version 18.1 of the Intel CPU OpenCL runtime and compiler. Thirty samples are taken for each data point and the mean speedups are reported if not stated otherwise. Error bars show the standard error of the mean and are in some cases barely visible.

7.2 Experimental Method

The goal of the experiments is to measure speedups obtained through mid-kernel migration over an ideal implementation of current kernel-by-kernel scheduled systems (see Section 6.1). The kernel-by-kernel approach assumes perfect scheduling decisions but cannot migrate application kernels once they are launched, in other words it is an implementation of the *Perfect Non-Migrating Scheduler* (PNS) introduced in Section 6.1. For the evaluation we use the scenario described in

Table 3. Application kernel instances. The fast devices of the kernels, and the relative speeds of the fast and slow devices, which are indicated by k are determined experimentally (see Section 6.3, and Section 7.2 for an explanation of k). Slice counts are computed based on the predicted slice sizes.

Suite	Bechmark	Input size	Fast dev.	CPU impl.	Slices	k
Rodinia	B+Tree Find K	Small	GPU	OpenCL	507	2.74
		Medium	GPU	OpenCL	1014	3.05
		Large	GPU	OpenCL	2264	2.95
	B+Tree Find Range	Small	GPU	OpenCL	949	2.45
		Medium	GPU	OpenCL	1756	2.84
		Large	GPU	OpenCL	3161	2.54
	Nearest Neighbor	Small	*CPU*	OpenCL	53	2.35
		Medium	*CPU*	OpenCL	103	2.34
		Large	*CPU*	OpenCL	230	2.37
SHOC	MD5Hash	Small	GPU	OpenCL	53	12.31
		Medium	GPU	OpenCL	115	12.94
		Large	GPU	OpenCL	245	13.02
	FFT	Small	*CPU*	OpenCL	36	1.32
		Medium	*CPU*	OpenCL	72	1.32
		Large	*CPU*	OpenCL	108	1.31
	GEMM	Small	GPU	OpenCL	64	7.19
		Medium	GPU	OpenCL	121	5.54
		Large	GPU	OpenCL	144	8.44
	Inverse FFT	Small	*CPU*	OpenCL	36	1.32
		Medium	*CPU*	OpenCL	72	1.31
		Large	*CPU*	OpenCL	108	1.31
	Reduction	Small	*CPU*	*OpenMP*	3	3.19
		Medium	*CPU*	*OpenMP*	6	3.20
		Large	*CPU*	*OpenMP*	12	3.21
	SPMV	Small	*CPU*	OpenCL	15	3.47
		Medium	*CPU*	OpenCL	22	3.36
		Large	*CPU*	OpenCL	30	3.23

Section 6, in which the new kernel's fast device[5] is initially unavailable. Speedups for different migration time points are measured because speedups change in response to how long the fast device is unavailable, as discussed in Section 6.7. Additionally, the geometric mean of the per migration point speedups are reported to determine if migration benefits overall performance. For a fair comparison, all sample points are equally spread out. The runtime costs of the slice size models are included in the measurements.

Results are reported with nine benchmarks from the SHOC and Rodinia benchmark suites (see Table 3) [7, 17]. We use all benchmarks of SHOC and Rodinia that consist of a single kernel that is not invoked repeatedly and are therefore relevant to mid-kernel migration. Current runtime systems are not applicable to applications that consist of such single kernels because they make migration decisions only on a kernel-by-kernel basis as discussed in Section 6.1. For measurements

[5]We determine device affinities experimentally. In a real deployment this would be replaced with the predictive models of prior work [26, 49] or the programmer would set the fast device as is the case with OpenCL and implicitly with CUDA.

with migration capable implementations all benchmarks are implemented with our parallel_for (see Section 5). To demonstrate the applicability of our API to kernels with indirect memory accesses we also implement SPMV with parallel_for in our experiments. However, as explained in Section 4.5 we assume that our runtime system knows when it executes SPMV and can use this knowledge to choose better slice sizes as this is what would happen in a realistic library deployment. We measure execution time spent in a region of interest that includes all code sections of the benchmark (kernel execution, data transfers, migration management, etc.) except the generation or reading in of input data. As shown in Table 3 both CPU and GPU are about equally often represented as the fast device, and the benchmarks cover a wide range of values for the performance difference k between the devices, which is introduced in Section 6.3.

We use CUDA on the GPU, and OpenCL for all CPU implementations except reduction for which we use OpenMP. We experimentally determined that CUDA performs better or equally well than OpenCL on the GPU, and that OpenCL outperforms OpenMP on the CPU in the majority of cases. OpenMP improves performance only marginally otherwise, except for reduction for which OpenMP performs significantly better. Reductions are a distinct computational pattern and so the best implementation can be identified before the system is deployed with one-off costs. Additionally, our runtime system knows when reductions are executed through our programming model (see Section 5). Because SHOC uses only CUDA and OpenCL, we ported the "reduction" benchmark to OpenMP with the `reduction` clause.

We do not use iteration space slicing on the fast device as discussed in Section 3.1 because in our evaluation scenario kernels never migrate from their fast to their slow device. We determine an application independent data transfer chunk size that introduces a maximum slowdown of 0.5% for parallel_for and 1.5% for reductions as discussed in Section 3.3. For practical reasons, we double the tested chunk size with each search step starting with a chunk size of 1MB. The chunk size is 64MB for standard parallel_for and 16MB if a parallel_for is specialised to a reduction (see Section 5).

We replace the outdated standard problem sizes of the benchmarks with three larger ones that roughly require these execution times on the slow device: 200-250ms (small), 400-500ms (medium), and 800-1000ms (large). The execution times on the fast devices range from 17ms to 784ms.

To train the slice size predictors we use leave-one-out cross-validation, which means we train the predictors separately for each benchmark, using only the other benchmark kernels as the training set. This way the predictors choose a slice size for a kernel they have not seen before.

Measurements with the PNS implementation, with which we compare migration, do not include CUDA and OpenCL setup times. The migration-capable implementations benefit from setup taking place ahead of time in the daemon. To focus on migration, we factor this out by giving the PNS the same benefit.

It is known that deliberate slicing can improve performance even if an application kernel never migrates, for orthogonal reasons such as better use of the caches [10, 34]. To avoid unfairly disadvantaging the PNS implementation through these effects we allow it to use slicing, but without migration, if this improves its performance. We determine experimentally for each benchmark instance if this is the case.

7.3 Speedups Over the Perfect Non-Migrating Scheduler

Figure 10 shows speedups with migration over the implementation of the PNS (see Section 6.1 and Section 7.2). As discussed in Section 6.1, the PNS includes a perfect scheduler and is, therefore, at least as good as any possible implementation of current mid-kernel migration incapable systems. Mid-kernel migration outperforms the PNS in all cases, as shown by the geometric means of the speedups. The maximum and average speedups are 1.30x and 1.08x. Figure 10 supports contribution (5) of Section 1 in two ways. Firstly, it shows that mid-kernel migration outperforms current systems,

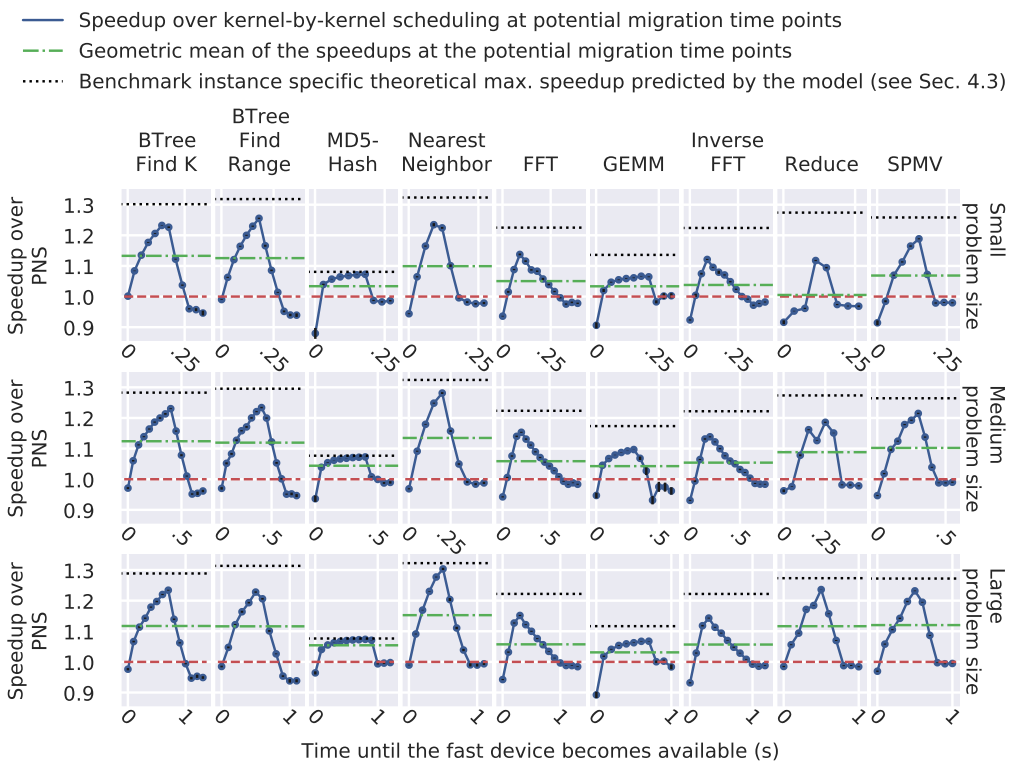


Fig. 10. Speedups with mid-kernel migration (blue dots) and their geometric means (green lines). We present speedups over current systems, which are represented by the PNS. As discussed in Section 6.1, the PNS is better than any possible real implementation. Mid-kernel migration outperforms current systems in all cases as indicated by the geo. means. Insights into the performance behaviour at different migration time points are also provided. Each value on the x-axes corresponds to a distinct potential migration time point. The green lines are the geo. means of the per migration point speedups, which are the blue dots. The red lines indicate the break-even speedup of 1x. The theoretical maximum (black dotted lines) is benchmark instance specific because it depends on k, which is the speedup of a benchmark on its fast over execution on its slow device as discussed in Section 6. At the arrival of a new benchmark its fast device remains unavailable for the time on the x-axis, so the benchmark executes for this time on its slow device before migrating. Migration causes slowdowns when a kernel migrates soon after its start, as shown on the left-hand sides of the subgraphs but speeds up the cases in the middle sections. Migration benefits performance of all kernels because the speedups outweigh the slowdowns as indicated by the geo. means of the speedups above 1x.

which are represented by the PNS, for all benchmark instances as indicated by the geometric mean speedups. Secondly, the figure provides detailed insights into the performance behaviour of mid-kernel migration at different migration time points during the execution of the kernels.

The remainder of this subsection first explains Figure 10 by describing the structure of one of its subgraphs, and then the experimental results in general. The subgraph in the top left corner shows speedups with mid-kernel migration over the PNS with the *B+Tree Find K* benchmark and its small problem size. The blue dots show speedups at distinct potential migration time points. For example, at the third blue dot the fast device of the kernel is unavailable for 60ms starting from

the arrival of the benchmark. In this case, the PNS decides to wait for the fast device. In contrast, with mid-kernel migration the benchmark makes progress on its slow device for the first 60ms instead of waiting, and then migrates to its fast device. In this case, mid-kernel migration is 1.14x faster than the PNS. The speedup closest to the theoretical maximum (indicated by the black line), occurs with a migration at time 150ms. This maximum is instance specific because it depends on k, which is the kernel specific speedup on its fast over its slow device. For this benchmark k is 2.74 as detailed in Table 3. The green line shows the geometric mean of the speedups at the potential migration time points (blue dots). Migration benefits overall performance because the geometric mean is above 1x, indicated by the red line. The points on the x-axis are *potential* migration points because migration does not happen at all of them as discussed below.

At the left-hand end of each subgraph, slowdowns are visible for small waiting times for the fast device. In these corner cases the new application kernel has spent little or no time on its slow device before it migrates and so the PNS chooses to wait for the fast device. Thus, little or no advantage can be gained from the ability to use the slow device before the fast one is available. In the worst case, interference during the migration caused by the slice kernel, which has started on the slow device, negates all benefits of mid-kernel migration and causes a slowdown. MD5Hash, and GEMM experience the worst slowdown in this area. However, they also have the shortest execution times of all benchmarks on their fast devices.

As expected, migration outperforms the PNS in the middle section of the subgraphs because application kernels can use the slow device first and then the fast device as discussed in Section 3. On the left-hand side of the middle section of each subgraph the PNS decides to wait for the fast device. If the time until the fast device becomes available, which is the value on the x-axis, is high enough the PNS decides not to use the fast device and launches the kernel immediately on the slow device (see Section 6.4).

The maximum speedups in these areas are up to 96% (and 74% on average) of the theoretical maximum speedups (see Section 6) that assume that kernels can migrate at any point in time, execution on the slow device does not cause interference, and no work is thrown away due to slice aborts (see Section 3). Reduction with the small input has the largest gap between the measured maximum speedus and the theoretical maximum. This is because the predicted slice size divides the iteration space into very few slices (see Table 3), this in turn means that significant amounts of work are thrown away when the kernel migrates because the slice that is at that time on its slow device is aborted (see Section 3.3).

Towards the right-hand end of each subgraph the new application kernel does not migrate because its fast device is unavailable for longer than the execution time of the new application kernel on its slow device. The PNS chooses to execute the new kernel on its slow device in these cases, but has no slicing overheads. Most of these tail points are just under 1x and the geometric mean of the slowdowns on the tails is 2.34%. This is as expected because the training slice sizes for the predictors have overheads of up to 2% to 4% (see Section 4.3)

The geometric means in Figure 10 exclude the final three points from the tail towards the right-hand end of each figure. This is because the means are intended to capture the trade-off between those areas of the subgraph in which our technique generates a speedup, and those areas in which it generates a slowdown. Since the right-hand end tail is arbitrarily extendable (i.e. the fast device could be unavailable for an arbitrary amount of time), and since it inevitably converges to be close to one, its impact would eventually swamp the mean and also converge it to one. This would leave us with no useful information about the areas of practical interest. The maximum per kernel instance geometric mean speedup is 1.15x with the Nearest Neighbor kernel and the large problem size. The same has been applied to the overall average stated above for the same reason. The rightmost tail points are above or below their preceding points in some cases. We confirmed experimentally

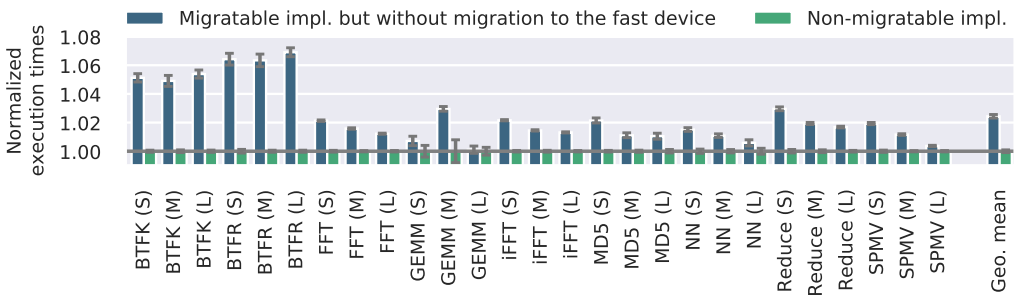


Fig. 11. This figure shows that the overheads of slicing if a kernel never migrates are small. Blue bars are execution times only on the slow devices but with our migration mechanism and slice size prediction, and green bars are execution times without slicing normalized to the latter (lower is better). All green bars have a height of one. The letters in brackets indicate the input sizes **s**mall, **m**edium, and **l**arge (see Section 7.2). **BT**ree **F**ind **K**, **BT**ree **F**ind **R**ange, Inverse FFT, MD5Hash and **N**earest **N**eighbor are abbreviated to BTFK, BTFR, iFFT, MD5, and NN respectively. We collected 100 samples for each data point.

that this is not the general trend of the subsequent migration time points. Subsequent points are closer to the other two tail points shown. In summary, mid-kernel migration outperforms the PNS in accordance with the model of Section 6 by up to 1.30x and 1.08x on average.

7.4 Overheads in the Absence of Migration

This section shows that our implementation of the migration mechanism introduces only small slowdowns of 2.44% on average if an application kernel never migrates from the slow device to the fast device. Time spent in additional code required for migration, like slicing code, which is continuously executed on the slow device introduces overheads that cannot be amortized in these situations (see Sections 3 and 4.1). Figure 11 shows execution times on only the slow devices with the migration capable implementations normalized to execution times without the migration mechanism. The slice size choices by the predictors also govern the slowdowns because they determine how often the slicing code is executed. Therefore, the results show that slicing can be implemented with low overheads and that predictors can learn slice size choices that introduce acceptable overheads in cases where a kernel never migrates.

7.5 Code Size Reduction with Parallel_For

Our parallel_for based programming model hides the complexity of mid-kernel migration. Table 4 demonstrates that the significant complexity that a manual implementation of mid-kernel migration would introduce can be hidden with our parallel_for. The parallel_for based code is at least an order of magnitude smaller for three reasons. Firstly, it implements code required for an efficient implementation of migration, which is discussed in Section 3, like slicing aware data transfers and chunked transfers. Secondly, programmers do not have to provide multiple versions of the same code for the target devices in CUDA, OpenCL, and OpenMP. Compared to a manual implementation our parallel_for reduces the code size by at least 88%.

8 RELATED WORK

Kernel slicing has been used in various contexts [2, 10, 11, 33, 39, 41, 53]. Lösch et al. present preliminary work on kernel migration [39]. However, the authors expose significantly more complexity to

Table 4. This table shows that mid-kernel migration would introduce significant complexity if implemented by hand, and that our parallel_for removes this complexity from application code. Lines of code (LOC) for code that is only concerned with migration are provided in brackets. The total LOC for the hand-implementation (not in brackets) includes this code. The LOC do not include comments, includes, defines, and blank lines. The final column presents the code size reduction.

Benchmark	LOC of the hand implementation	LOC with our parallel_for	Reduction
BTree Find K	731 (371)	55	92%
BTree Find Range	824 (382)	74	91%
Nearest Neighbor	634 (402)	29	95%
FFT	559 (392)	30	95%
Inverse FFT	559 (387)	30	95%
GEMM	633 (371)	67	89%
MD5Hash	598 (459)	74	88%
Reduction	589 (382)	6	99%
SPMV	820 (481)	45	95%

programmers because they require that different kernel implementations, one for each device, have to be provided and manually adjusted so that they are still correct in the presence of migration. In contrast, we hide the complexity of mid-kernel migration behind our parallel_for. Similarly, the granularity at which migration decisions are made is set manually in their system. In contrast, we make this, again, programmer transparent with our slice size predictors and programming interface. The authors also do not attempt to avoid unnecessary data transfers of inputs that have already been processed as we do with slicing aware data transfers. Besides that, slicing and data transfer chunking have been used without migration in real-time systems as mechanism to preempt GPU kernels and their data transfers [2, 33, 53]. Slicing has been also used by works that split kernel executions over multiple devices in heterogeneous systems. Cho et al. execute kernels in slices to parallelize the execution of a slice with the preprocessing of the next slice for split computation on the CPU and GPU [11]. Pandit et al. slice on the CPU to track which parts of the iteration space have been processed by the CPU [41] so that the GPU does not execute the same iteration points. Chen et al. control with slicing the number of concurrently active work groups on GPUs to reduce cache contention [10].

The following runtime systems make scheduling decisions on a *kernel-by-kernel* basis and, therefore, approximate the PNS (see Section 6.1). HTrOP uses a basic scheduling technique that does not estimate when an occupied device will become available, but only considers whether a device is available or not [45]. Rinnegan estimates waiting times for devices and per device kernel performance with an error of 8%-16% for the former and an initial training phase [42]. The error of the latter is not reported. CheCL checkpoints OpenCL applications between OpenCL kernels and can migrate and restart applications at checkpoints [48]. Harmony and StarPU dynamically schedule task graphs to heterogeneous devices without the ability to migrate launched tasks to another device [1, 22]. ConSerner automatically generates data transfer code for devices with separate memory from C code [24]. The authors pair ConSerner in their evaluation with a scheduler that is capable of kernel-by-kernel migration. Application kernels could manually be made migratable with kernel-by-kernel scheduling by decomposing them into multiple sub-kernels. However, this would expose to the programmer the complexity of the decomposition, slicing aware data transfers, data transfer chunking, and migration and kernel abort strategies (see Section 3). Without our slice

size predictors the decomposition granularity needs to be chosen manually, potentially with a time consuming brute force search.

Several works distribute the execution of a kernel over multiple heterogeneous devices at the same time [11, 25, 31, 40, 46]. However, these works are not concerned with migration. They also do not investigate the overheads that kernel partitioning mechanisms such as slicing introduce if a kernel executes on only one device at a time. In contrast, we show that slicing introduces negligible overheads even if a kernel never migrates.

Systems like Effisha [8] and GPUart [29] enable kernel preemption by inserting preemption points into CUDA kernels. This approach is not concerned with migration but with pausing kernels and saving their execution states. Guo et al. dynamically schedule kernels between multiple GPUs while allowing migration of unlaunched kernels to optimize load balancing [28].

We use idempotent function parameters for our parallel_for for efficient migration strategies (see Section 3.2). Previous work introduced *idempotent memory references* and exploited such references in speculatively executed code [36]. In contrast, we are concerned with idempotent computational kernels. We consider a kernel idempotent if it can be safely re-executed for any iteration point (see Section 3.2).

OpenMP's `guided` scheduler decreases chunk sizes over the execution of a parallel loop [5]. This strategy cannot replace our slice size predictors because none of OpenMP's chunk size choices are informed by the user code. Too large slice size choices can have a large negative performance impact. The larger the slice size the longer executes an already launched slice on the old device after migration and interferes with the execution on the new device. This would be the worst if a kernel migrates soon after it launched because the first slice sizes that `guided` would choose are the largest. Additionally, without user intervention `guided` would choose slice sizes of size one at the end of the execution of a kernel, which would vastly underutilize GPUs and also otherwise introduce high overheads.

9 CONCLUSION AND FUTURE WORK

Operating Systems do not yet have the same fine-grained migration capabilities for heterogeneous systems that they have for CPUs. We provide more flexibility in this respect, including the ability to switch between the underlying CUDA, OpenCL, and OpenMP runtimes. Our parallel_for based programming model hides the complexity of the migration mechanism, its efficient implementation, and slice size choices from application developers. Mid-kernel migration has two benefits: firstly, kernels can utilize better devices when they become available mid-kernel, and secondly, in our evaluation scenario, perfect scheduling decisions that require unattainable knowledge can be replaced with a better scheduling policy that does not require such knowledge. We show analytically that mid-kernel migration outperforms kernel-by-kernel scheduling, which is typical for current systems by up to 1.33x, and that an idealized implementation of the new scheduling policy never performs worse than kernel-by-kernel scheduled systems even if they make perfect scheduling decisions. Our experimental results show that mid-kernel migration performs better than these systems by up to 1.30x and 1.08x on average.

The proposed mechanism gives high-level scheduling policies more freedom. We briefly discuss examples of potential future work on policies that make use of this new flexibility. Firstly, systems with priority scheduling will benefit. Lower priority kernels could be migrated instead of just being preempted when kernels with higher priorities require their current device. Secondly, static scheduling heuristics for task graphs with known task-lengths could be extended [37]. With kernel-migration, extensions to these heuristics could reduce makespans by migrating tasks on the critical path to faster devices. Lastly, schedulers that co-schedule computations in interference-aware ways could benefit from mid-kernel migration [9, 21]. In this context, the best task-to-resource

assignments might change as applications come and go. For example, if a new kernel has a strong preference for a device, it might be beneficial for overall system performance to migrate other kernels, which would cause interference, from that device to other devices.

ACKNOWLEDGMENTS

We would like to thank Hugh Leather and Chris Cummins for helpful discussions. This work was supported by the Engineering and Physical Sciences Research Council, EPSRC Centre for Doctoral Training in Pervasive Parallelism at the University of Edinburgh, School of Informatics (grant EP/L01503X/1).

REFERENCES

[1] Cédric Augonnet, Samuel Thibault, Raymond Namyst, and Pierre-André Wacrenier. 2009. StarPU: A Unified Platform for Task Scheduling on Heterogeneous Multicore Architectures. In European Conf. on Parallel Processing. Springer, 863–874.

[2] Can Basaran and Kyoung-Don Kang. 2012. Supporting Preemptive Task Executions and Memory Copies in GPGPUs. In Euromicro Conf. on Real-Time Systems. IEEE, 287–296.

[3] David Alexander Beckingsale, Jason Burmark, Rich Hornung, Holger Jones, William Killian, Adam J. Kunen, Olga Pearce, Peter Robinson, Brian S. Ryujin, and Thomas R.W. Scogland. 2019. RAJA: Portable Performance for Large-Scale Scientific Applications. In Int'l. Workshop on Performance, Portability and Productivity in HPC. IEEE, 71–81.

[4] Ashikahmed Bhuiyan, Kecheng Yang, Samsil Arefin, Abusayeed Saifullah, Nan Guan, and Zhishan Guo. 2019. Mixed-Criticality Multicore Scheduling of Real-Time Gang Task Systems. In Real-Time Systems Symp. IEEE, 469–480.

[5] OpenMP Architecture Review Board. 2020. OpenMP Application Programming Interface. Version 5.1.

[6] Leo Breiman. 2001. Random Forests. Machine learning 45 (2001). Springer, 5–32.

[7] Shuai Che, Michael Boyer, Jiayuan Meng, David Tarjan, Jeremy W. Sheaffer, Sang-Ha Lee, and Kevin Skadron. 2009. Rodinia: A Benchmark Suite for Heterogeneous Computing. In Int'l. Symp. on Workload Characterization. IEEE, 44–54.

[8] Guoyang Chen, Yue Zhao, Xipeng Shen, and Huiyang Zhou. 2017. Effisha: A Software Framework for Enabling Efficient Preemptive Scheduling of GPU. In Symp. on Principles and Practice of Parallel Programming. ACM, 3–16.

[9] Quan Chen, Hailong Yang, Minyi Guo, Ram Srivatsa Kannan, Jason Mars, and Lingjia Tang. 2017. Prophet: Precise QoS Prediction on Non-Preemptive Accelerators to Improve Utilization in Warehouse-Scale Computers. In Int'l. Conf on Architectural Support for Programming Languages and Operating Systems. ACM, 17–32.

[10] Yanhao Chen, Ari B. Hayes, Chi Zhang, Timothy Salmon, and Eddy Z. Zhang. 2018. Locality-Aware Software Throttling for Sparse Matrix Operation on GPUs. In USENIX Annual Technical Conf. USENIX, 413–425.

[11] Younghyun Cho, Florian Negele, Seohong Park, Bernhard Egger, and Thomas R. Gross. 2018. On-The-Fly Workload Partitioning for Integrated CPU/GPU Architectures. In Int'l. Conf. on Parallel Architectures and Compil. Techniques. ACM, 1–13.

[12] Hongsuk Chung, Munsik Kang, and Hyun-Duk Cho. 2012. Heterogeneous Multi-Processing Solution of Exynos 5 Octa with ARM® big.LITTLE™ Technology. Samsung White Paper (2012).

[13] Ryan Cochran, Can Hankendi, Ayse K. Coskun, and Sherief Reda. 2011. Pack & Cap: Adaptive DVFS and Thread Packing Under Power Caps. In Int'l. Symp. on Microarchitecture. IEEE, 175–185.

[14] Murray I. Cole. 1989. Algorithmic Skeletons: Structured Management of Parallel Computation. MIT Press.

[15] Alexander Collins, Tim Harris, Murray Cole, and Christian Fensch. 2015. Lira: Adaptive Contention-Aware Thread Placement for Parallel Runtime Systems. In Int'l. Workshop on Runtime and Operating Systems for Supercomputers. ACM, 1–8.

[16] Timothy Creech, Aparna Kotha, and Rajeev Barua. 2013. Efficient Multiprogramming for Multicores with SCAF. In Int'l. Symp. on Microarchitecture. ACM, 334–345.

[17] Anthony Danalis, Gabriel Marin, Collin McCurdy, Jeremy S. Meredith, Philip C. Roth, Kyle Spafford, Vinod Tipparaju, and Jeffrey S. Vetter. 2010. The Scalable HeterOgeneous Computing (SHOC) Benchmark Suite. In Workshop on General-Purpose Computation on Graphics Processing Units. ACM, 63–74.

[18] Marc de Kruijf, Karthikeyan Sankaralingam, and Somesh Jha. 2012. Static Analysis and Compiler Design for Idempotent Processing. In Conf. on Programming Language Design and Implementation. ACM, 475–486.

[19] Dionisio de Niz, Karthik Lakshmanan, and Ragunathan (Raj) Rajkumar. 2009. On the Scheduling of Mixed-Criticality Real-Time Task Sets. In Real-Time Systems Symp. IEEE, 291–300.

[20] Jeffrey Dean and Sanjay Ghemawat. 2008. MapReduce: Simplified Data Processing on Large Clusters. Commun. ACM 51, 1 (2008). ACM, 107–113.

[21] Christina Delimitrou and Christos Kozyrakis. 2013. Paragon: QoS-Aware Scheduling for Heterogeneous Datacenters. ACM SIGPLAN Notices 48, 4 (2013). 77–88, ACM.
[22] Gregory Diamos and Sudhakar Yalamanchili. 2008. Harmony: An Execution Model and Runtime for Heterogeneous Many Core Systems. In Int'l. Symp. on High Perf. Distributed Computing. ACM, 197–200.
[23] H. Carter Edwards, Christian R. Trott, and Daniel Sunderland. 2014. Kokkos: Enabling Manycore Performance Portability Through Polymorphic Memory Access Patterns. J. Parallel and Distrib. Comput. 74, 12 (2014). Elsevier, 3202–3216.
[24] Ramy Gad, Tim Süß, and André Brinkmann. 2014. Compiler Driven Automatic Kernel Context Migration for Heterogeneous Computing. In Int'l. Conf. on Distributed Computing Systems. IEEE, 389–398.
[25] Dominik Grewe and Michael F.P. O'Boyle. 2011. A Static Task Partitioning Approach for Heterogeneous Systems Using OpenCL. In Int'l. Conf. on Compiler Construction. Springer, 286–305.
[26] Dominik Grewe, Zheng Wang, and Michael F.P. O'Boyle. 2013. Portable Mapping of Data Parallel Programs to OpenCL for Heterogeneous Systems. In Int'l. Symp. on Code Generation and Optimization. IEEE, 1–10.
[27] Khronos SYCL™ Working Group. 2020. SYCL™ Specification, SYCL™ Integrates OpenCL™ Devices with Modern C++. Version 1.2.1.
[28] Fan Guo, Yongkun Li, John C.S. Lui, and Yinlong Xu. 2019. DCUDA: Dynamic GPU Scheduling with Live Migration Support. In Symp. on Cloud Computing. ACM, 114–125.
[29] Christoph Hartmann and Ulrich Margull. 2019. GPUart - An Application-Based Limited Preemptive GPU Real-Time Scheduler for Embedded Systems. Journal of Systems Architecture 97 (2019). Elsevier, 304–319.
[30] John L. Hennessy and David A. Patterson. 2017. Computer Architecture: A Quantitative Approach (6th ed.). Morgan Kaufmann Publishers. The subsection on Amdahl's Law in Section 1.9, 49–50.
[31] Rashid Kaleem, Rajkishore Barik, Tatiana Shpeisman, Brian T. Lewis, Chunling Hu, and Keshav Pingali. 2014. Adaptive Heterogeneous Scheduling for Integrated GPUs. In Int'l. Conf. on Parallel Architectures and Compil. Techniques. ACM, 151–162.
[32] Melanie Kambadur, Tipp Moseley, Rick Hank, and Martha A. Kim. 2012. Measuring Interference Between Live Datacenter Applications. In Int'l. Conf. on High Perf. Computing, Networking, Storage and Analysis. IEEE, 1–12.
[33] Shinpei Kato, Karthik Lakshmanan, Aman Kumar, Mihir Kelkar, Yutaka Ishikawa, and Ragunathan (Raj) Rajkumar. 2011. RGEM: A Responsive GPGPU Execution Model for Runtime Engines. In Real-Time Systems Symp. IEEE, 57–66.
[34] Onur Kayıran, Adwait Jog, Mahmut T. Kandemir, and Chita R. Das. 2013. Neither More Nor Less: Optimizing Thread-level Parallelism for GPGPUs. In Int'l. Conf. on Parallel Architectures and Compil. Techniques. IEEE, 157–166.
[35] Hyunjun Kim, Sungin Hong, Hyeonsu Lee, Euiseong Seo, and Hwansoo Han. 2019. Compiler-Assisted GPU Thread Throttling for Reduced Cache Contention. In Int'l. Conf. on Parallel Processing. ACM, 1–10.
[36] Seon Wook Kim, Chong-Liang Ooi, Rudolf Eigenmann, Babak Falsafi, and T.N. Vijaykumar. 2001. Reference Idempotency Analysis: A Framework for Optimizing Speculative Execution. In Symp. on Principles and Practices of Parallel Programming. ACM, 2–11.
[37] Yu-Kwong Kwok and Ishfaq Ahmad. 1999. Static Scheduling Algorithms for Allocating Directed Task Graphs to Multiprocessors. Comput. Surveys 31, 4 (1999). ACM, 406–471.
[38] David Lo, Liqun Cheng, Rama Govindaraju, Parthasarathy Ranganathan, and Christos Kozyrakis. 2015. Heracles: Improving Resource Efficiency at Scale. In Int'l. Symp. on Computer Architecture. ACM, 450–462.
[39] Achim Lösch and Marco Platzner. 2020. MigHEFT: DAG-based Scheduling of Migratable Tasks on Heterogeneous Compute Nodes. In Int'l. Parallel and Distributed Processing Symp. Workshops. IEEE, 6–16.
[40] Chi-Keung Luk, Sunpyo Hong, and Hyesoon Kim. 2009. Qilin: Exploiting Parallelism on Heterogeneous Multiprocessors with Adaptive Mapping. In Int'l. Symp. on Microarchitecture. ACM, 45–55.
[41] Prasanna Pandit and R. Govindarajan. 2014. Fluidic Kernels: Cooperative Execution of OpenCL Programs on Multiple Heterogeneous Devices. In Int'l. Symp. on Code Generation and Optimization. ACM, 273–283.
[42] Sankaralingam Panneerselvam and Michael Swift. 2016. Rinnegan: Efficient Resource Use in Heterogeneous Architectures. In Int'l. Conf. on Parallel Architectures and Compil. Techniques. ACM, 373–386.
[43] Raghu Prabhakar, Yaqi Zhang, David Koeplinger, Matt Feldman, Tian Zhao, Stefan Hadjis, Ardavan Pedram, Christos Kozyrakis, and Kunle Olukotun. 2017. Plasticine: A Reconfigurable Architecture for Parallel Patterns. In Int'l. Symp. on Computer Architecture. IEEE, 389–402.
[44] Arun Raman, Ayal Zaks, Jae W. Lee, and David I. August. 2012. Parcae: A System for Flexible Parallel Execution. ACM SIGPLAN Notices 47, 6 (2012). ACM, 133–144.
[45] Heinrich Riebler, Gavin Vaz, Tobias Kenter, and Christian Plessl. 2019. Transparent Acceleration for Heterogeneous Platforms With Compilation to OpenCL. Transactions on Architecture and Code Optimization 16, 2 (2019). ACM, 1–26.
[46] Jie Shen, Ana Lucia Varbanescu, Peng Zou, Yutong Lu, and Henk Sips. 2014. Improving Performance by Matching Imbalanced Workloads with Heterogeneous Platforms. In Int'l. Conf. on Supercomputing. ACM, 241–250.

[47] Srinath Sridharan, Gagan Gupta, and Gurindar S. Sohi. 2014. Adaptive, Efficient, Parallel Execution of Parallel Programs. In Conf. on Programming Language Design and Implementation. ACM, 169–180.
[48] Hiroyuki Takizawa, Kentaro Koyama, Katsuto Sato, Kazuhiko Komatsu, and Hiroaki Kobayashi. 2011. CheCL: Transparent Checkpointing and Process Migration of OpenCL Applications. In Int'l. Parallel & Distributed Processing Symp. IEEE, 864–876.
[49] Ben Taylor, Vicent Sanz Marco, and Zheng Wang. 2017. Adaptive Optimization for OpenCL Programs on Embedded Heterogeneous Systems. In Conf. on Languages, Compilers, and Tools for Embedded Systems. ACM, 11–20.
[50] Abhishek Verma, Luis Pedrosa, Madhukar Korupolu, David Oppenheimer, Eric Tune, and John Wilkes. 2015. Large-Scale Cluster Management at Google with Borg. In European Conf. on Computer Systems. ACM, 1–17.
[51] Steve Vestal. 2007. Preemptive Scheduling of Multi-Criticality Systems with Varying Degrees of Execution Time Assurance. In Int'l. Real-Time Systems Symp. IEEE, 239–243.
[52] Zheng Wang and Michael F.P. O'Boyle. 2009. Mapping Parallelism to Multi-cores: A Machine Learning Based Approach. In Symp. on Principles and Practice of Parallel Programming. ACM, 75–84.
[53] Husheng Zhou, Guangmo Tong, and Cong Liu. 2015. GPES: A Preemptive Execution System for GPGPU Computing. In Real-Time and Embedded Technology and Applications Symp. IEEE, 87–97.