The Journal of Supercomputing (2020) 76:5038–5056
https://doi.org/10.1007/s11227-019-02824-7

Hybrid CPU–GPU execution support in the skeleton programming framework SkePU

Tomas Öhberg[1] · **August Ernstsson**[1] · **Christoph Kessler**[1]

Published online: 25 March 2019

Abstract
In this paper, we present a hybrid execution backend for the skeleton programming framework SkePU. The backend is capable of automatically dividing the workload and simultaneously executing the computation on a multi-core CPU and any number of accelerators, such as GPUs. We show how to efficiently partition the workload of skeletons such as Map, MapReduce, and Scan to allow hybrid execution on heterogeneous computer systems. We also show a unified way of predicting how the workload should be partitioned based on performance modeling. With experiments on typical skeleton instances, we show the speedup for all skeletons when using the new hybrid backend. We also evaluate the performance on some real-world applications. Finally, we show that the new implementation gives higher and more reliable performance compared to an old hybrid execution implementation based on dynamic scheduling.

Keywords Heterogeneous computing · Hybrid execution · Skeleton programming · Workload partitioning

1 Introduction

The ever-growing demand for higher performance in computing, puts requirements on modern programming tools. Today parallelism stands for the majority of the performance potential and even if heterogeneous, multi-core and accelerator equipped systems have been the norm for more than a decade, we still face the challenge of automatically exploiting the performance potential of such systems. An effective parallel programming framework should not only let the programmer implement the

✉ August Ernstsson
august.ernstsson@liu.se

Tomas Öhberg
tomasolof@hotmail.com

[1] PELAB, Department of Computer and Information Science, Linköping University, Linköping, Sweden

applications to run on any processing unit the hardware provides, but also to run on *all* processing units, dividing the workload between multiple processing units, possibly of different kind. This way of simultaneously executing an algorithm on multiple, heterogeneous processing units is referred to as *hybrid execution*. To relieve the burden of partitioning and scheduling from the programmers, the frameworks should preferably figure out the best way to divide the workload automatically. Such a system must take the relative performance of the hardware components of the system executing the application into consideration, as well as the characteristics of the computation.

One approach to hide the complexities of parallel architectures from the programmer is *algorithmic skeletons* [2]. The idea of skeletons is based on higher-level functions (i.e., functions taking other functions as arguments) from the functional programming paradigm. Skeleton programming uses this idea to provide a number of algorithm templates to the programmer, which can be instantiated to solve the problem at hand and at the same time hide a potentially complex parallel implementation of the skeleton behind its interface.

SkePU [6, 8] is a high-level C++ skeleton programming framework for heterogeneous architectures, providing a clean and flexible interface to the programmer. The focus of SkePU is on multi-core, multi-accelerator systems.

A redesign of SkePU was accomplished in 2016 to give the library a new, more flexible and type safe interface by utilizing modern C++11 metaprogramming features. The new version of the library is referred to as SkePU 2. An experimental variant of the old version of SkePU had support for simultaneous CPU–GPU execution through partitioning of skeleton calls into multiple tasks scheduled by the runtime system StarPU.

This paper presents a new hybrid execution backend for the SkePU 2 skeleton framework. The work is based on Öhberg's master's thesis [13]. The main contributions of our work are the following:

- We introduce workload partitioning implementations for all data parallel skeletons in SkePU 2, capable of dividing the work between an arbitrary number of CPU cores and accelerators.
- We present a way to automatically tune the workload partitioning of the skeletons, by using performance benchmarking.
- We show that our hybrid execution implementation can improve the execution time compared to the already existing backends of SkePU, without any changes to the application code.
- We show that the performance of our solution is faster and more robust than the experimental StarPU-based hybrid execution implementation from SkePU 1, by porting its implementation to SkePU 2.

The rest of this paper is structured as follows: Sect. 2 starts by introducing SkePU 2 and its available skeletons. Section 3 introduces the task-based programming library StarPU. Section 4 presents the new hybrid backend implementation and how the workload is partitioned in all skeletons. This is followed by Sect. 5, where the auto-tuner is described. Section 6 describes the evaluations made on the hybrid execution implementation and presents the results. Section 7 discusses related libraries and frameworks with support for heterogeneous architectures. Finally, Sect. 8 contains conclusions and ideas for future work.

2 SkePU 2

SkePU[1] is a high-level C++ skeleton programming framework targeting multi-core, multi-accelerator systems. SkePU was initially presented in 2010 by Enmyren and Kessler [6] as a macro-based library for computations using algorithmic skeletons. SkePU was revised in 2016 by Ernstsson [8] to replace the preprocessor macros by a precompiler and type safe metaprogramming using modern C++11 features. The new version is called SkePU 2 and the rest of this paper will refer to this version of SkePU unless otherwise stated. The old macro-based version of SkePU will be referred to as SkePU 1. Earlier research on SkePU has demonstrated features such as automatic backend selection tuning [5] as well as smart containers for implicit multi-device memory management [3]. SkePU 1 also had an experimental version including support for hybrid execution using the dynamic task parallel runtime system StarPU [4]. Several industry-class applications have been parallelized with SkePU, such as the computational fluid dynamics flow solver EDGE [15].

SkePU has four different *backends*, implementing the skeletons for different hardware. These are:

- **Sequential CPU backend**, mainly used as a reference implementation and baseline.
- **OpenMP backend**, for multi-core CPUs.
- **CUDA backend**, for NVIDIA GPUs, both single and multiple.
- **OpenCL backend**, for single and multiple GPUs of any OpenCL supported model, including other accelerators such as Intel Xeon Phis.

The skeletons in SkePU are instantiated with *user functions* to create skeleton instances. The programmer can write the user functions as regular C++ functions or lambda expressions and let the precompiler translate and generate kernels for the desired backends. An example of a dot product skeleton using MapReduce is shown in Listing 1.

[1] https://www.ida.liu.se/labs/pelab/skepu/.

Listing 1: Dot product computation in SkePU 2.

```
float add(float a, float b) {
  return a + b;
}

float mult(float a, float b) {
  return a * b;
}

skepu2::Vector<float> v1, v2;

auto dotprod = skepu2::MapReduce<2>(mult, add);
float res = dotprod(v1, v2);
```

SkePU 2 includes the following skeletons:

- **Map**, a generic element-wise data parallel skeleton working on vectors and matrices. It has support for element-wise arguments of arbitrary number, as well as random access and uniform user function arguments.
- **Reduce**, generic reduction operation with a single binary, associative operator. Available for vectors and row-wise or column-wise matrix reduction.
- **MapReduce**, an efficient combination of a Map skeleton followed by Reduce.
- **MapOverlap**, stencil operation in one or two dimensions with support for different edge handling schemes.
- **Scan**, a generalize prefix sum with a binary, associative operator.
- **Call**, a generic, multi-variant component.

The Call skeleton will not be discussed further in this paper, as the semantics of the multi-variant component implies that the user must provide a hybrid execution implementation themselves.

SkePU implements its own container types, called *smart containers* [3]. They are available as vectors and matrices, building upon the interface of the C++ standard library containers. Smart containers reside in main memory, but can temporarily copy subsets of its data to accelerator memory to be accessed by skeleton backends executing on these devices. The containers automatically handle all data movement and keep track of which devices have copies of the data including the validity of these copies. Lazy memory copying is applied to ensure data is only copied back to CPU memory when actually needed.

3 StarPU

The old implementation of hybrid execution in SkePU 1 used the StarPU library as a backend. This implementation was ported to SkePU 2 as a baseline to compare the new hybrid backend to. StarPU[2] is a C-based task programming library for

[2] http://starpu.gforge.inria.fr.

hybrid architectures. The goal of StarPU is to provide a unified runtime system for heterogeneous computer systems, including different execution units and programming models. StarPU also offers a high-level C++ interface or, optionally, compiler-extension pragmas.

A task in StarPU is defined in terms of *codelets*. Describing a computational task, codelets are combined with input data to form *tasks*. Tasks are passed to the runtime system asynchronously, and later mapped and scheduled to be executed on any of the available computing resources. The codelets can contain code written in C/C++, CUDA, and OpenCL. StarPU's modular implementation ensures that different scheduling policies and performance models can be used. Examples of scheduling policies include **eager-based**, **priority-based**, and **random-based** schedulers. It is also possible to construct custom schedulers using the pre-implemented scheduling components. Similar to SkePU, StarPU performs its own data transfer optimization by caching data on the computational units where it was last accessed [18].

StarPU has been used in a number of application scenarios, recent examples including finite-volume CFD [1] and seismic wave modeling [12].

4 Workload partitioning and implementation

The new hybrid execution implementation in SkePU is made as a new backend, allowing the programmer to explicitly choose whether or not to use it. During precompilation the hybrid backend is automatically included if the OpenMP and either CUDA or OpenCL is selected. The hybrid backend works with both CUDA and OpenCL. Which accelerator implementation will be used is determined by availability and the programmer's preference.

In the first stage of a skeleton invocation, the workload is partitioned into two parts by the hybrid backend: one for the CPU and one for the accelerators. The CPU and accelerator parts are then further divided between the CPU threads and any number of accelerators, respectively. The hybrid skeleton implementations use OpenMP, where the first thread will manage the accelerators and the rest of the threads will work on the CPU partition. The implementation is very similar to the already existing OpenMP backend, in order to match its performance. To reduce duplication of code within SkePU, the accelerator partition is computed by the already existing CUDA or OpenCL backend implementations. To make this work, some of the internal APIs of the accelerator backends (CUDA and OpenCL) had to be generalized to work on subparts of containers. As both accelerator backends already have support for multi-accelerator computations, also the hybrid backend has support for hybrid execution with multiple accelerators. The workload partitioning in the accelerator backends is however, still limited, as the work is evenly divided between all accelerators. This works well when all accelerators are of the same type, but will not be optimal in case different accelerator models are used.

Listing 2: Using the hybrid backend with a manually set partition ratio.

```
const int NUM_THREADS = 16;
const int NUM_GPUS = 1;
const float PARTITION_RATIO = 0.2;

skepu2::Vector<int> in, out;

skepu2::BackendSpec spec(skepu2::Backend::Type::Hybrid);
spec.setCPUThreads(NUM_THREADS);
spec.setDevices(NUM_GPUS);
spec.setCPUPartitionRatio(PARTITION_RATIO);
skeleton_instance.setBackend(spec);

skeleton_instance(out, in);
```

The workload is partitioned according to a single parameter: the *partition ratio*. The ratio defines the proportion of the workload that should be computed by the CPU; the rest is computed by the accelerators. The partition ratio can either be manually set by the programmer, or automatically tuned per skeleton instance to make SkePU predict the optimal partition ratio for a given input size. The auto-tuning will be described later in this paper. How to use hybrid execution with a manually configured partition ratio is shown in Listing 2. This example shows how to set up hybrid execution for 16 CPU threads and one accelerator, where 20% of the workload will be computed by the CPU threads, the rest by the accelerator. Which accelerator implementation (CUDA or OpenCL) to use is specified by compiler flags.

Map is highly data parallel by nature and is therefore straightforward to partition. The ratio defines how many output elements to compute on the CPU, the rest is computed by the accelerator backend. The CPU partition is further divided into equal-sized blocks, one for each CPU thread. The partitioning scheme of the Map skeleton is shown for three CPU threads in Fig. 1.

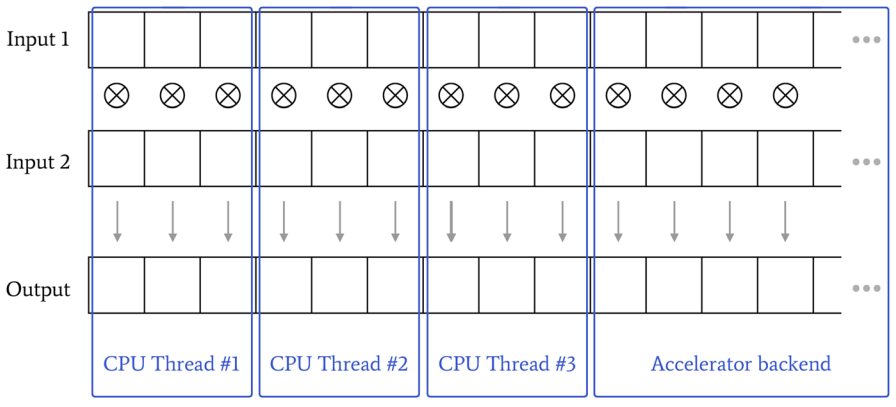


Fig. 1 Partitioning of the Map skeleton

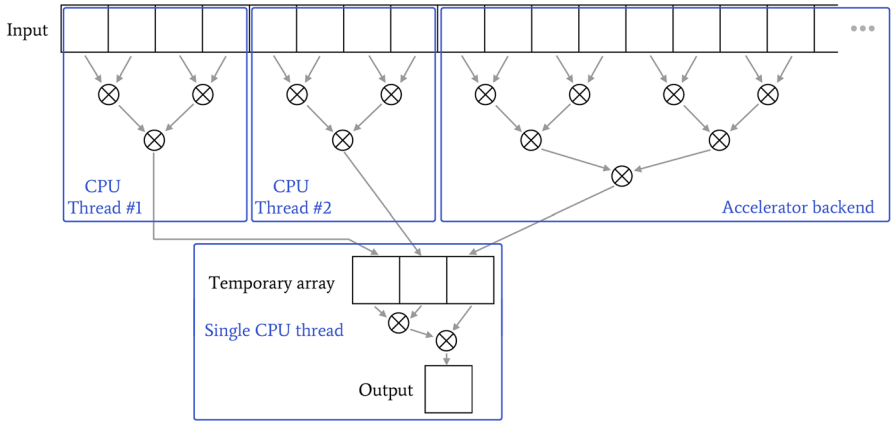


Fig. 2 Partitioning of reduce skeleton

Reduce is performed in two steps. The partition ratio defines how many input elements to be reduced on the CPU, the rest is reduced by the accelerators. The CPU partition is further divided into equally sized blocks, one per CPU thread. First, each CPU thread and the accelerator backend reduce their block of the input data to produce a temporary array of partial reductions. This small array is then reduced by a single CPU thread to a global result. Partitioning of the Reduce skeleton for one-dimensional input containers with two CPU threads is shown in Fig. 2.

MapReduce is implemented in a similar way to the Reduce skeleton. The input arrays are first partitioned as in the Reduce skeleton and the CPU partition is evenly divided between the threads. Each CPU thread and the accelerator backend reduce their part of the data, by first performing the Map step. The intermediate results are then reduced down by a single CPU thread. Partitioning of the MapReduce skeleton is shown in Fig. 3.

MapOverlap is similar to the Map skeleton. The partition ratio defines how many output elements to compute on the CPU, the rest being computed by the accelerators. The CPU partition is then divided into one block per CPU thread. Extra consideration had to be taken to all variations of edge handling and different corner cases caused by the size of the overlap region. Partitioning of the one-dimensional MapOverlap skeleton with an overlap of 1 element on each side is shown in Fig. 4. The work of a single user function call is highlighted in yellow.

Scan has more data dependencies than the other skeletons and requires a more complex partitioning implementation. The input array is partitioned into a CPU and an accelerator part as before, and the CPU partition is further divided into equally sized blocks, one per CPU thread. Each CPU thread and the accelerator backend start by performing a local Scan of their block of the input data. After this step, each block misses the Scan offset of the preceding blocks. The last resulting element of the local Scan of each CPU block are collected into an temporary array and a single CPU thread performs a Scan on that array. This produces an array of the missing offset values of each block. In the second step,

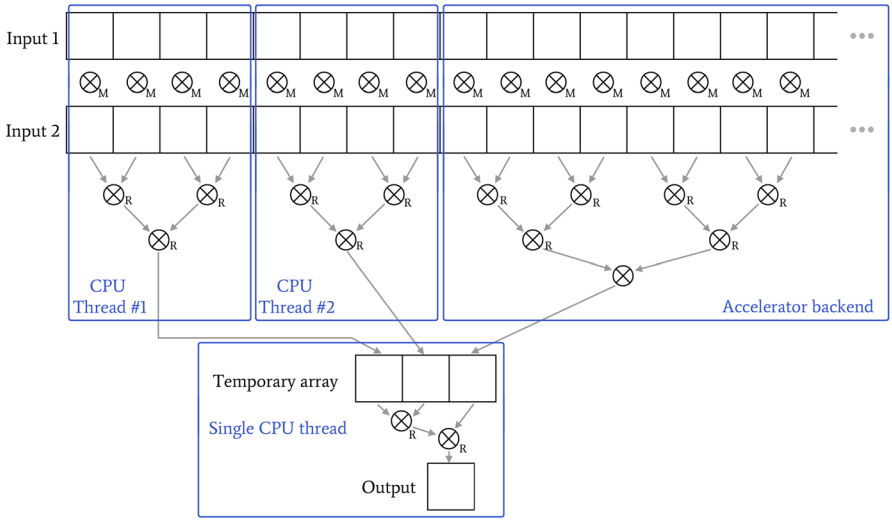


Fig. 3 Partitioning of MapReduce skeleton

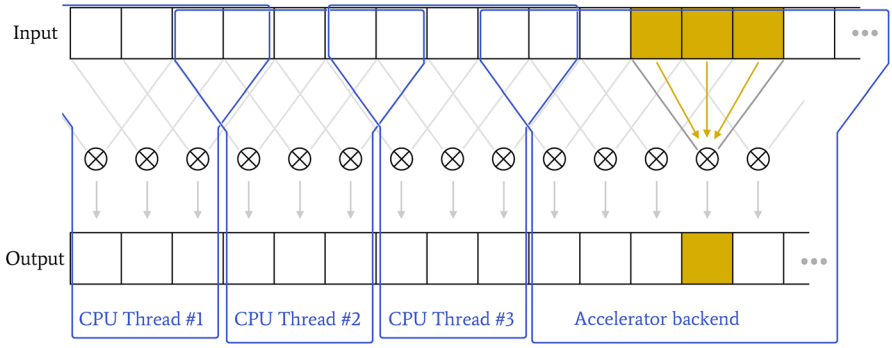


Fig. 4 Partitioning of MapOverlap skeleton

each CPU thread (except for the first, as its block is already complete) combine their local Scan result with the missing value from the array. For the CPU, the local Scan step and the combining step require the same number of operations, one per element in the block. This makes the two steps take approximately the same amount of time. This is not the case for the accelerators on the other hand, especially not a GPU. The first step is much less data parallel and takes longer than the second step where a number of independent operations are made on different data elements. This means that a GPU will go idle if the first and second steps are to be made synchronized with the CPU, as it will finish its second step much faster than the CPU. This was solved by letting the accelerator backend take care of the last part of the input array. As nothing is dependent of the result of the last block, the result of the local Scan of the accelerators' partition is not needed

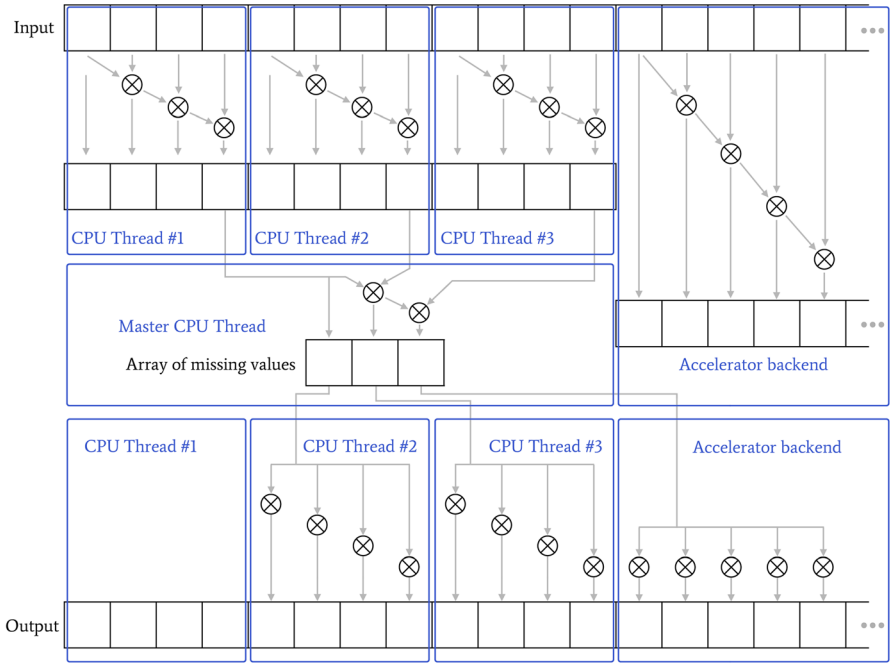


Fig. 5 Partitioning of Scan skeleton

in the missing values array. We can thus let the accelerators spend more time on the first step than the CPU threads are spending, and only make the accelerators check that the CPUs have produced the array of missing values before starting the second step. This makes load balancing between CPU and accelerators much easier and utilizes the available processing capacity better. Partitioning of the Scan skeleton is shown in Fig. 5.

Apart from the partitions shown here, there are also variants for Map on matrices, one-dimensional and two-dimensional Reduce on matrices as well as row-wise MapOverlap on matrices. The Map implementation partitions the elements between the PUs based on the partition ratio, just as if it was an array. In the case of Reduce and MapOverlap, the matrix is partitioned row-wise, so all PUs get whole rows to operate on. This can make it hard to balance the workload for matrices with few rows. However, in connection with automatic backend selection tuning [5], such cases would probably not select the hybrid execution at all. A more sophisticated partitioning for matrices would be hard to realize, especially for the MapOverlap skeleton, due to the many corner cases and complex data access patterns.

4.1 StarPU backend implementation

To show the advantages of the static workload partitioning in the new hybrid execution backend, the experimental StarPU integration from SkePU 1 was ported to SkePU 2. Similar to SkePU, StarPU uses its own custom data management system. In order to keep SkePU's smart container API [3], the smart containers automatically transfer the control to StarPU once they are used with the StarPU backend. The control is taken back by SkePU once the container is used with one of the other backends. The memory management code in SkePU 2 was not changed since SkePU 1, allowing this part of the code to be reused from the old SkePU 1 integration of StarPU. StarPU is integrated into SkePU 2 as a separate backend, just as our hybrid execution implementation. This, together with the memory management implementation, allows the already existing SkePU backends to be used alongside the StarPU backend. No changes had to be made to the API of SkePU, apart from adding StarPU as an extra backend type. Currently, only the main features of the Map, Reduce and MapReduce skeletons are ported, but more skeletons and features will be ported in the future.

As StarPU is a task-based programming framework, a SkePU skeleton invocation must be mapped to a number of tasks. This is done by splitting the workload into a number of equal-sized chunks. The programmer can manually tweak the number of chunks, as this affects the performance. More chunks are desirable for larger input sizes as it makes load balancing easier, but for small input sizes, too many chunks will lead to significant scheduling overheads. The StarPU backend has two implementation variants, one using OpenMP and one using CUDA. The already existing OpenMP and CUDA backend implementations could not be reused due to the abstraction gap between SkePU and StarPU. This gap was noticed already during the integration of StarPU into SkePU 1, and it has since grown even more in SkePU 2 with the increased use of metaprogramming and other high-level C++ features. StarPU uses a lower level C-style API and passes arguments using void pointers and runtime type casting. SkePU 2, on the other hand, builds argument lists at compile time using variadic templates and parameter packs. It was still possible to integrate them by implementing the StarPU functions as static member functions, creating the argument handling code at compile time.

5 Auto-tuning

Apart from manually setting the partition ratio, SkePU can automatically predict a suitable partition ratio by performance benchmarking. Due to how general and flexible the SkePU framework is, implementing an auto-tuner that works well for every imaginable skeleton instance may not be possible. The execution time could, for example, be bound by the size of the random access containers, by uniform arguments, or even be data-dependent on container elements. Predicting optimal partition ratio for such skeleton instances would require very sophisticated and time-consuming algorithms and/or user interaction. Instead focus was put on implementing a tuner that would give good predictions for common cases, where the execution time

grows linearly with the size of the element-wise accessed containers. For specific skeleton instances, the partition ratio can always be set by hand.

The auto-tuning presented in this paper resembles the tuning presented by Luk et al. [11] in their Qilin framework, although our tuner extends this work by supporting multiple accelerators. This is possible as our implementation sees multiple accelerators as one single device, thanks to the already existing multi-device implementations in the CUDA and OpenCL backend. Our tuner builds two execution time models, one for the CPU and one for the accelerator backend. At tuning time, the programmer must choose the number of CPU threads and accelerators to tune for. The tuning is performed once per each skeleton instance in the application for a specific machine. In case the configuration of the machine changes (for example if accelerators are added/removed or if more or less CPU cores are used), the skeleton instance has to be re-tuned. The tuning is performed on the OpenMP and CUDA/OpenCL backends. The OpenMP backend will be executed with one thread less than specified by the programmer, as one thread in the hybrid backend will be fully dedicated to running the accelerator backend. The programmer can choose upper and lower limits to the input size, as well as the number of input sizes to benchmark. The input sizes to benchmark is then evenly spread over the interval defined by the limits. The OpenMP and CUDA/OpenCL backends are executed five times on each input size, and the median value is inserted into the execution time model of that backend. This is made to minimize the impact of temporary fluctuations.

Once the execution time benchmarks are stored in the model, the model is fitted to a linear curve using least-squares fitting. As the execution time grows linearly with all skeletons (assuming the user function takes $\mathcal{O}(1)$ time), a linear approximation of the execution time is sufficient for our needs. The fitting will create two linear equations on the form:

$$t = ax + b \tag{1}$$

where t is the execution time, x is the input size and a and b are parameters found by the least-squares fitting.

Let us consider a problem size N and a (CPU) partition ratio R. The partition size of the CPU will then be NR and the partition size of the accelerator $N(1-R)$. This gives us execution times t_{cpu} and t_{acc} for the CPU and accelerator, respectively:

$$t_{\text{cpu}} = a_{\text{cpu}} NR + b_{\text{cpu}} \tag{2}$$

$$t_{\text{acc}} = a_{\text{acc}} N(1-R) + b_{\text{acc}} \tag{3}$$

The workload is perfectly balanced between the CPU and the accelerator if $t_{\text{cpu}} = t_{\text{acc}}$. Combining the Eqs. 2 and 3 and solving for the partition ratio R gives:

$$R = \frac{a_{\text{acc}} N + b_{\text{acc}} - b_{\text{cpu}}}{N(a_{\text{cpu}} + a_{\text{acc}})} \tag{4}$$

At runtime Eq. (4) is used to predict the optimal partition ratio for a given input size N. In practice, the value of R can be in three intervals: the interval $0 < R < 1$, where hybrid execution is predicted the optimal strategy, and the value of R is used

as the partition ratio; $R \leq 0$, where accelerator-only execution is considered optimal; or $R \geq 1$, where CPU-only execution is considered optimal. In the last two cases, the hybrid backend will automatically fall back to executing the skeleton using the OpenMP or CUDA/OpenCL backends, as the overhead of hybrid execution is predicted to be too high.

6 Performance evaluation

The implementation was evaluated on a system consisting of two octa-core Intel Xeon E5-2660 (16 cores in total) clocked at 2.2 GHz with 64 GB of memory and a NVIDIA Tesla K20x GPU with 2688 processor cores and 6 GB of device memory. The programs were compiled with nvcc (v7.5.17) using g++ (v4.9.2) as host compiler.

6.1 Single skeleton evaluation

First each skeleton type was evaluated with typical user functions. Input sizes ranging from 100,000 to 4,000,000 in increments of 100,000 were used. Each input size and backend combination was executed seven times, and the median execution time was noted to eliminate outliers caused by other operating system processes occasionally running on the CPU. The predicted partition ratio used in the hybrid backend was also noted for each input size. The hybrid backend was tuned with the auto-tuner a single time, and the same execution time model was then used for all input sizes. The results are shown in Fig. 6. As can be seen in the graphs, the hybrid backend improves upon the performance of the OpenMP and CUDA backends for all skeletons, at least as the input size grows. For most skeletons, the hybrid backend even manages to match the performance of the OpenMP and CUDA backends for small input sizes, by switching to CPU-only or GPU-only execution. For the Scan skeleton however, a leap in the hybrid backend curve can be seen, where the partition ratio prediction switches from CPU-only to hybrid too early, as the predictor overestimates the performance of hybrid execution. This is likely due to the extra complexity of the hybrid execution implementation of the Scan skeleton, where the performance of the CPU and the accelerator partitions do not completely match the performance of the OpenMP and CUDA/OpenCL backends used in the auto-tuning.

6.2 Generic application evaluation

To evaluate the implementation in a more realistic context, we also compared the performance of the new hybrid scheme on some of the example applications provided with the SkePU source code. A presentation of the applications and which skeletons they use is shown in Table 1. In the Skeletons column, the number within <> tells the arity of the skeleton instance, i.e., how many element-wise accessed input containers it uses.

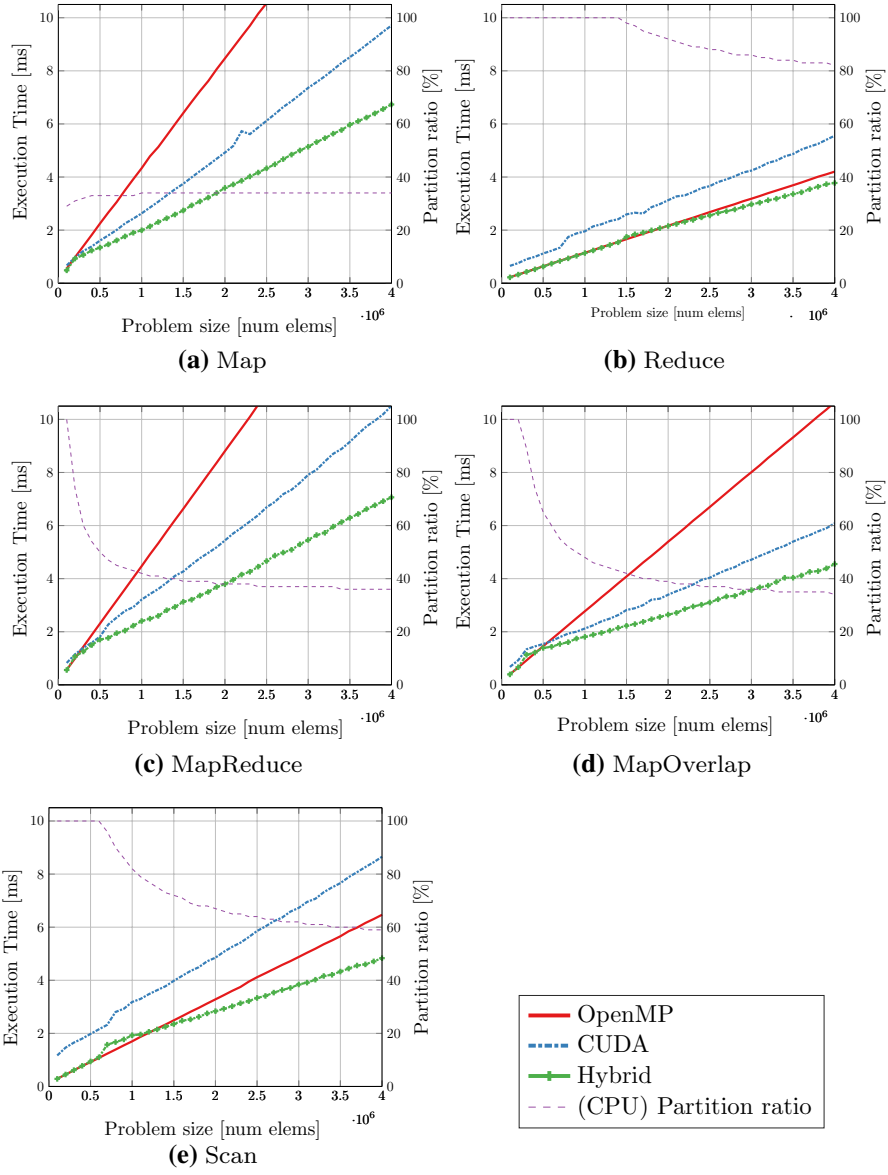


Fig. 6 Execution time of skeletons

The applications were executed with five different configurations. First with the sequential CPU backend as a baseline. Then the OpenMP, CUDA and hybrid backends. For the hybrid backend, all skeletons were tuned with the auto-tuner. Tuning was done with 10 steps and the tuning time was not included in the measured execution time. Finally, we used an *oracle* to find the upper limit to the speedup possible to achieve with the hybrid backend implementation, given an

Table 1 List of applications used in the evaluation

Application	Algorithm	Skeletons
CMA	Cumulative moving average	Map<1>, Scan
Coulombic	Coulombic potential	Map<1>
Dotproduct	Dot product	MapReduce<2>
Mandelbrot	Mandelbrot fractal	Map<0>
PPMCC	Pearson product-moment correlation coeff.	Reduce, MapReduce<1>, MapReduce<2>
PSNR	Peak signal to noise ratio	Map<2>, MapReduce<2>
Taylor	Taylor series expansion of $\log(1+x)$	MapReduce<0>

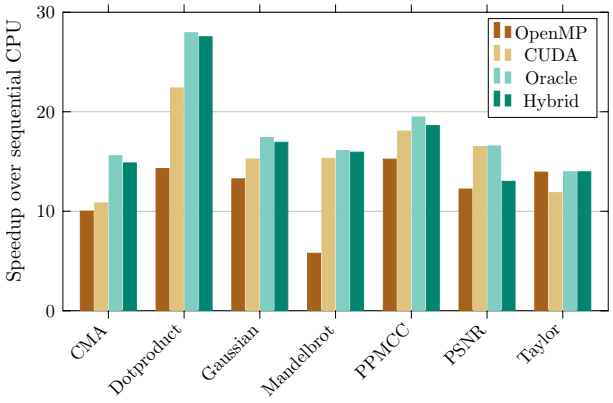


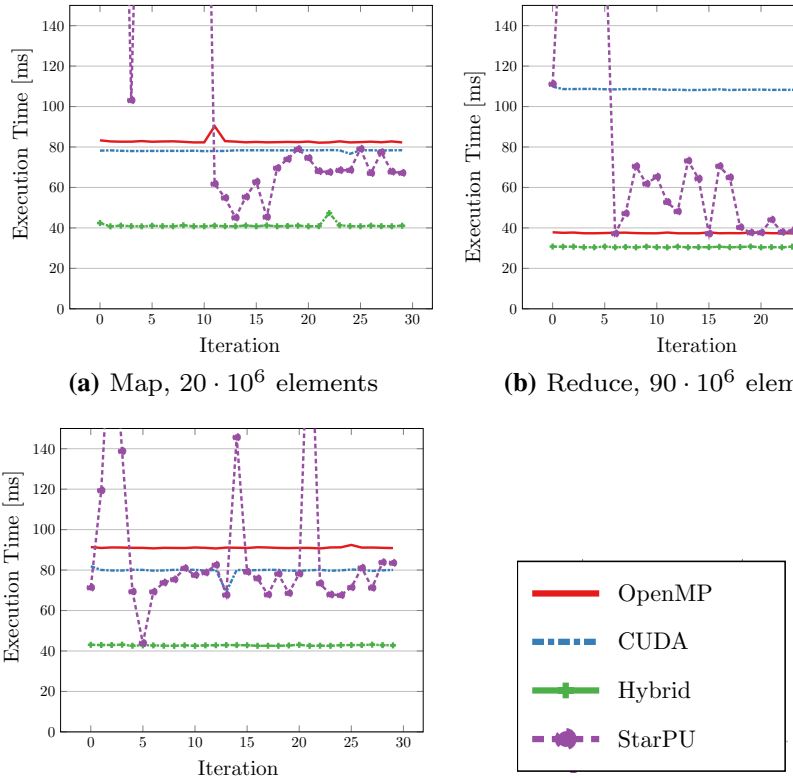
Fig. 7 Speedups comparisons of example applications

optimal partition ratio choice. Oracles has been used in earlier research to show the upper bound of hybrid execution implementations [9, 11, 14]. We let the oracle execute the application using the hybrid backend with a manually set partition ratio for each skeleton instance, ranging from 0 to 100% in increments of five percentage points. For multi-skeleton applications, all combinations of ratios were tested. The fastest of these execution times was then saved as the oracle's time. All backends, including all partition ratio combinations tested by the oracle, were executed seven times, and the median execution time was used. The results are shown in Fig. 7. The figure shows that the hybrid backend improves upon the OpenMP and CUDA backends in most applications. By comparing the hybrid bar to the oracle bar we can see that the auto-tuning finds good partition ratios, but there is some room for improvement. According to the oracle, two of the applications (PSNR and Taylor) does not gain from hybridization, at least not the tested problem sizes. This is also found by the auto-tuner in the Taylor case, as it falls back to CPU-only execution. PSNR is the only application where the hybrid backend fails to improve upon the performance of the OpenMP and CUDA backends. The reason for this is that the auto-tuning finds the optimal partition ratio to be

40% for the Map skeleton and CPU-only for the MapReduce skeleton. Although this is the optimal partition ratio for each individual skeleton instance, it is not the optimal choice when both skeletons are considered because of the need to move data between CPU and GPU memory. According to the oracle, offloading all data to the GPU gives the best execution time in this case.

6.3 Comparison to dynamic hybrid scheduling using StarPU

Finally, we show the improvement over the experimental hybrid execution implementation based on the StarPU runtime system that was implemented in SkePU 1. To make a fair comparison, parts of the old StarPU implementation was ported to SkePU 2. We also compared the execution time to the OpenMP and CUDA backends. As StarPU is supposed to get better over time by learning how to schedule the work, we tried executing the same skeleton multiple times. Each backend was executed 30 times in a row. New input containers were allocated each time to rule out the impact of data locality. The results are shown in Fig. 8. In the graphs, we



(a) Map, $20 \cdot 10^6$ elements

(b) Reduce, $90 \cdot 10^6$ elements

(c) MapReduce, $20 \cdot 10^6$ elements

Fig. 8 Execution time of repeated invocations of the same skeleton

can see the stability of the OpenMP, CUDA and hybrid backends. It is also apparent that the hybrid backend with the auto-tuning manages to find a good load balance and improves upon the execution time of the individual processing units. The performance of the StarPU backend is unstable, even though it manages to match the performance of the hybrid backend in some iterations. For the Reduce skeleton, both the hybrid and the StarPU backend have a hard time to improve the performance, as the skeleton works much better on the CPU compared to the GPU.

The execution time of the StarPU backend stabilizes somewhat with time, but it is still uneven after 20–30 repeated executions. This is likely due to the low number of tasks (manually found to be between 3 and 14) each skeleton instance had to be divided into for the best performance. This in turn is a result of the relatively small input size that was used in the evaluation. StarPU comes with a substantial overhead and might therefore be better suited for applications with even larger input sizes. The StarPU backend can also be of interest for special kinds of user functions with a very skewed workload, where adaptation is needed at runtime. But as these corner cases were not the target of the new auto-tuning implementation, no experiments were performed with such applications for this paper.

7 Related work

The skeleton programming library ***Marrow*** [16] consists of a mixed set of data and task parallel skeletons. The library uses nesting of skeletons to allow for more complex computations. Marrow has support for multi-GPU computations using OpenCL as well as hybrid execution on multi-core CPU, multi-GPU systems. The workload is statically distributed between the CPU threads and the GPUs, just like it is in SkePU. Marrow identifies load imbalances between the CPU and the GPUs and improves the models continuously to adapt to changes in the workload of the system. The partitioning between multiple GPUs is determined by their relative performance, as found by a benchmark suite.

Muesli (*Muenster skeleton library*) [7] is a C++ skeleton library built on top of OpenMP, CUDA and MPI, with support for multi-core CPUs, GPUs as well as clusters. Muesli implements both data and task parallel skeletons, but does not have support for as many data parallel skeletons with the same flexibility as in SkePU 2. Muesli has support for hybrid execution using a static approach [20], where a single value determines the partition ratio between the CPU and the GPUs, just as in SkePUs hybrid backend. The library also supports hybrid execution using multiple GPUs, with the assumption that they are of the same model. The library currently does not provide an automatic way of finding a good workload distribution which requires the user to manually specify it per skeleton instance.

Qilin [11] is a programming model for heterogeneous architectures, based on Intel's Threading Building Blocks (TBB) and CUDA. Qilin provides the user with a number of pre-defined operations, similar to the skeletons in SkePU 2. The library has support for hybrid execution by automatically splitting the work between a multi-core CPU and a single NVIDIA GPU. Just as in SkePU, one of the CPU threads is dedicated to communication with the GPU. The partitioning is based on

linear performance models created from training runs, much like SkePU's auto-tuner implementation.

The open source skeleton library ***SkelCL*** [17] is an extension to the OpenCL programming language. The library contains skeletons similar to the ones available in SkePU 2. SkelCL has support for dividing the workload between multiple GPUs, but does not support simultaneous hybrid CPU–GPU execution. As it is based on OpenCL and lacks a precompilation step, the user functions must be defined as string literals, lacking the compile time type checking available in SkePU 2.

Recent work in hybrid CPU–GPU execution of skeleton-like programming constructs include ***Lapedo*** [10], an extension of the *Skel* Erlang library for stream-based skeleton programming, specifically providing hybrid variants of the Farm and Cluster skeletons where the workload partitioning is tuned by models built through performance benchmarking; and Vilches' et al. [19] TBB-based heterogeneous parallel for template, which actively monitors the load balance and adjusts the partitioning during the execution of the for loop. Both approaches exclusively use OpenCL for GPU-based computation.

8 Conclusions and future work

In this paper, we have presented a new hybrid execution backend for the skeleton programming framework SkePU, while preserving the existing API. This has been done by letting the hybrid backend partition the workload of a skeleton into a CPU part and an accelerator part. By reusing the already existing accelerator backend implementations, code duplication was avoided and future optimizations to the CUDA and OpenCL backends will automatically also improve the performance of the hybrid backend. Although not yet shown in this paper, but planned for a future version, the implementation also works for any number of accelerators using either CUDA or OpenCL.

The contribution of this paper has several benefits: the performance of already written SkePU applications can be increased without any code modification by using the hybrid backend, programmers will get an easy-to-use tool with support for hybrid execution, and the SkePU framework can now fully utilize the potential of heterogeneous computer systems.

Future work includes adding auto-tuning also in the accelerator backend to improve the load balancing for multi-accelerator systems with mixed accelerator models. The hybrid performance of applications with multiple skeletons can be improved by taking the data dependencies between consecutive skeleton calls into consideration when predicting the partition ratios. We would also like to examine whether the auto-tuner could use static code analysis based on data from the precompiler, similar to the work of Grewe and O'Boyle [9].

Acknowledgements This work has been partly funded by EU H2020 project EXA2PRO (801015), by SeRC (http://www.e-science.se), and by the Swedish National Graduate School in Computer Science (CUGS). We want to thank Samuel Thibault for his help with the StarPU integration. We also thank the National Supercomputing Centre (NSC) and SNIC for access to their GPU-based computing resources.

References

1. Carpaye JMC, Roman J, Brenner P (2017) Design and analysis of a task-based parallelization over a runtime system of an explicit finite-volume CFD code with adaptive time stepping. J Comput Sci 28:439–454. https://doi.org/10.1016/j.jocs.2017.03.008
2. Cole MI (1989) Algorithmic skeletons: structured management of parallel computation. Pitman/MIT Press, Cambridge
3. Dastgeer U, Kessler C (2016) Smart containers and skeleton programming for GPU-based systems. Int J Parallel Program 44(3):506–530
4. Dastgeer U, Kessler C, Thibault S (2011) Flexible runtime support for efficient skeleton programming on heterogeneous GPU-based systems. In: Advances in parallel computing, volume 22: applications, tools and techniques on the road to exascale computing, vol 22, pp 159–166
5. Dastgeer U, Li L, Kessler C (2013) Adaptive implementation selection in the SkePU skeleton programming library. In: Wu C, Cohen A (eds) Advanced parallel processing technologies. Lecture notes in computer science, vol 8299. Springer, Berlin, Heidelberg
6. Enmyren J, Kessler CW (2010) SkePU: a multi-backend skeleton programming library for multi-GPU systems. In: Proceedings of the Fourth International Workshop on High-Level Parallel Programming and Applications. ACM, pp 5–14
7. Ernsting S, Kuchen H (2012) Algorithmic skeletons for multi-core, multi-GPU systems and clusters. Int J High Perform Comput Netw 7:129–138
8. Ernstsson A, Li L, Kessler C (2017) SkePU 2: flexible and type-safe skeleton programming for heterogeneous parallel systems. Int J Parallel Program 46(1):62–80. https://doi.org/10.1007/s10766-017-0490-5
9. Grewe D, O'Boyle MFP (2011) A static task partitioning approach for heterogeneous systems using openCL. In: Knoop J (ed) Compiler construction. Lecture notes in computer science, vol 6601. Springer, Berlin, Heidelberg
10. Janjic V, Brown C, Hammond K (2016) Lapedo: hybrid skeletons for programming heterogeneous multicore machines in Erlang. Parallel Comput Road Exascale 27:185
11. Luk CK, Hong S, Kim H (2009) Qilin: exploiting parallelism on heterogeneous multiprocessors with adaptive mapping. In: Proceedings of the 42nd Annual IEEE/ACM International Symposium on Microarchitecture. ACM, pp 45–55
12. Martínez V, Michéa D, Dupros F, Aumage O, Thibault S, Aochi H, Navaux POA (2015) Towards seismic wave modeling on heterogeneous many-core architectures using task-based runtime system. In: 2015 27th International Symposium on Computer Architecture and High Performance Computing (SBAC-PAD), pp 1–8
13. Öhberg T (2018) Auto-tuning hybrid CPU–GPU execution of algorithmic skeletons in SkePU. Master's thesis, Linköping University, Linköping, Sweden. LIU-IDA/LITH-EX-A-18/019-SE
14. Shen J, Varbanescu AL, Lu Y, Zou P, Sips H (2016) Workload partitioning for accelerating applications on heterogeneous platforms. IEEE Trans Parallel Distrib Syst 27(9):2766–2780
15. Sjöström O, Ko SH, Dastgeer U, Li L, Kessler C (2016) Portable parallelization of the EDGE CFD application for GPU-based systems using the SkePU skeleton programming library. In: Joubert GR, Leather H, Parsons M, Peters F, Sawyer M (eds) Advances in parallel computing, volume 27: parallel computing: on the road to exascale. Proceedings of the ParCo-2015 Conference, Edinburgh, UK, Sep. 2015. IOS Press, pp 135–144
16. Soldado F, Alexandre F, Paulino H (2016) Execution of compound multi-kernel OpenCL computations in multi-CPU/multi-GPU environments. Concurr Comput Pract Exp 28(3):768–787
17. Steuwer M, Gorlatch S (2013) SkelCL: Enhancing OpenCL for high-level programming of multi-GPU systems. In: International Conference on Parallel Computing Technologies. Springer, Berlin, pp 258–272
18. Universite de Bordeaux, CNRS, Inria: StarPU handbook ver. 1.2.4. Tech. rep. (2018)

19. Vilches A, Navarro A, Corbera F, Rodriguez A, Asenjo R (2017) Heterogeneous parallel for template based on TBB. In: Informal Proceedings; presented at HLPP 2017, Valladolid, Spain
20. Wrede F, Ernsting S (2018) Simultaneous CPU–GPU execution of data parallel algorithmic skeletons. Int J Parallel Program 46(1):42–61