Flexible Runtime Support for Efficient Skeleton Programming on Heterogeneous GPU-based Systems[1]

Usman DASTGEER [a], Christoph KESSLER [a] and Samuel THIBAULT [b]
[a] *PELAB, Dept. of Computer and Information Science, Linköping University.*
[b] *INRIA Bordeaux, LaBRI, University of Bordeaux.*

Abstract. SkePU is a skeleton programming framework for multicore CPU and multi-GPU systems. StarPU is a runtime system that provides dynamic scheduling and memory management support for heterogeneous, accelerator-based systems.

We have implemented support for StarPU as a possible backend for SkePU while keeping the generic SkePU interface intact. The mapping of a SkePU skeleton call to one or more StarPU tasks allows StarPU to exploit independence between different skeleton calls as well as within a single skeleton call. Support for different StarPU features, such as data partitioning and different scheduling policies (e.g. history based performance models) is implemented and discussed in this paper.

The integration proved beneficial for both StarPU and SkePU. StarPU got a high level interface to run data-parallel computations on it while SkePU has achieved dynamic scheduling and heterogeneous parallelism support. Several benchmarks including ODE solver, separable Gaussian blur filter, Successive Over-Relaxation (SOR) and Coulombic potential are implemented. Initial experiments show that we can even achieve super-linear speedups for realistic applications and can observe clear improvements in performance with the simultaneous use of both CPUs and GPU (heterogeneous execution).

Keywords. SkePU, StarPU, skeleton programming, dynamic scheduling, heterogeneous multicore architectures.

Introduction

Multi- and many-core architectures are increasingly becoming part of mainstream computing. There is a clear evidence that future architectures will be heterogeneous, containing specialized hardware, such as accelerator devices (e.g. GPGPUs) or integrated co-processors (e.g. Cell's SPUs) besides general purpose CPUs. These heterogeneous architectures provide desired power efficiency at the expense of increased programming complexity as they expose the programmer to different (and possibly low-level) programming models for different devices in the system. Besides the programming problem, the lack

[1]This is the author's version. The final publication is available at IOS press: Advances in Parallel Computing, Volume 22: Applications, Tools and Techniques on the Road to Exascale Computing
Usman Dastgeer, Christoph Kessler and Samuel Thibault. Flexible Runtime Support for Efficient Skeleton Programming on Heterogeneous GPU-based Systems. Proc. ParCo 2011: International Conference on Parallel Computing. Ghent, Belgium, 2011.

of a universal parallel programming model for these different architectures immediately leads to a portability problem.

Skeleton programming [5] is an approach that could solve the portability problem to a large degree. It requires the programmer to rewrite a program using so-called *skeletons*, pre-defined generic components derived from higher-order functions that can be parameterized in sequential problem-specific code, and for which efficient implementations for a given target platform may exist. A program gets parallelism and leverages other architectural features almost for free for skeleton-expressed computations as skeleton instances can easily be expanded or bound to equivalent expert-written efficient target code that can encapsulate low-level details such as managing parallelism, load balancing, communication, utilization of SIMD instructions etc. SkePU [1,2] is such a skeleton programming framework for modern heterogeneous systems that contain multiple CPU cores as well as some specialized accelerators (GPUs).

Besides the programming and portability problem with these heterogeneous architectures, there is a problem with performance. These heterogeneous architectures are composed of multicore general-purpose processors and one or more accelerators with separate memory address spaces. One common way to program these architectures is to offload computation-intensive parts of the application to the accelerators. However, to fully tap into the potential of these machines, simple offloading techniques are not sufficient. Rather, the challenge is to build systems which can spread an application across the entire machine by scheduling tasks dynamically over multiple different devices considering the current resource usage and suitability of a task to a specific device. StarPU is a runtime system for modern heterogeneous architectures that provides dynamic scheduling capabilities and can use runtime feedback and data locality information to guide the scheduling decisions at runtime.

Need for runtime support: SkePU provides several implementations for each skeleton type including an OpenMP implementation for multi-core CPUs and CUDA and OpenCL[2] implementations for single as well as multiple GPUs. In a platform containing both CPUs and GPUs, several *implementation variants* for a given skeleton call are applicable and an optimal variant should be selected for skeleton call. Selecting an offline optimal variant for an independent SkePU skeleton call is already implemented in SkePU itself using offline measurements and machine learning [2]. However, an application composed of multiple skeleton calls will most likely have data flow based constraints between different skeleton calls. Tuning such a composed application requires runtime information about resource consumption and data locality. Furthermore, the current SkePU implementation has no efficient means to use multiple kinds of resources (CPUs and GPUs) simultaneously for a skeleton execution.

In this paper, we present our work on integrating the SkePU skeleton library with the StarPU runtime system to leverage the capabilities of the runtime system that are missing in SkePU. We will briefly introduce SkePU and StarPU, followed by the translation details. Section 4 will discuss several important aspects of performance with several real applications and show how SkePU actually benefits from the integration in terms of performance. Finally, we will conclude and present future work possibilities.

Related work: Besides related work to SkePU (see [5]) and StarPU (see [3]), some work is done on static scheduling of OpenCL code on heterogeneous platforms in [6]. In

[2]The OpenCL implementation can be used with CPUs as well but it is optimized for GPUs in our case.

[7], load balancing and work distribution strategies for multi-GPUs systems are considered, however it ignores usage of CPUs altogether. To the best of our knowledge, no prior work is reported on dynamic scheduling support for a generic skeleton programming framework on modern heterogeneous multicore systems.

1. SkePU

SkePU [1,2] is a C++ based skeleton programming library for single- and multi-GPU systems supporting multiple back-ends (CUDA, OpenCL, OpenMP and sequential C). It currently implements several data-parallel skeletons including map, reduce, mapreduce, map-with-overlap, map-array, and scan. All skeleton calls accept SkePU generic containers (Vector, Matrix) as arguments.

The SkePU containers implicitly manage the data transfers between host and GPU memory and keep track of multiple copies of the data residing on different memory units. The containers use lazy memory copying to optimize memory transfers for skeleton operand data at GPU execution. *Lazy memory copying* delays data transfer back to main memory for results of computations done on the GPU until it is actually accessed in the main memory (for example through the [] operator). Lazy memory copying is of great use if several skeleton calls are made with the same container, one after the other, with no modifications of the container data by the host in between. A more detailed description of the SkePU library can be found in [1].

2. StarPU

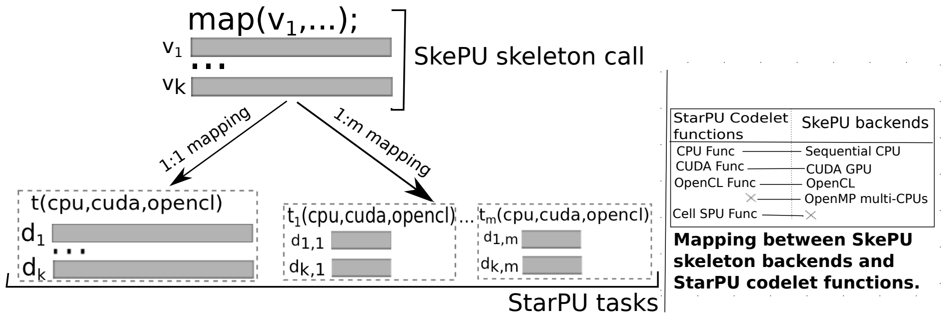
StarPU [3,4] is a C based unified runtime system for heterogeneous multicore platforms with generic scheduling facilities. Three main components of StarPU are task and codelet abstraction, data management, and dynamic scheduling framework.

StarPU task-model: StarPU uses the concept of *codelet*, a C structure containing different implementations of the same functionality for different computation units (e.g. CPU and GPU). A StarPU task is then an instance of a codelet applied to some data. The programmer has to explicitly submit all tasks and register all the input and output data for all tasks. The submission of tasks is asynchronous and termination is signaled through a callback[3]. This lets the application submit several tasks, including tasks which depend on others. Dependencies between different tasks can be found either by StarPU implicitly, by considering data dependencies (RAW/WAR/WAW) between submitted tasks, and/or can be explicitly specified by the programmer using integers called *tags*.

Data management: StarPU provides a virtual shared memory subsystem and keeps track of data across different memory units in the machine by implementing a weak consistency model using the MSI coherency protocol. This allows StarPU to avoid unnecessary data movement when possible. Moreover, the runtime can estimate data transfer cost and can do prefetching to optimize the data transfers. The runtime can also use this information to make better scheduling decisions (e.g. data aware scheduling policies).

StarPU defines the concept of *filter* to partition data logically into smaller chunks (block- or tile-wise) to suit the application needs. For example, the filter for 1D vector

[3]Task execution can be made synchronous as well, by setting the `synchronous` flag for a StarPU task. This makes the task submission call blocking and returns control after the submitted task finishes its execution.



(a) Mapping between SkePU skeleton calls and StarPU tasks.

(b)

Figure 1. (a) shows how a SkePU skeleton call is mapped to one or more StarPU tasks and (b) shows how different backend implementation choices are mapped between SkePU and StarPU.

data is *block filter* which divides the vector into equal-size chunks, while for a dense 2D matrix, the filters include partitioning the matrix into horizontal and/or vertical blocks. Multiple filters could be applied in a recursive manner to partition data in any nested order (e.g. dividing a matrix into 3×3 blocks by applying both horizontal and vertical filters).

Dynamic Scheduling: Mapping of tasks to different execution units is provided using dynamic scheduling. There are several built-in scheduling strategies including greedy (priority, no-priority), work-stealing, and several performance-model aware policies (e.g. based on historical execution records).

3. Integration

StarPU is implemented as another possible backend to the SkePU skeleton calls. The main objective of the integration was to keep the generic, high-level interface of SkePU intact while leveraging the full capabilities of StarPU underneath that interface. Some minimal changes in the skeleton interfaces were required to allow for the desired flexibility at the programmer level, but these interface extensions are kept minimal (and optional e.g. by using default parameter values) to most extent, allowing previously written SkePU programs to smoothly run with this new backend in many situations while requiring very small syntactic changes in other cases.

Containers: The data management feature (including lazy memory copying) in the SkePU containers was overlapping with StarPU data management capabilities. To enable the integration, that data management part of the SkePU containers is disabled. In essence, the containers are modified to automatically register/unregister data as data handlers to StarPU for skeleton calls. This means, that while using StarPU, containers act as smart wrappers and delegate actual memory management to the StarPU runtime system. The interface of the containers is not modified in the process. Furthermore, a custom memory allocator is written and configured with the SkePU containers to allow page-locked memory allocation/deallocation for data used with the CUDA calls[4].

[4]In CUDA with GT200 or newer GPUs, memory allocated through `cudaHostAlloc` becomes directly accessible asynchronously from the GPU via DMA and is referred to as page locked memory.

Abstraction gap: SkePU heavily relies on features of C++ (e.g. templates, functors, classes and inheritance) while StarPU is a pure C based runtime system (e.g. using function and raw pointers). The integration is achieved by wrapping the skeleton code in static member functions which can be passed to StarPU as a normal C function pointer. Furthermore, the possibility of asynchronous SkePU skeleton executions resulted in severe concurrency issues (e.g. race conditions, data consistency problems). Certain programming techniques such as *data-replication* are used to resolve these issues while avoiding mutual exclusion.

Mapping SkePU skeletons to StarPU tasks: In StarPU, the unit of execution is a task. An application needs to explicitly create and submit tasks to the runtime system. However, in our case, the creation and submission of tasks is transparently provided behind the skeleton interface. The mapping technique is illustrated in Figure 1 and explained below.

A SkePU skeleton call S with k operands v_i, where $1 \leq i \leq k$, each of size N, can be translated into one or more StarPU tasks. In direct (1:1) mapping, it translates to 1 StarPU task t with k operands d_i, each of size N and $d_i = v_i \,\forall i$. In 1:m mapping, m StarPU tasks t_j where $1 \leq j \leq m$ are generated, each taking k operands d_{ij} where $1 \leq i \leq k$ and $1 \leq j \leq m$, each of size $N^{'}$. In our case, $N^{'} \leq N/m$ as we divide a skeleton call into equally partitioned tasks based on operand data, considering the fact that computational complexity of data-parallel skeletons is usually the same for individual elements. For the MapArray skeleton, partitioning works as described above except for the first argument which is not partitioned[5].

Data Partitioning: Partitioning support is implemented using StarPU filters in many existing skeletons by adding an optional last argument to the skeleton call, specifying the number of desired partitions for that skeleton call (defaults to 1, no-partition). The layout of partitioning depends upon the skeleton type (e.g. partition a Matrix horizontally and vertically for 2D MapOverlap row-wise and column-wise respectively).

Scheduling support: StarPU pre-defines multiple scheduling policies, including some based on performance models. The performance models could be either execution history based or some application specific parametric models (e.g. $an^3 + bn^2 + cn + d$). With the history based performance model, StarPU keeps track of the execution time of a task from its actual executions to guide the scheduling decisions in future [4]. We have configured all skeletons to use the history based performance model. This could be enabled, just by defining the USE_HISTORY_PERFORMANCE_MODEL flag in the application. The actual scheduling policy can later be altered at the execution time using the STARPU_SCHED environment variable.

4. Evaluation

All experiments are carried out on a GPU server with dual-quadcore Intel(R) Xeon (R) E5520 server clocked at 2.27 GHz with 2 NVIDIA Tesla M2050 GPUs.

Data Partitioning and locality on CPUs: The efficiency of the data partitioning (task-parallelism) approach in comparison to OpenMP is shown in Figure 2 for applying Gaussian blur column wise, using a MapOverlap skeleton with 2D SkePU Matrix.

[5]This is due to the semantics of the MapArray skeleton where for each element of the second operand, *all* elements of the first operand should be available.

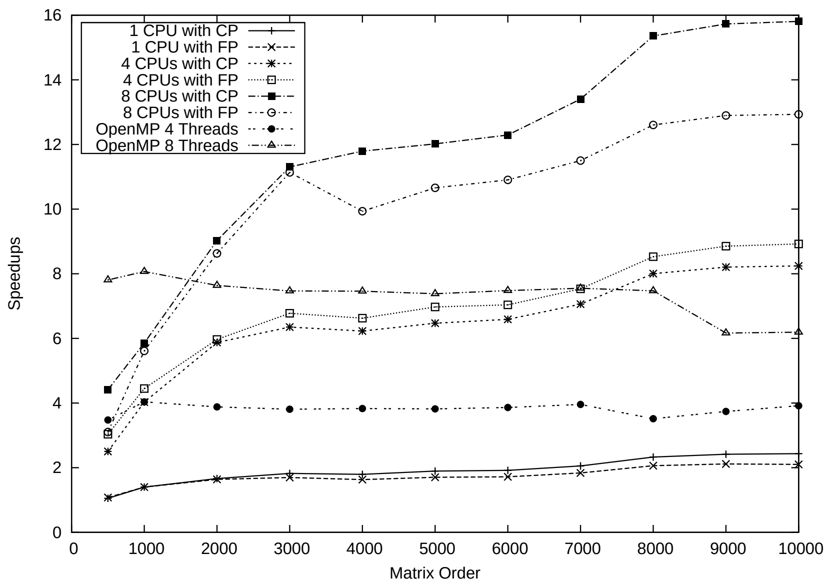
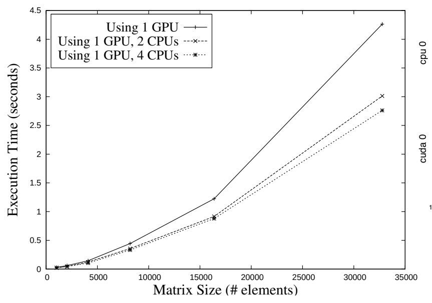


Figure 2. Applying Gaussian blur column-wise using MapOverlap column-wise skeleton on one or more CPUs. Results represent OpenMP as well as both fine partitioning (FP, 20 columns per task) and coarse partitioning (CP, 100 columns per task) version.

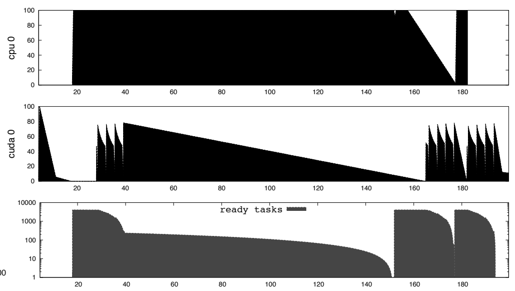
The column-wise Gaussian blur is chosen as it accesses matrices (both input and output) column-wise, which is cache inefficient for regular C matrices that are stored row-wise. The OpenMP implementation is optimized to distribute work column-wise, using static scheduling policy, as the dynamic policy performs poorly in this regular work load case. The partitioning using filters is also done vertically for column-wise MapOverlap operation and multiple independent sub-tasks are created. The baseline version is the sequential version, without any partitioning, executed on a single CPU. As shown in the figure, we are able to achieve super-linear speedups while using multiple CPUs and also with a single CPU by partitioning a data-parallel task into sub-tasks, thanks to improved cache usage. The OpenMP versions are rewritten from a sequential version to divide work column-wise and achieve linear speedups even for smaller matrix sizes. However, the partitioning based approach was using the same sequential code (baseline), written for a single CPU and achieved better speedups than OpenMP without any significant tuning effort (no tuning of partitioning granularity).

Heterogeneous execution: A Coulombic potential grid calculation micro-benchmark is implemented using Map and MapArray skeletons. Figure 3(a) shows the improvements from using multiple resources present in the system. The usage of CPUs along the GPU yield better performance (less execution time) even for an application that is known for its suitability to GPU execution [6].

Data-locality aware scheduling: Figure 3(b) shows how a data-aware scheduling policy improves by learning at runtime. The data-aware scheduling policy considers the current data location and expected data transfer costs between different memory locations while scheduling a task for execution [4]. The estimate of a task execution time based on earlier executions is also used for scheduling. This is shown in Figure 3(b), while using 1 CPU and 1 GPU for three consecutive Coulombic calculations using data-

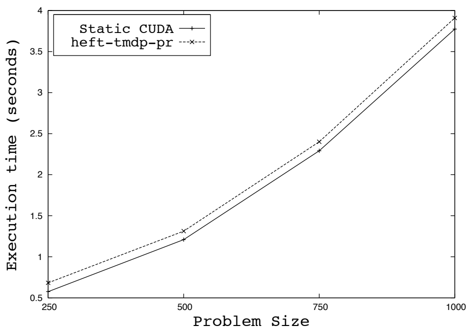


(a)



(b)

Figure 3. Coulombic potential grid execution a) on a heterogeneous platform (CPUs and GPU) for different matrix sizes and b) with a data aware scheduling policy for $24K \times 24K$ matrix, with 3 successive executions.



(a) LibSolve execution - static and dynamic scheduling

Scheduling Policy	Avg Speedup
heft-tm (HEFT based on Task duration Models)	1.38055
heft-tm-pr (heft-tm with data PRefetch)	1.38422
heft-tmdp (heft-tm with remote Data Penalty)	2.07955
heft-tmdp-pr (heft-tmdp with data PRefetch)	2.13410

(b) Iterative SOR execution.

Figure 4. Performance implications of StarPU dynamic scheduling policies.

aware scheduling policy. In the first execution, the data was not available on the GPU and estimates from earlier executions were also not available yet, hence work is divided across CPU and GPU in some greedy fashion. However, as time passes, more input data becomes available on the GPU and execution time estimates become available, resulting in more performance-aware decisions in the later part of the execution. The scheduling decisions improved significantly over successive executions as the execution time reduced by almost 7 times in the third execution in comparison to the first execution.

Performance-model based scheduling policies: There are several performance-aware scheduling policies available in StarPU that try to minimize the make-span of a program execution on a heterogeneous platform (known as **H**eterogeneous **E**arliest **F**inish **T**ime (HEFT) scheduling policies). Figure 4(b) shows execution of Iterative SOR with such performance-aware scheduling policies available in StarPU. Average speedups are calculated over different matrix sizes with respect to static scheduling (CUDA execution) policy. The result shows that usage of scheduling policies with data prefetching support yield significant performance gains.

Static scheduling: The performance gains by using dynamic scheduling capabilities of StarPU in a heterogeneous execution platform are shown earlier for different applications. These performance gains come from the task-level parallelism which depends on inter(or intra)-skeleton independence (cf. 1:1 and 1:m mapping in Figure 1). Now, we consider an extreme example where static scheduling on a powerful CUDA GPU supersedes any known dynamic scheduling configuration using CPUs in conjunction. An application with strong data dependency across different skeleton calls and small computational complexity of each skeleton call can limit performance opportunities for the runtime system to exploit. The ODE solver is such an application, containing lots of repetitive, simple mathematical operations represented as skeleton calls. The tight data dependency between these skeleton calls allows almost no inter-skeleton parallelism. Furthermore, as tasks are computationally small, the overhead of creation of subtasks (including data partitioning) to exploit intra-task parallelism limits its performance gains. Although static scheduling proved better for this application as shown in Figure 4(a), a performance-aware dynamic scheduling comes quite close to it. This shows that even for such an extreme scenario, using dynamic scheduling comes quite close to static scheduling including all the overhead.

5. Conclusion and Future work

The SkePU translation to StarPU gives multifold benefits. Dynamic scheduling support combined with performance models can generate more informed scheduling decisions. Furthermore, it allows usage of multiple backends in parallel for a single skeleton call. It can also achieve better cache behavior for otherwise non-cache friendly skeleton operations. The performance benefits depend upon parallelism which can be exploited both inside a single skeleton call as well as between independent skeleton calls. This allows to exploit it for many diverse applications. We have shown that a performance-aware dynamic scheduling policy can perform well even for extreme scenarios where static scheduling on a single backend proves better than exploitation of multiple resources available in the system.

This work can be extended in many ways. New skeleton types can be implemented (e.g. farm, divide and conquer, pipelining). New dynamic scheduling policies can be implemented that can consider other factors while making the scheduling decision (e.g. limited lookahead information about repetitive execution). Finally, we plan a public release of this work like original SkePU library (`http://www.ida.liu.se/~chrke/skepu`).

Acknowledgements: This work was funded by EU FP7, project PEPPHER, grant #248481 (www.peppher.eu).

References

[1] J. Enmyren and C. W. Kessler. SkePU: A Multi-Backend Skeleton Programming Library for Multi-GPU Systems. Proc. 4th Int. Workshop on High-Level Parallel Programming and Applications (HLPP-2010), Baltimore, USA, Sep. 2010.

[2] U. Dastgeer, J. Enmyren and C. W. Kessler. Auto-tuning SkePU: A Multi-Backend Skeleton Programming Framework for Multi-GPU Systems. 4th Int. Workshop on Multicore Software Engineering (IWMSE11), Waikiki, Honolulu, HI, USA, May. 2011.

[3] Cédric Augonnet, Samuel Thibault, Raymond Namyst and Pierre-André Wacrenier. StarPU: A Unified Platform for Task Scheduling on Heterogeneous Multicore Architectures. In Concurrency and Computation: Practice and Experience, Special Issue: Euro-Par 2009, vol. 23(2), pages 187–198, Feb. 2011.

[4] Cédric Augonnet, Jérôme Clet-Ortega, Samuel Thibault and Raymond Namyst. Data-Aware Task Scheduling on Multi-Accelerator based Platforms. Proc. 16th Int. Conference on Parallel and Distributed Systems (ICPADS), Shanghai, China, December 2010.

[5] Christoph Kessler, Sergei Gorlatch, Johan Enmyren, Usman Dastgeer, Michel Steuwer, and Philipp Kegel. Skeleton Programming for Portable Many-Core Computing. In S. Pllana, and F. Xhafa, eds., Programming multi-core and many-core computing systems, 2011.

[6] Dominik Grewe and Michael F.P. O'Boyle. A Static Task Partitioning Approach for Heterogeneous Systems Using OpenCL. In International Conference on Compiler Construction, March 2011.

[7] L. Chen, O. Villa, S. Krishnamoorthy, and G.R. Gao. *Dynamic load balancing on single- and multi-GPU systems*. Int. Parallel & Distributed Processing Symposium (IPDPS), pp.1-12, 2010.