Towards a Tunable Multi-Backend Skeleton Programming Framework for Multi-GPU Systems

Johan Enmyren
PELAB
Dept. of Computer and Information Science
Linköping University
x10johen@ida.liu.se

Usman Dastgeer
PELAB
Dept. of Computer and Information Science
Linköping University
usmda@ida.liu.se

Christoph W. Kessler
PELAB
Dept. of Computer and Information Science
Linköping University
chrke@ida.liu.se

ABSTRACT

SkePU is a C++ template library that provides a simple and unified interface for specifying data-parallel computations with the help of skeletons on GPUs using CUDA and OpenCL. The interface is also general enough to support other architectures, and SkePU implements both a sequential CPU and a parallel OpenMP backend. It also supports multi-GPU systems. Currently available skeletons in SkePU include map, reduce, mapreduce, map-with-overlap, map-array, and scan. The performance of SkePU generated code is comparable to that of hand-written code, even for more complex applications such as ODE solving.

In this paper, we describe how to make SkePU tunable, by adding the mechanism of *execution plans* that can configure a skeleton so that, at run time, the predicted best suitable resource and platform is chosen automatically, depending on operand data sizes. We also discuss how the approach can be extended to provide a fully auto-tunable skeleton programming system, which is a work in progress.

Categories and Subject Descriptors

C.1.4 [**Processor Architectures**]: Parallel Architectures; D.1.3 [**Programming Techniques**]: Concurrent Programming—*Parallel Programming*

General Terms

Algorithms, Languages, Performance

Keywords

Skeleton Programming, GPU, CUDA, OpenCL, Data Parallelism, Auto-tuning

1. INTRODUCTION

The general trend towards multi- and many-core based systems constitutes a disruptive change in the fundamental programming model in mainstream computing and requires rewriting of sequential application programs into parallel form to turn the steadily increasing number of cores into performance. Worse yet, there is a number of very different architectural paradigms such as homogeneous SMP-like multicores, heterogeneous multicores like Cell Broadband Engine, or hybrid CPU/GPU systems, sometimes with a very high complexity of programming such systems efficiently. Moreover, we observe a quick evolution process on the hardware side, pushing new architecture generations and variations on the market with short time intervals. The lack of a universal parallel programming model immediately leads to a portability problem.

Skeleton programming [2, 11] 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 implementation for a given target platform may exist. Skeleton programming constrains the programmer to using only the given set of skeletons for the code that is to be parallelized or ported automatically—computations that do not fit any predefined skeleton (combination) still have to be rewritten manually. In turn, parallelism and leveraging other architectural features comes almost for free for skeleton-expressed computations, as skeleton instances can easily be expanded or bound to equivalent expert-written efficient target code that encapsulates all low-level platform-specific details such as managing parallelism, load balancing, communication, utilization of SIMD instructions etc.

In previous work [4], we described the design and implementation of *SkePU*, a new C++ based skeleton programming library for single- and multi-GPU systems that supports multiple back-ends, namely CUDA and OpenCL for GPUs and OpenMP for multi-core CPUs, but also sequential C for single-core execution. In order to optimize memory transfers for skeleton operand data at GPU execution, SkePU implements a lazy copying technique in the vector data container that is used in SkePU to represent array operands. Our experimental evaluation showed that code written in terms of SkePU skeletons is portable across all platforms with implemented back-ends, and achieves performance close to hand-written code.

In particular, the benchmarks in [4] showed significant speedup for GPU execution compared to CPU execution when run with larger data sizes on a GPU compared to a fast

MCC'10, November 18-19, 2010, Göteborg, Sweden

CPU core, while CPU execution, with or without OpenMP thread-level parallelism is faster for smaller problem sizes. However, the transition points when to switch from an implementation for one CPU to an OpenMP parallelization or to code for a single GPU or for multiple GPUs are strongly dependent on the characteristics of the target system.

This led to the idea to provide a mechanism for (semi-)automatic adaptation at run-time to let SkePU select the best implementation variant depending on the actual problem size (and possibly further problem parameters).

In this paper, we describe how SkePU can be made tunable by adding the concept of *execution plans*. An execution plan sets the intervals of problem size (and possibly further parameters) within which a certain back-end with certain parameters (grid-size and block-size for GPUs, 'number of threads' for OpenMP) should be chosen. In a longer term perspective, execution plans will be computed automatically by machine learning techniques from training data generated by micro-benchmarking. The machine learning approach for automatically generating execution plans is demonstrated for very simple cases in this paper, but will be extended in future work to tuning for backend-selection and composition of skeletons.

In first results, we demonstrate the potential for tunability, both in terms of choosing parameters for a backend and selecting which backend implementation to use. We discuss composition of skeletons and how auto-tuning can be applied to a bigger application such as an ODE solver. Furthermore, it shows the resulting potential for performance portability at transition from one to another target architecture, for the Reduce skeleton in SkePU.

```
BINARY_FUNC(plus, double, a, b,
    return a+b;
)

// expands to:

struct plus
{
    skepu::FuncType funcType;
    std::string func_CL;
    std::string funcName_CL;
    std::string datatype_CL;
    plus()
    {
        funcType = skepu::BINARY;
        funcName_CL.append("plus");
        datatype_CL.append("double");
        func_CL.append(
        "double plus(double a, double b)\n"
        "{\n"
        "   return a+b;\n"
        "}\n");
    }
    double CPU(double a, double b)
    {
        return a+b;
    }
    __device__ double CU(double a, double b)
    {
        return a+b;
    }
};
```

Figure 1: User function, macro expansion.

2. SKEPU

SkePU is a C++ template library designed to make parallel programming easier with the use of higher-order functions, skeletons. SkePU is geared towards GPU-based and hybrid systems, using CUDA and/or OpenCL as backend for GPUs. A large portion of the library therefore consists of GPU memory management, kernels and, in the case of OpenCL, code generation and compilation. The interface is however fairly general and does not make the library bound to only GPUs. This can also be seen in SkePU as there is a sequential CPU and an OpenMP based implementation of all the skeletons. Modifications of the source code are not necessary since the interface is the same for all implementation variants.

In addition to the skeletal functions, SkePU also includes one container which must be used when doing computations with the skeletons. It is a vector/array type, designed after the STL container `vector`. Its implementation uses the STL `vector` internally and its interface is mostly compatible with STL `vector`.

The SkePU `vector` hides GPU memory management and also uses lazy memory copying to avoid unnecessary memory transfer operations between main memory and device memory. The SkePU `vector` keeps track of which parts of it are currently allocated and uploaded to the GPU. If a computation is done, changing the vector in the GPU memory, it is not directly transferred back to the host memory. Instead, the vector waits until an element is accessed on the host side before any copying is done (for example through the [] operator); this lazy memory copying is of great use if several skeletons are called one after the other, with no modifications of the vector by the host in between. In that case, the vectors are kept on the device (GPU) through all the computations, which greatly improves performance. Most of the memory copying is done implicitly but the vector also contains a `flush` operation which updates the vector from the device and deallocates its memory.

Specification of User Functions.

In order to provide a simple way of defining functions that can be used with the skeletons regardless of the target architecture, SkePU provides a *macro language* where preprocessor macros expand, depending on the target selection, to the right kind of structure that constitutes the function. The SkePU user functions generated from a macro based specification are basically a `struct` with member functions for CUDA and CPU, and strings for OpenCL. Figure 1 shows one of the macros and its expansion.

Skeleton Functions.

Customized instantiations of skeleton functions are created by expanding a class template of the generic skeleton function parameterized in the macro-based user function specification. These generated skeleton functions in SkePU are then represented by (singleton) objects. By overloading `operator()` they can be made behave similarly to regular functions. All skeletons contain member functions representing each of the different implementations, CUDA, OpenCL, OpenMP and CPU. The member functions are called CU, CL, OMP and CPU respectively. If the skeleton is called with `operator()`, the library decides which one to use depending on what is available. In the OpenCL case, the

```
#include <iostream>

#include "skepu/vector.h"
#include "skepu/mapreduce.h"

BINARY_FUNC(plus, double, a, b,
    return a+b;
)

BINARY_FUNC(mult, double, a, b,
    return a*b;
)

int main()
{
    skepu::MapReduce<mult, plus> dotProduct(new mult,
                                            new plus);
    skepu::Vector<double> v0(1000,2);
    skepu::Vector<double> v1(1000,2);

    double r = dotProduct(v0,v1);

    std::cout<<"Result: " <<r <<"\n";
    return 0;
}

// Output
// Result: 4000
```

Figure 2: Dot product with MapReduce in SkePU.

skeleton objects also contain the necessary code generation and compilation procedures. When a skeleton is instantiated, it creates an environment to execute in, containing all available OpenCL or CUDA devices in the system. This environment is created as a singleton so that it is shared among all skeletons in the program.

Currently, SkePU implements the following skeletons:

In the *Map* skeleton, every element in the result vector r is a function f of the corresponding elements in one or more input vectors $v_1 \dots v_k$.

The *Reduce* skeleton computes a scalar result by applying a commutative associative binary operator $\oplus$ to accumulate all elements in the input vector.

Given a binary associative function $\oplus$, the *Scan*$< \oplus >$ skeleton computes the prefix-$\oplus$ vector of its input vector, such as the prefix sums vector where $\oplus$ is standard addition. Scan is an important basic building block of many scalable parallel algorithms, such as parallel integer sorting.

MapReduce is a combination of Map and Reduce: It produces the same result as if one would first Map one or more vectors to an intermediate result vector and then do a reduction on that result. It is provided since it combines the mapping and reduction in the same computation kernel and therefore avoids some synchronization, which speeds up the calculation. In Figure 2 a dot product of two vectors is expressed as a MapReduce skeleton with `mult` and `plus` as provided user functions.

Map-overlap is similar to a Map, but each element $r[i]$ of the result vector is a function of several adjacent elements of one input vector that reside at a certain constant maximum distance d from i in the input vector, where d is a skeleton template parameter. Convolution is an example of a calculation that fits into this pattern.

MapArray is another variant of Map where each element of the result, $r[i]$, is a function of the corresponding element of one of the input vectors, $v_1[i]$, and any number of elements from the other input vector v_2.

```
skepu::Reduce<plus> globalSum(new plus);
skepu::ExecPlan plan;
plan.add(1,3500, skepu::CPU_BACKEND);
plan.add(3501,3200000, skepu::OMP_BACKEND, 8);
plan.add(3200001,5400000, skepu::CL_BACKEND, 65536, 128);
plan.add(5400001,MAX_INT, skepu::CLM_BACKEND, 65536, 128);
globalSum.setExecPlan(plan);
```

Figure 3: Defining an executon plan and applying it to a Reduce skeleton.

Multi-GPU support.

SkePU has support for carrying out computations with the help of several GPUs on a data-parallel level. It utilizes the different GPUs by dividing the input vectors equally amongst them and doing the calculations in parallel on the devices. Here CUDA and OpenCL differ a lot. In OpenCL, one CPU thread can control several GPUs, by switching queues. In CUDA, or to be more precise, in the CUDA runtime system which SkePU is based on, this is not possible. Instead, each CPU thread is bound to one device. To make multi-GPU computation possible, several host threads must then be created. This is done in SkePU, but in the current version new threads are started for each skeleton call and bound to devices; this binding can take a lot of time, and hence the multi-GPU support in SkePU with CUDA is not very efficient. With OpenCL however, it works much better.

Execution Plan.

A new feature added to SkePU, which is the focus of this paper, is the notion of an *execution plan*. It is an object containing different parameters which will affect the execution time of a skeleton. The parameters include a list of vector sizes and adjoining backends which is used to decide which backend to use at certain input sizes. Other parameters are group and grid size for the GPU backends.

All skeletons include an execution plan and also support for changing it manually. A default exectution plan is created at skeleton instantiation time containing default parameters chosen by the implementation. Figure 3 shows how to define an execution plan and apply it to a skeleton.

3. SKEPU TUNABILITY

Tuning opportunities for SkePU can be discussed at the following three levels.

3.1 Tuning parameters for a specific backend

For GPU implementations of skeletons, varying grid-size and block-size have considerable effect on the overall skeleton performance. The shared memory size is often a dependent parameter calculated from the corresponding grid-size and block-size and cannot be varied, for keeping correctness of the implementation. For multi-CPU (OpenMP) implementations of different skeletons, the number of OpenMP threads can have a great impact. Varying OpenMP scheduling (e.g. different chunksizes) can be another tunable parameter which can be considered in future.

Tuning these parameters for each implementation and for different problem sizes, we often get different parameter val-

ues for different problem sizes. However, in certain skeletons, differences between optimal and second or third best can be quite small and hence it could be decided not to switch quite often back and forth to the new parameter values. This could be handy for two reasons. First, each new configuration will result in an entry in the corresponding execution plan which increases the runtime overhead for the look-up. Second, for the OpenMP backend as witnessed during the experiments, switching the number of threads frequently in an OpenMP program can have a negative impact on the overall performance.

3.2 Tuning selection of backend

Besides determining optimal parameters for individual backend implementations, we need to choose between different implementations for a single skeleton call. For instance, for very small problem sizes, the sequential CPU implementation could be efficient and for slightly larger problem sizes, the OpenMP implementation could take less execution time. Similarly, for repetitive executions with data residing on device, GPU implementations often outperform (Multi-)CPU-based implementations.

3.3 Tuning composition of different skeletons

Composition of different skeletons in a single program is often what is required in real-world applications. This composition poses extra challenges on the tuning-framework as tuning individual skeleton calls independently without program composition knowledge will most likely yield poor performance.

One real-world application ported to SkePU is the Runge Kutta ODE Solver [4]. It uses twenty different skeletons calls to Map, Mapreduce, Reduce and MapArray skeletons often nested in loops. Listing 1 shows a synthetic example that is extracted from the ODE solver application which was earlier ported to SkePU [4]. Tuning such a composed application that has different types of skeletons and also different variations of the same skeleton (e.g. different user functions) requires flexibility in the tuning framework. For example, the decision made for the `s1` call in listing 1 can affect the optimal choices for later skeleton calls. Likewise, knowledge about susequent skeleton calls along with their execution frequency can affect optimality of decision at the `s1` call. To provide such functionality with current settings, the execution plan will need to be altered for each skeleton invocation rather than for each skeleton definition, which can result in significant overhead. In the results section, we will show the ODE solver application with empirical-tuning considering composition of skeletons.

4. FIRST RESULTS

Results for SkePU with several benchmarks including a Runge-Kutta ODE solver can be found in [4], showing that performance close to hand-tuned code can be achieved. Here we focus on the new tunability feature added to SkePU.

We consider two different GPU-based target architectures:

1. *Target architecture 1*: Dual-quadcore Intel(R) Xeon (R) E5520 server clocked at 2.27 GHz with 2 NVIDIA GT200 (Tesla C1060) GPUs.

2. *Target architecture 2*: Intel Core 2 Duo E6600 with one GeForce GTS250 GPU.

```
...
s1(v1, v2, out1); // MapArray skeleton call
while(...)
{
  s2(out1, v1); // Map call 1
  for(...)
  {
    s3(v1, v3); // Map call 2
  }
  res = s4(v2) // Reduce call

  s5(out1,v5, out2); // Map call 3
}
res2 = s6(out1, out2); //MapReduce call
...
```

Listing 1: Simplified example code showing composition of different skeletons

4.1 Tuning Parameters

Figure 4 shows that even for very simple skeletons (element-wise map), different combinations of grid-sizes and block-sizes can have a profound impact on the overall performance. In this case we have shown three hardcoded configurations of grid-sizes and block-sizes for the OpenCL backend and one tuned configuration for the same backend. The tuned configuration is obtained by exercising different combinations of the grid-sizes and block-sizes for different problem sizes using a genetic algorithm and choosing better (with respect to execution time) for each case. The set of choices is too large to hand-prune and often can be non-intuitive. In the following, we show a small part of the execution plan which is automatically generated by the tuning algorithm, where columns denote `lower input limit`, `upper input limit`, `backend`, `block-size`, and `grid-size` respectively:

```
1       --- 750000  CL_BACKEND 32  32768
750001  --- 1250000 CL_BACKEND 32  512
1250001 --- 2250000 CL_BACKEND 128 2048
2250001 --- 3750000 CL_BACKEND 32  2048
3750001 --- 4250000 CL_BACKEND 32  16384
4250001 --- 5250000 CL_BACKEND 128 32768
5250001 --- 5750000 CL_BACKEND 128 16384
5750001 --- 6250000 CL_BACKEND 128 32768
6250001 --- 7250000 CL_BACKEND 512 16384
7250001 --- 8250000 CL_BACKEND 256 16384
...
```

To reduce entries in the execution plan and consequently the runtime lookup overhead, we use a threshold value to keep a previous configuration entry for new problem sizes if it lies within the desired threshold limit. Choosing a threshold value is often a tradeoff between the runtime overhead and the precision of tuning process.

4.2 Performance portability for different backends

We consider a simple vector sum computation, expressed by a single reduce skeleton instance, which is repeated 100 times for the measurements.

For the target architecture 1, we construct manually an execution plan with empirically found intervals:

```
skepu::Reduce<plus> globalSum(new plus);

skepu::ExecPlan plan;
```

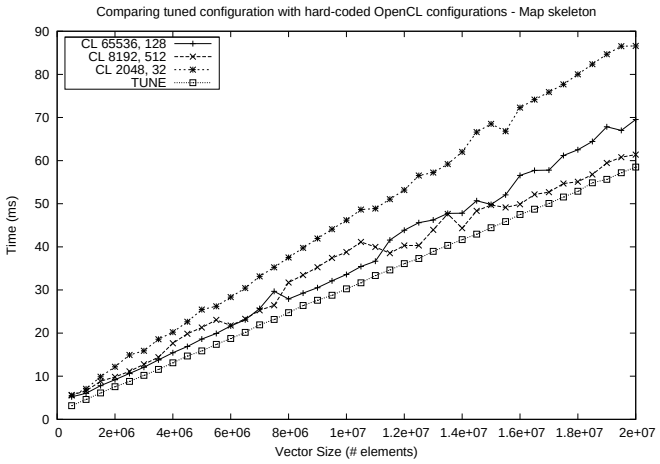

Figure 4: Tuning grid-size and block-size for OpenCL backend for Map skeleton on target architecture 1. Numbers in titles show 'grid-size, block-size' combinations used for execution.

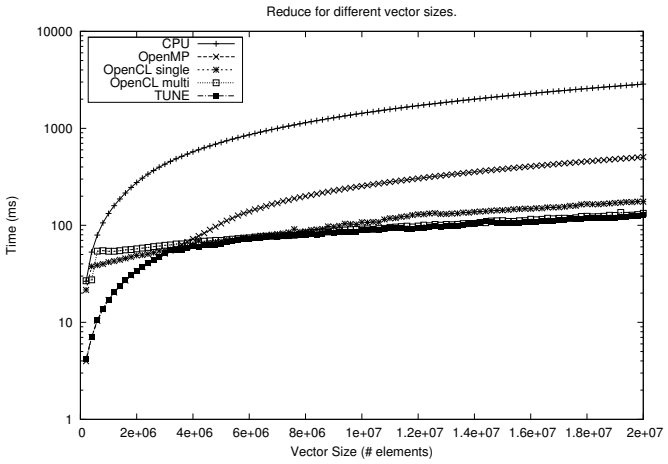

Figure 5: Vector sum with reduce, computed with an empirically determined execution plan on target architecture 1.

```
plan.add(1,3500, skepu::CPU_BACKEND);
plan.add(3501,3200000, skepu::OMP_BACKEND, 8);
plan.add(3200001,5400000, skepu::CL_BACKEND, 65535, 128);
plan.add(5400001,MAX_INT, skepu::CLM_BACKEND, 65535, 128);

globalSum.setExecPlan(plan);
```

Figure 5 shows the behavior of the empirically tuned configuration as it switches to the optimal backend for different problem sizes. Note that, for technical reasons, it is not possible (without major effort) to mix OpenCL and CUDA code within the same program. Here we decided to use OpenCL as it allows better support for multi-GPU computing. The tunable version (TUNE) selects the OpenMP Reduce for small problem sizes, OpenCL on a single GPU for medium ones, and OpenCL on two GPUs for large ones.

In order to demonstrate the potential for performance portability, we now consider target architecture 2. For this platform, we preset the execution plan with the following empirically found parameters:

```
skepu::Reduce<plus> globalSum(new plus);

skepu::ExecPlan plan;
plan.add(1,3500, skepu::CPU_BACKEND);
plan.add(3501,900000, skepu::OMP_BACKEND, 8);
plan.add(900001,MAX_INT, skepu::CL_BACKEND, 65535, 128);

globalSum.setExecPlan(plan);
```

Note that the plan does not contain an entry for dual-GPU computing because the target system only has a single GPU. Figure 6 shows the resulting performance with the new plan on the new target architecture.

In both cases, the tunable version (TUNE) follows the best back-end for the current problem data size. Also, the overhead for looking up the plan entry at run time turned out to be negligible in the considered example.

4.3 The ODE Solver

Figure 7 shows empirical tuning for the ODE solver application where we have different SkePU versions for the ODE solver, including one using the OpenMP backend and another using the OpenCL backend with a default configuration. The tuned configuration for the OpenCL(`skepu-CL tune`) uses appropriate grid-size and block-size combinations for different problem-sizes and gives significant improvement in the performance. The overall tuned configuration (`skepu TUNE`) uses a combination of the OpenMP and the OpenCL backends for different skeleton calls which is determined statically by using knowledge of the program composition. This tuning is most-likely non-optimal as it is rather impossible to statically exercise all possible combinations of different backends for each skeleton call in the program. In future, this tuning for composition of different skeletons is going to be done automatically.

5. FUTURE WORK

At the moment, setting up execution plans is still to be done manually except for the tuning parameters for GPU backends. Future work will apply machine learning techniques to construct such plans automatically from training data taken by microbenchmarks.

Currently, SkePU macro-definitions of user functions parameterizing skeleton instantiations support the same function body for all back-ends (OpenMP, CPU, CUDA, OpenCL). This works well for simple functions, but for more complex function definitions, SkePU may need to support different function definitions for different back-ends.

SkePU itself is work in progress and several additions to the skeleton library are planned for, such as support for an STL container *Matrix* for two-dimensional data. We may also consider task-parallel skeletons, such as Farm, and nesting of skeletons.

6. RELATED WORK

A lot of work and research have been made in the area of skeletal parallel programming. With the arrival of CUDA and OpenCL, which has provided an easy way of utilizing the parallel processing power of graphics hardware, the skeleton approach has also been tried in this fairly new area of

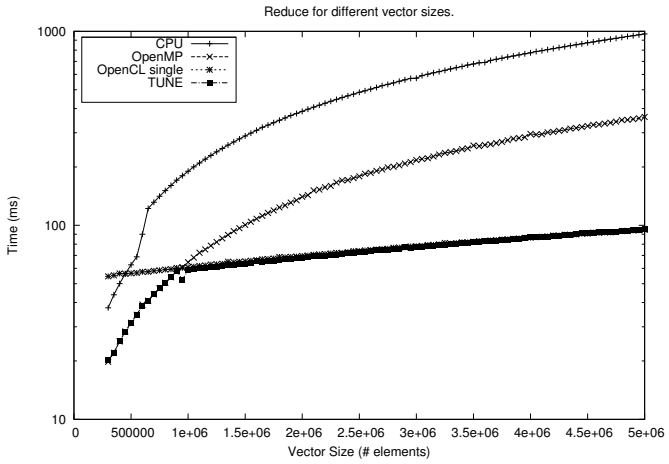

Figure 6: Vector sum with reduce, computed with an empirically determined execution plan on target architecture 2.

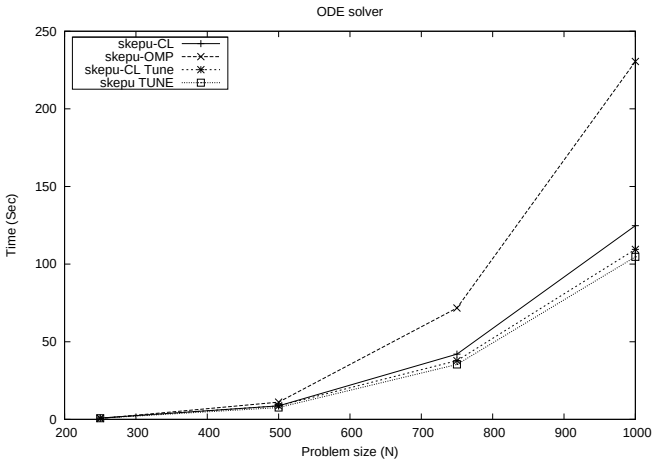

Figure 7: The ODE libsolver application with hand-tuning on the target architecture 1, highlighting potential for composed skeleton tuning for real-world applications.

parallel computing. The development of SkePU, which was presented in this paper, has been inspired by several other similar projects such as *Thrust* [6], *CUDPP* [5], or the skeleton programming frameworks by Sato and Iwasaki [12] and Kirschenmann et al. [7]. More details and references about these and further related approaches can be found in the Related Work section of our recent paper on SkePU [4].

So far SkePU implements one container, a vector type which is built around the STL vector and is largely inspired by *CuPP* [1].

Automated selection among different algorithmic variants of reductions on shared-memory multiprocessors has been considered by Yu and Rauchwerger [14].

StarPU [13] is a run-time system for accelerator based systems including GPU based systems. It contains a history based mechanism that, at run time, records the performance effect of executing code units on different kinds of resources and enables automated selection of the resource predicted most suitable based on this performance history data.

Acknowledgments

This work was funded by EU FP7, project PEPPHER, grant #248481 (www.peppher.eu), and SSF project ePUMA.

7. REFERENCES

[1] J. Breitbart. CuPP - A framework for easy CUDA integration. In *IPDPS'09: Proc. IEEE Int. Symposium on Parallel&Distributed Processing*, pages 1–8, Washington, DC, USA, 2009. IEEE Computer Society.

[2] M. Cole. *Algorithmic skeletons: structured management of parallel computation*. Pitman and MIT Press, 1989.

[3] M. Cole. Bringing skeletons out of the closet: a pragmatic manifesto for skeletal parallel programming. *Parallel Computing*, 30(3):389–406, 2004.

[4] 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.

[5] M. Harris, J. Owens, S. Sengupta, Y. Zhang, and A. Davidson. CUDPP: CUDA Data Parallel Primitives Library. `http://gpgpu.org/developer/cudpp`, 2009.

[6] J. Hoberock and N. Bell. Thrust: C++ Template Library for CUDA . `http://code.google.com/p/thrust/`, 2009.

[7] W. Kirschenmann, L. Plagne, and S. Vialle. Multi-target C++ implementation of parallel skeletons. In *POOSC '09: Proc. 8th workshop on Parallel/High-Performance Object-Oriented Scientific Computing*, pages 1–10, New York, NY, USA, 2009. ACM.

[8] Christoph Kessler and Welf Löwe. A Framework for Performance-Aware Composition of Explicitly Parallel Components. Proc. ParCo-2007 conference, Jülich/Aachen, Germany, Sept. 2007.

[9] A. Munshi. The OpenCL specification version 1.0. *Khronos OpenCL Working Group*, 2009.

[10] Nvidia. CUDA Programming Guide Version 2.3.1. *NVIDIA Corporation*, 2009.

[11] F. A. Rabhi and S. Gorlatch, editors. *Patterns and skeletons for parallel and distributed computing*. Springer-Verlag, London, UK, 2003.

[12] S. Sato and H. Iwasaki. A skeletal parallel framework with fusion optimizer for GPGPU programming. In *APLAS '09: Proceedings of the 7th Asian Symposium on Programming Languages and Systems*, pages 79–94, Berlin, Heidelberg, 2009. Springer-Verlag.

[13] StarPU project. http://runtime.bordeaux.inria.fr/StarPU/

[14] H. Yu and L. Rauchwerger. An Adaptive Algorithm Selection Framework for Reduction Parallelization. *IEEE Trans. Parallel and Distr. Syst.* 17(10):1084–1096, 2006.