Adaptive Implementation Selection in the SkePU Skeleton Programming Library

Usman Dastgeer, Lu Li, and Christoph Kessler

IDA, Linköping University, 58183 Linköping, Sweden
{usman.dastgeer,lu.li,christoph.kessler}@liu.se

Abstract. In earlier work, we have developed the SkePU skeleton programming library for modern multicore systems equipped with one or more programmable GPUs. The library internally provides four types of implementations (*implementation variants*) for each skeleton: serial C++, OpenMP, CUDA and OpenCL targeting either CPU or GPU execution respectively. Deciding which implementation would run faster for a given skeleton call depends upon the computation, problem size(s), system architecture and data locality.

In this paper, we present our work on automatic selection between these implementation variants by an offline machine learning method which generates a compact decision tree with low training overhead. The proposed selection mechanism is flexible yet high-level allowing a skeleton programmer to control different training choices at a higher abstraction level. We have evaluated our optimization strategy with 9 applications/kernels ported to our skeleton library and achieve on average more than 94% (90%) accuracy with just 0.53% (0.58%) training space exploration on two systems. Moreover, we discuss one application scenario where local optimization considering a single skeleton call can prove sub-optimal, and propose a heuristic for bulk implementation selection considering more than one skeleton call to address such application scenarios.

Keywords: Skeleton programming, GPU programming, implementation selection, adaptive offline learning, automated performance tuning.

1 Introduction

The need for power efficient computing has lead to heterogeneity and parallelism in today's computing systems. Heterogeneous systems such as GPU-based systems with disjoint memory address space already became part of mainstream computing. There exist various programming models (CUDA, OpenCL, OpenMP etc.) to program different devices present in these systems and, with GPUs becoming more general purpose every day, more and more computations can be performed on either of the CPU or GPU devices present in these systems.

Known for their performance potential, these systems expose programming difficulty as the programmer often needs to program in different programming models to do the same computation on different devices present in the system which limits code-portability. Furthermore, sustaining performance when porting an application between

C. Wu and A. Cohen (Eds.): APPT 2013, LNCS 8299, pp. 170–183, 2013.
© Springer-Verlag Berlin Heidelberg 2013

different GPU devices (*performance portability*) is a non-trivial task. The skeleton programming approach can provide a viable solution for computations that can be expressed in the form of skeletons, where *skeletons* [1, 2] are pre-defined generic components derived from higher-order functions that can be parameterized with sequential problem-specific code. A skeleton program looks like a sequential program where a skeleton computation can internally exploit parallelism and leverage other architectural features transparently by e.g. keeping different implementations for a single skeleton targeting different architectural features of the system. Map/Zip and Farm are examples of data and task-parallel skeletons respectively.

We have developed the SkePU skeleton programming library for GPU-based systems in our earlier work [3]. The library targets single-node GPU-based systems and provide code portability for skeleton programs by providing sequential C++, OpenMP, CUDA and OpenCL implementations for each of its skeleton. In this paper, we present an adaptive offline machine learning method to tune implementation selection in the SkePU library automatically. The proposed technique is implemented inside the SkePU library allowing automatic implementation selection on a given GPU-based system, for any skeleton program written using the library. To the best of our knowledge, this makes SkePU the first skeleton library for GPU-based systems that provides general-purpose, automatic implementation selection mechanism for calls to its skeleton.

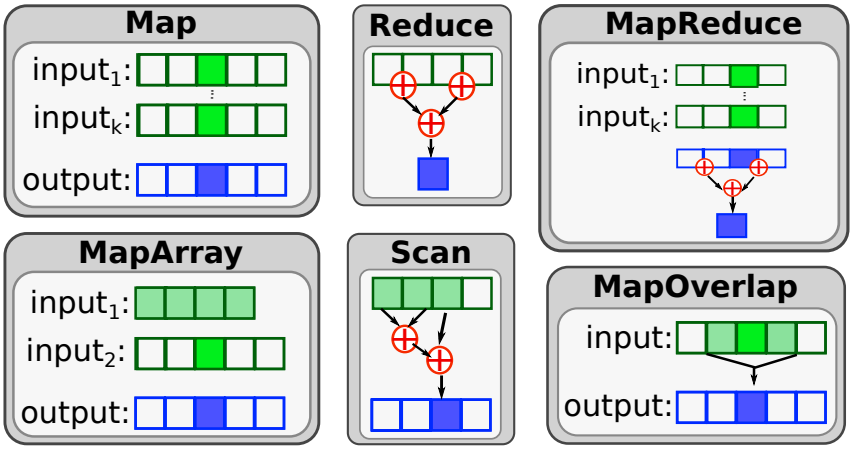


Fig. 1. Six data-parallel skeletons, here shown for vector operands: Map applies a user-defined function element-wise to input vectors. Reduce accumulates an associative binary user function over all input vector elements to produce a scalar output. MapReduce combines Map and Reduce in one step. MapArray is similar to map but all elements from the 1st operand are accessible. MapOverlap is similar to Map where elements within a (user-defined) neighbourhood are also accessible in the user function. Scan is a generic prefix-sums operation.

The paper is structured as follows: In Session 2 we briefly describe our SkePU skeleton library. The proposed adaptive tuning framework is explained in Section 3 followed by evaluation in Section 4. Related work is discussed in Section 5 while Section 6 concludes the work.

2 The SkePU Skeleton Library

The SkePU skeleton library [3] is designed in C++ and offers several data parallel skeletons including map, reduce, mapreduce, maparray, mapoverlap and scan. The operand data to skeleton calls is passed using 1D Vector and 2D Dense matrix containers. These containers internally keep track of data residing on different memory units (main memory, GPU device memory etc.) and can transparently optimize data transfers by copying data only when it is necessary. The memory management for skeleton calls' operand data is implicitly handled by the library. This can for example allow multiple skeleton operations (reads, writes) on the same data on a GPU and copies data back to main memory only when the program accesses the actual data (detected using the `[]` operator for vector elements).

Figure 1 shows a graphical description of different skeletons when used with the 1D vector container. The MapReduce skeleton is just a combination of Map and Reduce skeletons applied in a single step which is different from Google MapReduce. For a 2D matrix container operand, semantics are extended to, e.g., apply MapOverlap across all row vectors and/or across all column vectors.

```
1  // #include directives
2
3  // generates a user function 'mult_f' to be used in skeleton instantiation
4  BINARY_FUNC(mult_f, double, a, b,
5      return a*b;
6  )
7
8  int main()
9  {
10   skepu::Map<mult_f> vecMultiply(new mult_f);/* creates a map skel. object */
11
12   skepu::Vector<double> v0(50); /* 1st input vector, 50 elements */
13
14   skepu::Vector<double> v1(50); /* 2nd input vector, 50 elements */
15
16   skepu::Vector<double> res(50); /* output vector, 50 elements */
17
18   ...
19
20   vecMultiply(v0, v1, res); /* skeleton call on vectors */
21
22   std::cout<<"Result: " << res <<"\n"; /* output result vector */
23
24   ...
25 }
```

Listing 1. Multiplying two vectors element-wise using the Map skeleton

Listing 1 shows a simple operation of multiplying two vectors element-wise and writing the result into an output vector. As each skeleton in the library has multiple implementations (C++, OpenMP, CUDA, OpenCL) available, the skeleton call on Line

12 will internally be mapped to one of those implementations. Up to now, this decision was controlled by the user or was statically determined (e.g., to always use the CUDA implementation when a CUDA GPU is available). In the next section, we will explain the mechanism for tuning this implementation selection in a more intelligent manner automatically.

3 Adaptive Tuning Framework

Any skeleton call in our library enables implementation selection choice which can have performance implications. In the ideal case, we would like to select an implementation which would result in the shortest execution time. For this purpose, we devise an empirical prediction technique based on offline sample executions. In the following, we describe the technique and how it is implemented.

3.1 Idea

A simple way to do empirical offline tuning could be to exhaustively try out all *variants* for different *call context instances* to find out the *best variant* and use that for actual skeleton calls. In our case, variants are mainly the different skeleton implementations (CPU, OpenMP etc.), *call context instances* are characterized by the sizes of operand data and the *best variant* is one which results in shortest execution time. Trying out all possible context instances using exhaustive search is practically infeasible. Our offline tuning technique is rather an adaptive hierarchical search based upon a heuristic *convexity assumption* which basically means that if a certain implementation is performing best on all vertices of a D-dimensional context parameter subspace then we assume it is also the best choice for all points within the subspace. For example, considering a 1-dimensional space (i.e. only 1 input size parameter), if we find out that a certain implementation performs best on two distinct input sizes i and j we consider it best for all points between these points (i.e. for the whole range $[i, j]$). This concept is extended to D-dimensional space as described below.

3.2 Algorithm

The space $C = I_1 \times ... I_D$ of context instances for a skeleton with D possibly performance-relevant properties in the context instances is spanned by the D context property axes with considered (user-specified or default) finite intervals I_i of discrete values, for $i = 1, ..., D$. A continuous subinterval of an I_i is called a (context property value) *range*, and any cross product of such subintervals on the D axes is called a *subspace* of C. Hence, subspaces are "rectangular", i.e., subspace borders are orthogonal to the axes of C.

Our idea is to find sufficiently precise approximations by adaptively recursive splitting of subspaces by splitting the intervals I_i, $i = 1, ..., D$. Hence, subspaces are organized in a hierarchical way (following the subspace inclusion relation) and represented by a 2^D-ary tree T_C (cf. quadtrees/octrees etc.).

Our algorithm for off-line measurement starts from a trivial tree T_C that has just one node, the root (corresponding to the whole C), which is linked to its 2^D corner points

(here, the 2^D outer corners of C) that are stored in a separate table of recorded performance measurements. The implementation variants of the skeleton under examination are run with each of the corresponding 2^D context instances, possibly multiple times for averaging; a variant whose execution exceeds a timeout for a context instance is aborted and not considered further for that context instance. Now we know the winning implementation variant for each corner point and store it in the performance table, too, and T_C is properly initialized.

Consider any leaf node v in the current tree T_C representing a subspace $S_v = R_1^v \times ... \times R_D^v$ where $R_i^v \subset I_i$, $i = 1,...,D$. If the same specific implementation variant runs fastest on all context instances corresponding to the 2^D corners of S_v, we stop further exploration of that subspace and will always select that implementation whenever a context instance at run-time falls within that subspace. Otherwise, the subspace S_v may be refined further. Accordingly, the tree is extended by creating new children below v which correspond to the newly created subspaces of S_v. By iteratively refining the subspaces in breadth-first order, we generate an adaptive tree structure to represent the performance data and selection choices, which we call *dispatch tree*.

The user can specify a *maximum depth* (training depth) for this iterative refinement of the dispatch tree, which implies an upper limit on the runtime lookup time, and also a maximum tree size (number of nodes) beyond which any further refinement is cut off. Third, the user may specify a timeout for overall training time, after which the dispatch tree is considered final.

At every skeleton invocation, a run-time lookup searches through the dispatch tree starting from the root and descending into subspace nodes according to the current run-time context instance. If the search ends at a *closed leaf*, i.e., a leaf node with equal winners on all corners of its subspace, the winning implementation variant can be looked up in the node. If the search ends in an *open leaf* with different winners on its borders (e.g., due to reaching the specified cut-off depth), we perform an approximation within that range by choosing the implementation that runs fastest on the subspace corner with the shortest Euclidean distance from the current run-time context instance.

The deeper the algorithm explores the tree, the better precision the dynamic composer can offer for the composition choice; however, it requires more off-line training time and more runtime lookup overhead as well. We give the option to let the user decide the trade-off between training time and precision by setting the cut-off depth, size and time in the component interface descriptor. Figure 2 shows an example for 1-dimensional space exploration. The algorithm can recursively split and refine subspaces until it finds common winners for all points for a subspace (i.e. the subspace becomes closed) or the user-specified maximum depth is reached.

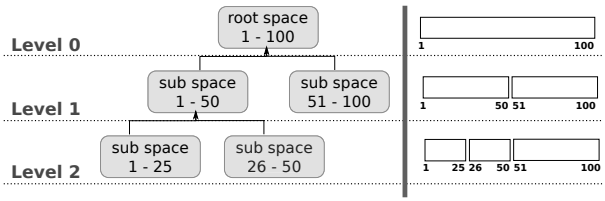


Fig. 2. Depiction of how a 1-dimensional space is recursively cut into subspaces (right) and the resulting dispatch tree (left)

3.3 Implementation Details

In order to transparently integrate the tuning mechanism in our existing skeleton library, we have designed it using C++ templates as an include header. As shown in Listing 2, a `Tuner` class is introduced which is parameterized by the skeleton type and user function(s). The user needs to supply a unique `ID` (string) for the *skeletonlet*[1] being tuned as well as lower and upper bounds for the size of each operand. The `ID` decouples the skeletonlet and tuner, and allows e.g. multiple tuning scenarios even for the same skeletonlet to co-exist. Internally the tuner applies certain optimizations (e.g. dimensionality reduction) and returns an execution plan which is later assigned to the skeleton object. An execution plan is a simple data structure that internally tracks the best implementation for each subspace and provides lookup facilities. After the execution plan is set, the expected best implementation for any skeleton in a given call context will be automatically selected.

The `Tuner` supports automatic persistence and loading of execution plans. If the execution plan with same configuration already exists, it loads and returns it from a repository without any tuning overhead; otherwise it invokes the tuning algorithm and constructs an execution plan. The generated tuning plan is stored for future usages to avoid re-tuning every time the skeleton program is executed. Furthermore, the tuning and actual execution can happen during the same program execution, as shown in Listing 2. When porting the same skeleton program to a new architecture, the tuner would automatically construct an exection plan for the new architecture without requiring any changes in the user program.

```
...

int main()
{
  skepu::Map<mult_f> vecMultiply(new mult_f);

/* specify where input and output operand data (need to) resides */

  int opInLoc[]  = {0, -1}; /* 1st/2nd input operand in GPU/main memory */
  int opOutFlag[] = {1};    /* copy result back to main memory or not */

/* specify lower and upper bounds for training range */

  int lowerBounds[] = {10, 10, 10};
  int upperBounds[] = {50000000, 50000000, 50000000};

/* invoke the tuner whichs returns the execution plan */

  skepu::ExecPlan plan = skepu::Tuner<mult_f, MAP>("vMult", 3, lowerBounds,
                                        upperBounds, opInLoc, opOutFlag)();

/* assign the execution plan to the skeleton object */

  vecMultiply.setExecPlan(execPlan);

  ...
}
```

Listing 2. Tuning the vector multiply skeleton call

[1] A pair of user-function(s), skeleton type.

Dimensionality Reduction. We apply several optimizations based on domain specific knowledge that each skeleton implementation exposes. For example, considering the fact that all operands (inputs, output) in a map skeleton should be of exactly the same size, we have considered it as 1-dimensional space instead of 3-dimensional space (with 2 input and 1 output operands). This significantly reduces the training cost and is transparently done by considering semantics of the skeleton being used. Similar optimizations are applied for MapOverlap, MapReduce and Scan skeletons.

Data Locality. Current GPU based systems internally have disjoint physical memory and both the Vector and Matrix containers in our skeleton library can track their payload data on different memory units. The operand data locality matters when measuring the execution time for both CPU and GPU execution for a skeleton implementation, as operand data may or may not exist in the right memory unit; in case it is not available in the right place, extra overhead for data copy needs to be encountered. Selection of the expected best implementation for a given problem size cannot be made without considering where the input data resides and where the output data needs to be copied back as the data copying overhead between different memory units could affect the selection of the best performing variant. One solution could be to assume that operand data is always located in a specific memory unit (e.g., main memory) and, depending upon where the skeleton implementation executes, a copy may or may not be required. This solution is simple but unflexible as even different operands of a single skeleton call may reside at different memory spaces depending upon their previous usage with other skeleton calls. On the other hand, delaying the decision about operand data locality to runtime is infeasible as we need to know the data transfer cost to determine, offline, the best variant for a given problem size.

We have devised a simple mechanism for the programmer to specify knowledge about operands' data locality. By default, we assume that operand data resides in main memory and cost for transferring output data back to main memory is not included in the skeleton execution. However, the programmer can easily override this behavior by specifying:

- An integer flag for each input operand specifying the memory unit where it is residing (default main memory = -1). In the example in Listing 2 (line 6), the tuner will determine the best implementation considering that the first operand resides in the GPU device memory while the second input operand resides in main memory.
- A binary flag for each output operand specifying whether it should be transferred back to main memory or not. In the example in Listing 2 (line 7), the best implementation is determined considering that the output operand needs to be copied back to main memory after skeleton execution.

4 Evaluation

For evaluation, we have implemented five applications (NBody simulation, **S**mooth **P**article **H**ydrodynamics, LU factorization, Mandelbrot, Taylor series) and four kernels (**M**ean **S**quared **E**rror, **P**eak **S**ignal-to-**N**oise **R**atio, **P**earson **P**roduct-**M**oment

Correlation **C**oefficient, dot product) with skeletons available in our skeleton library. We use two different systems to demonstrate effectivness of our tuning mechanism in doing implementation selection while adjusting to platform differences: **System A** with Xeon® E5520 CPUs running at 2.27GHz with 1 NVIDIA® C2050 GPU with L1/L2 cache support and **System B** with Xeon® X5550 CPUs running at 2.67GHz with a lower-end GPU (NVIDIA® C1060 GPU).

For each application/kernel, we call the tuner on a given training range (i.e., problem size ranges for each operand) and it internally explores some points in the training space and construct an execution plan using the algorithm described in Section 3.2. Afterwards, we do the actual execution by selecting a set of sample points (different from the training points) within that range and do the actual execution using the tuned version as well as using each implementation variant (CPU, OpenMP, CUDA) on those selected points[2]. The same problem size ranges are used for experiments on both systems and no modifications in the program source code are made when porting the applications between both systems. Furthermore, for all experiments, we set the maximum training depth to 10 and Euclidian distance is used to estimate the best variant if no best variant is found for a subspace until depth 10.

4.1 Tuning Efficiency

Figure 3 shows execution of eight applications/kernels on System A. On the horizontal axis, we list the problem sizes whereas the vertical axis represents the execution time. For each application/kernel, we list the percentage of training space that is explored by the tuner to construct the execution plan as well as average accuracy of execution with the tuned version[3]. As it is practically infeasible to try out all points in the training range, accuracy is measured by averaging over the ratio of execution time with the tuned configuration with execution time of the best from direct execution (CPU, OpenMP, CUDA) over all sample points. Due to small variations in execution times during actual measurements, accuracy could become more than 100% in some cases (e.g. an OpenMP implementation can take slightly different time even between successive executions [18]). Averaging over all eight applications/kernels, 94% accuracy has been achieved with just 0.2% training space exploration.

When porting the applications to System B, no changes in the applications' source code are required and the execution plan is automatically tuned before first execution on the new platform. As shown in Figure 4, the tuner is able to effectively adjust to platform differences without requiring any user intervention and we achieved on average 91% accuracy with just 0.3% training space exploration. For execution with the tuned version, the overhead of looking up the best implementation in the execution plan for a given call context is included in the measurement, which proved to be negligible.

[2] We did not consider the OpenCL implementations for experiments as they are similar to CUDA in performance on NVIDIA GPUs and are primarily written for execution on accelerator devices not supporting CUDA.

[3] In the tuned version, implementation selection is made based upon the execution plan returned by the tuner.

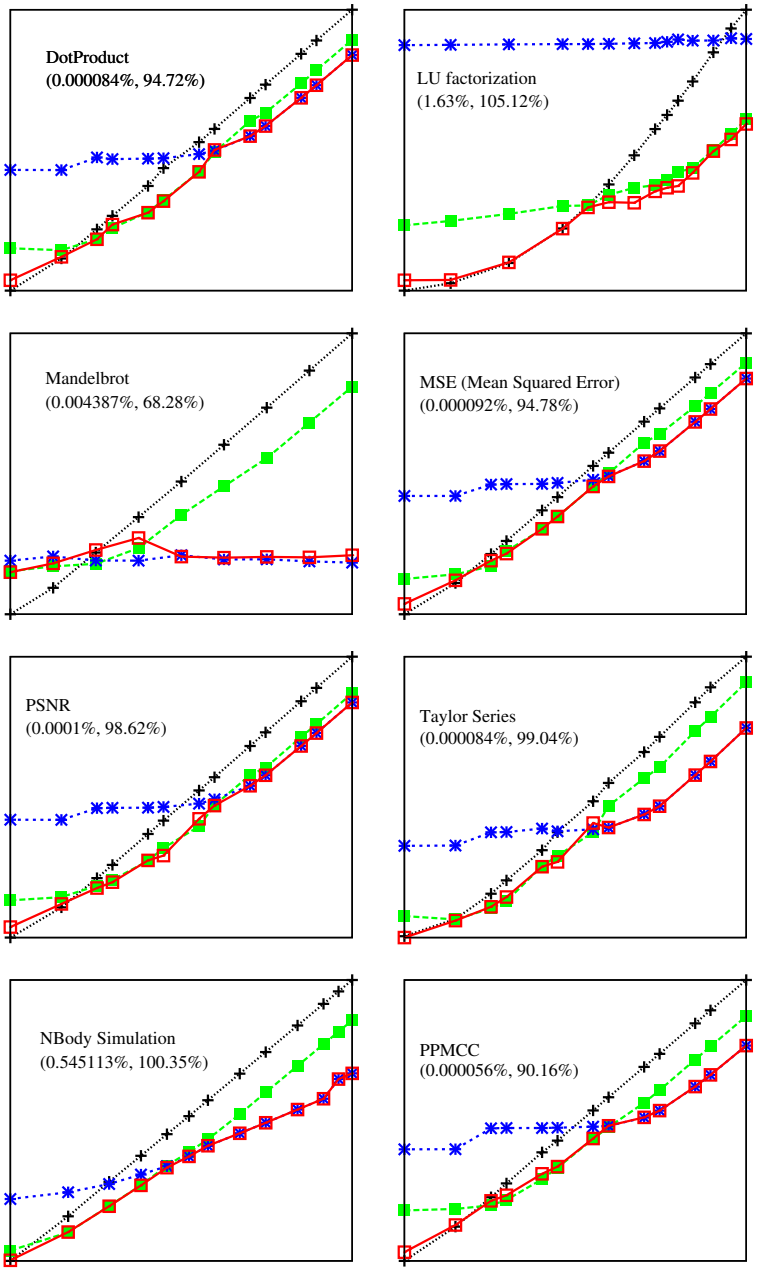


Fig. 3. Execution time of eight applications/kernels for different problem sizes on System A with respective *training space, accuracy* figures. On average, 94% accuracy has been achieved with just 0.2% training space exploration. [**Legend:** Black(CPU, **+**), Green(OpenMP ■), Blue(CUDA ∗), Red(Tuned □)] .

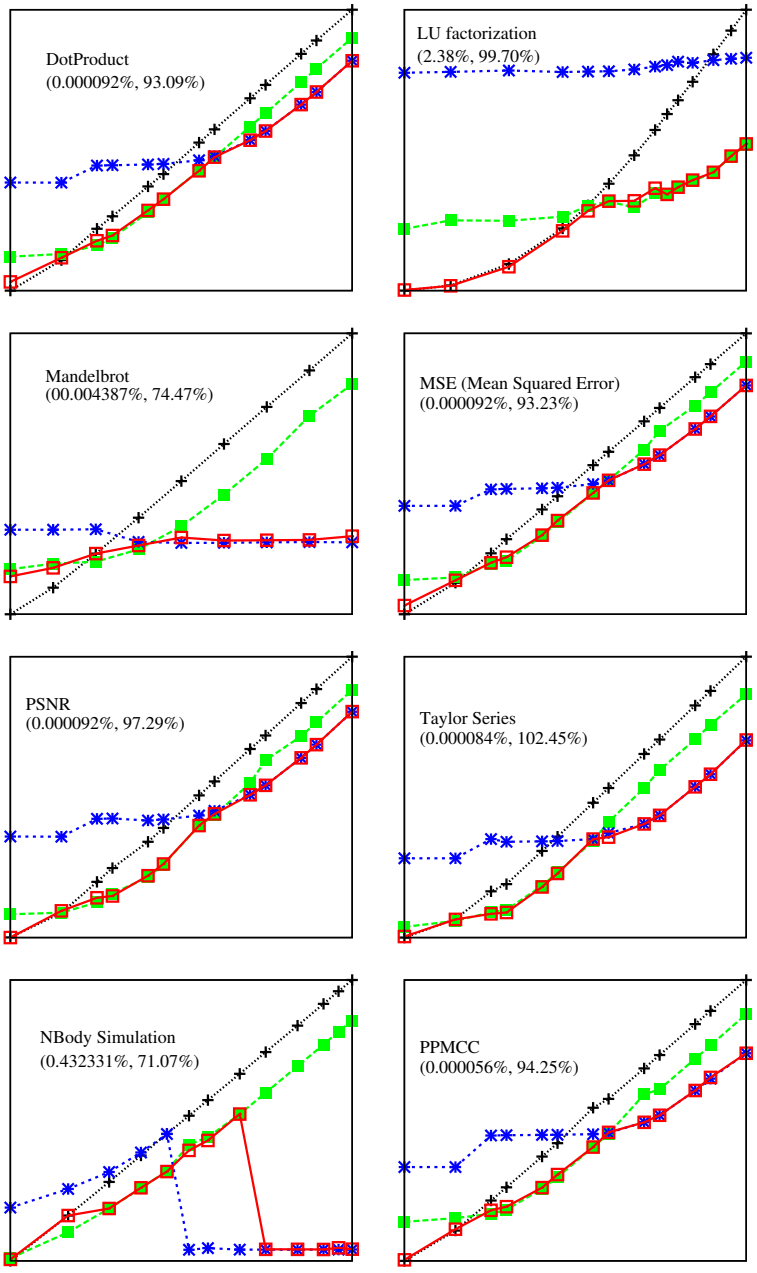


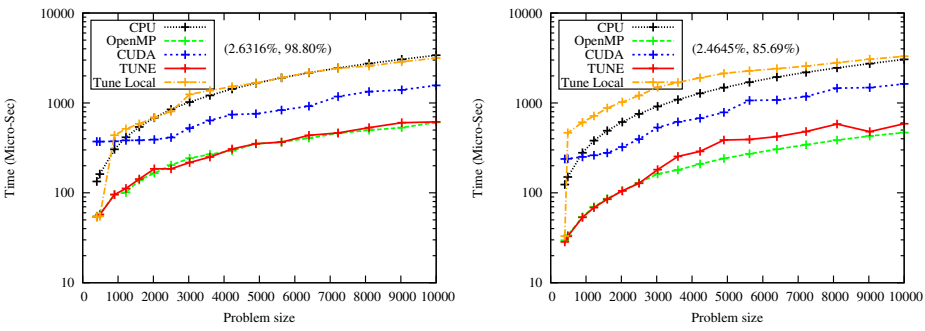
Fig. 4. Execution time of eight applications/kernels for different problem sizes on System B with respective *training space, accuracy* figures. On average, 91% accuracy has been achieved with just 0.3% training space exploration. [**Legend:** Black(CPU, **+**), Green(OpenMP ■), Blue(CUDA ∗), Red(Tuned □)].

4.2 Bulk Execution

The Tuner predicts the best implementation for each skeleton call individually based on operand data locality and execution time of each skeleton implementation available. As we have seen in the previous section, this works fine for both simple and complex applications/kernels with skeleton calls of one or more types. However, in some applications with multiple skeleton calls having different computational complexity and constrained in a data dependency chain, locally optimal decisions for each skeleton call may result in a globally sub-optimal decision. Listing 3 shows such an application scenario in the SPH (Smooth Particle Hydrodynamics) application. This application has three different types of skeleton calls with different computational complexity, operating on the same data inside a loop. For a given problem size, the tuner might determine OpenMP, CUDA and OpenMP execution as best for `skeleton_1`, `skeleton_2` and `skeleton_3` calls respectively. Although making the best decision for each skeleton call individually, this would result in lot of expensive data transfers (over PCIe bus between main and GPU device memory) as output produced by the `skeleton_1` call becomes an input to the `skeleton_2` call and so forth. Doing it inside a loop makes it even worse as these data transfers would need to be done in each loop iteration.

```
1  ...
2  for(....)
3  {
4     skeleton_1(v0, v1, v1);
5     skeleton_2(v1, v0, v0);
6     skeleton_3(v0, v0);
7  }
8  ...
```

Listing 3. SPH pseudo-code



(a) System A (b) System B

Fig. 5. Execution of SPH on both systems with respective *training space, accuracy* figures

For such skeleton calls with different computational complexity and tight data dependency, we implement a simple bulk selection heuristic in our tuner. For a sequence of skeleton calls constrained in a data dependency chain, the skeleton programmer can

specify a `group id` as a last (optional) argument to each skeleton call. All skeleton calls with same group id are scheduled on compute units with the same memory address space which is determined as the expected best one for the first skeleton call in the group.

Figure 5 shows execution of SPH on both evaluation systems. As shown in the figure, predicting the best implementation for each skeleton call individually (the `Tune Local` version) yields poor performance in this case. The tuner version with the bulk heuristic performs better in this case by considering the interconnection between the skeleton calls and mapping them to the same backend.

5 Related Work

Besides SkePU, SkelCL [4] and Muesli [8] are currently the two main skeleton programming libraries for GPU-based systems. They all provide some common data parallel skeletons such as Map/Zip and also provide memory management for GPU-execution. Hybrid execution and automatic implementation selection are some important capabilities of SkePU that distinguish it from the other two libraries (see [3] for details).

MultiSkel [9] provides a CUDA code generation facility for skeleton programs written in C++. Bones [11] targets automatic transformation of C programs to CUDA for GPU execution by identifying occurences of pre-defined patterns/skeletons in the sequential code. Buono et al. [10] describe a set of low-level algorithmic constructs that can be composed in a hierarchical manner to match application-level patterns.

In our earlier work [6], we used a genetic algorithm to do offline implementation selection as well as selection of some tuning parameters for each implementation type (number of threads for OpenMP and thread block size for CUDA and OpenCL backend). However, the current approach using the convexity assumption requires much less traning space exploration while achieving better accuracy for a variety of applications.

Empirical exploration is employed by Collins et al. [12] in their FastFlow parallel skeleton framework. They use Monte Carlo search of a random subset of the space and use knowledge about variable dependencies to further reduce the search space. However, their tuning is about finding the suitable values for the tuning parameters rather than implementation selection; also the FastFlow library currently targets multicore homogeneous systems and does not support GPU-based systems.

There exist a large body of work in empirical tuning (e.g. [17, 13]) as well as usage of decision trees [15, 14] and C4.5 algorithm [16]. Our work differs from other empirical auto-tuning approaches in two ways: First, our focus is on implementation selection rather than tuning (machine- or algorithm- specific) parameters for an implementation. This enables us to use the convexity assumption to significantly reduce the training cost compared to random sampling employed by other parametric tuning frameworks. A similar approach using the convexity assumption is used in our earlier work [7] for PEPPHER components composition [5]. Secondly, we use an adaptive method to explore the sampling space selectively in an attempt to minimize the sampling and training cost while building the dispatch tree simultaneously. This is in contrast to classical approaches that do the sampling and learning separately; thereby considering many uninteresting but expensive sample points.

6 Conclusion

Having different implementations for a computation, possibly in different programming models, can give both performance and portability if some intelligent selection mechanism is in place. We proposed and implemented an efficient empirical auto-tuning method for doing implementation selection in a skeleton library for GPU-based systems. It uses an adaptive algorithm based on a heuristic convexity assumption to build up a decision tree by exploring parameter subspaces in a recursive manner. Evaluation with nine applications/kernels have demonstrated effectiveness of our approach in predicting the best implementation, with great accuracy (more than 90%), for a given execution context with just 0.5% training space exploration on two different systems. The selection and tuning mechanism is implemented inside the SkePU skeleton library, requiring no modifications in the user-code when porting the application to a new system.

References

[1] Cole, M.: Algorithmic Skeletons: Structured management of parallel computation. MIT Press, Cambdridge (1989)

[2] Kessler, C., Gorlatch, S., Enmyren, J., Dastgeer, U., Steuwer, M., Kegel, P.: Skeleton Programming for Portable Many-Core Computing. In: Pllana, S., Xhafa, F. (eds.) Programming Multi-Core and Many-Core Computing Systems, 20 pages. Wiley Interscience, New York (2013)

[3] Dastgeer, U.: Skeleton Programming for Heterogeneous GPU-based Systems. Licentiate thesis. Thesis No 1504. Dept. of Comp. and Inf. Sci., Linköping University (October 2011)

[4] Steuwer, M., Kegel, P., Gorlatch, S.: SkelCL - A Portable Skeleton Library for High-Level GPU Programming. In: IEEE Int. Sym. on Par. and Dist. Proc. Workshop and Phd Forum (IPDPSW), Anchorage, USA (2011)

[5] Dastgeer, U., Li, L., Kessler, C.: The PEPPHER Composition Tool: Performance-Aware Dynamic Composition of Applications for GPU-based Systems. In: Proc. 2012 Int. Workshop on Multi-Core Computing Systems (MuCoCoS 2012), in conjunction with Supercomputing Conference (SC 2012), Salt Lake City, Utah, USA (2012)

[6] Dastgeer, U., Enmyren, J., Kessler, C.W.: Auto-tuning SkePU: A multi-backend skeleton programming framework for multi-GPU systems. In: Proc. of the 4th Int. Workshop on Multicore Soft. Eng (IWMSE 2011). ACM, NY (2011)

[7] Li, L., Dastgeer, U., Kessler, C.: Adaptive off-line tuning for optimized composition of components for heterogeneous many-core systems. In: Daydé, M., Marques, O., Nakajima, K. (eds.) VECPAR. LNCS, vol. 7851, pp. 329–345. Springer, Heidelberg (2013)

[8] Ernsting, S., Kuchen, H.: Algorithmic skeletons for multi-core, multi-GPU systems and clusters. Int. J. of High Perf. Comp. and Netw. 7(2), 129–138 (2012)

[9] Tung, L.D., Duc, N.H., Anh, P.T., Hoang, N.H., Thap, N.M.: An Intermediate Library for Multi-GPUs Computing Skeletons. In: IEEE RIVF International Conference on Computing and Communication Technologies, Research, Innovation, and Vision for the Future, RIVF (2012)

[10] Buono, D., Danelutto, M., Lametti, S., Torquati, M.: Parallel Patterns for General Purpose Many-Core. In: Euromicro International Conference on Parallel, Distributed and Network-based Processing, PDP 2013. IEEE Computer Society Press (2013)

[11] Nugteren, C., Corporaal, H.: Introducing 'Bones': A parallelizing source-to-source compiler based on algorithmic skeletons. In: Proc. 5th Annual Workshop on General Purpose Proc. with Graph. Proc. Units (GPGPU-5). ACM, NY (2012)
[12] Collins, A., Fensch, C., Leather, H.: Auto-tuning parallel skeletons. Parallel Processing Letters 22(02) (2012)
[13] Thomas, N., Tanase, G., Tkachyshyn, O., Perdue, J., Amato, N.M., Rauchwerger, L.: A framework for adaptive algorithm selection in STAPL. In: Proc. 10th Symposium on Principles and Practice of Parallel Programming (PPoPP 2005). ACM, New York (2005)
[14] Kohavi, R.: Scaling Up the Accuracy of Naive-Bayes Classifiers: A Decision-Tree Hybrid. In: Proc. 2nd Int. Conf. on Knowledge Discovery and Data Mining. AAAI Press (1996)
[15] Püschel, M., Moura, J.M.F., Johnson, J., Padua, D., Veloso, M., Singer, B., Xiong, J., Franchetti, F., Gacic, A., Voronenko, Y., Chen, K., Johnson, R.W., Rizzolo, N.: SPIRAL: Code Generation for DSP Transforms. Proceedings of the IEEE, Special issue on *Program Generation, Optimization, and Adaptation* 93(2), 232–275 (2005)
[16] Quinlan, J.R.: C4.5: Programs for Machine Learning. Morgan Kaufmann Publishers Inc., San Francisco (1993)
[17] Vuduc, R., Demmel, J.W., Bilmes, J.A.: Statistical Models for Empirical Search-Based Performance Tuning. Int. J. High Perform. Comput. Appl. 18(1) (2004)
[18] Mazouz, A., Touati, S., Barthou, D.: Performance evaluation and analysis of thread pinning strategies on multi-core platforms: Case study of SPEC OMP applications on Intel architectures. In: International Conference on High Performance Computing and Simulation, HPCS (2011)