A Programming System for Future Proofing Performance Critical Libraries

Li-Wen Chang[1], Izzat El Hajj[1], Hee-Seok Kim[1], Juan Gómez-Luna[2], Abdul Dakkak[1], Wen-mei Hwu[1]

[1]University of Illinois at Urbana-Champaign, [2]Universidad de Córdoba

{lchang20,elhajj2,kim868}@illinois.edu, el1goluj@uco.es, {dakkak,w-hwu}@illinois.edu

Abstract

We present Tangram, a programming system for writing performance-portable programs. The language enables programmers to write computation and composition codelets, supported by tuning knobs and primitives for expressing data parallelism and work decomposition. The compiler and runtime use a set of techniques such as hierarchical composition, coarsening, data placement, tuning, and runtime selection based on input characteristics and micro-profiling. The resulting performance is competitive with optimized vendor libraries.

1. Introduction

With heterogeneity becoming ubiquitous in modern computing systems, performance-portability has become a growing concern. Ideally, programs would achieve high performance on different devices and device generations without software re-development. However, portability requires architecture-neutral program representation while performance requires architecture-specific customization and tuning. These two conflicting goals that make the ideal of performance portability challenging.

Architecture types and generations differ in many ways that make optimizations non-portable. These differences include: granularity of parallelism, hierarchical organization of hardware, memory system features, resource size and abundance, and others. Various techniques have been proposed to tackle these differences. Overexpressing parallelism in data-parallel workloads allows compilers to coarsen work according to the granularity of parallelism in the device [5, 7, 11]. Nested parallelism with algorithmic choice and recursive composition rules [3, 6] adapts programs to devices with different hierarchies. Automatic data placement [1, 4] assigns data structures to suitable memories based on memory system features. Autotuning [2, 10, 12] selects optimal execution parameters that match hardware resource constraints.

We present Tangram, a programming framework that integrates many of these techniques into a single system and manages interactions between them. Tangram programs express computation in terms of *interchangeable*, *composable*, and *tunable* building blocks called codelets. The language supports expressing computations and composition rules interchangeably, supplies primitives for expressing data parallelism and work decomposition, and provides syntax for creating tuning knobs. The compiler composes kernels based on the device hierarchy and performs automatic data placement and tuning, pruning the design space throughout to minimize compilation time and the final number of candidates. The runtime further prunes candidates based on properties of the input data and low-overhead profiling to select a final candidate.

Permission to make digital or hard copies of part or all of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, to republish, to post on servers, or to redistribute to lists, contact the Owner/Author. Request permissions from permissions@acm.org or Publications Dept., ACM, Inc., fax +1 (212) 869-0481. Copyright 2016 held by Owner/Author. Publication Rights Licensed to ACM.

PPoPP '16, March 12-16, 2016, Barcelona, Spain
Copyright © 2016 ACM 978-1-4503-4092-2/16/03...$15.00
DOI: http://dx.doi.org/10.1145/2851141.2851178

2. System Overview

2.1 Language

The Tangram language is built around *spectrums* and *codelets*. A spectrum represents a unique computation, while a codelet is a code fragment that implements that computation. A spectrum can have multiple codelets all having the same name and function signature, but different implementations. Qualifiers are used to mark spectrums and codelets in code.

Codelets are *atomic* if they are self contained, and *compound* if they compose/decompose work and invoke spectrums. Atomic codelets can be *scalar* or *vector*. Scalar codelets are oblivious to other parallel workers, while vector codelets can communicate and synchronize with other workers in the same vector. Compound codelets use built-in primitives to express data parallelism and work decomposition. All codelets use built-in containers to facillitate compiler analysis of memory accesses. Qualifiers are used to specify tunable parameters in code.

A detailed list of Tangram's qualifiers, primitives, and built-in containers as well as a full example can be found in [8].

2.2 Compiler

Figure 1 shows the overall workflow of Tangram's compiler and runtime. For a given spectrum, the compiler composes codelets in different combinations to produce multiple candidate ASTs, pruning bad combinations along the way. For each AST, the compiler makes data-placement decisions for each data structure and selects values for each tunable variable. This phase also results in multiple combinations of decisions which are pruned. The final result is a set of candidate kernels which are passed to the runtime.

Composition involves choosing the compound codelet to perform the work decomposition at each level of hierarchy in the device except the lowest, then for each level choosing the atomic codelet to perform the computation. Compound codelets are chosen according to the decompositions that expose the most parallelism, have the best locality, and provide the greatest tuning freedom. For atomic codelets, scalar or vector codelets are selected based on availability of vectorization at the level. At each decision point, multiple codelets can be selected which captures algorithmic choice, and results in multiple output ASTs.

The candidate ASTs are traversed to generate the final candidate kernels. A data placement decision is made for each container

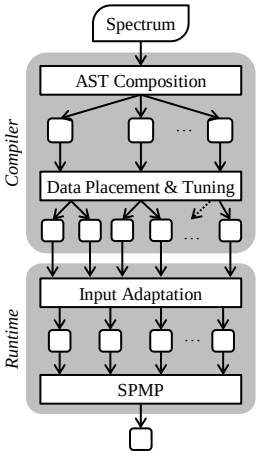

Figure 1: Tangram workflow

in the AST. A rule-based [1] data placement scheme is used, but the framework can also support a model-based [4] scheme. A tuning decision is also made for each tunable variable in the AST. Tunable variables are usually vector widths or container partition sizes which eventually specializes into traditionally tuned parameters (tile size, work-group size, coarsening factor, etc.). Tuning variables are assigned based on a performance model which considers hardware vector width, cache/scratchpad size, occupancy constraints, etc. Different combinations of decisions result in multiple candidate kernels which are finally passed to the runtime.

2.3 Runtime

Two actions are performed at runtime to select the final candidate: input adaptation and profiling.

Input adaptation refers to selecting between kernels based on properties of the input known at runtime. For example, deciding between a kernel that places data in scratchpad memory and one that doesn't can be done based on whether the size of the input is small enough. This is carried out via decision trees generated by the compiler.

Profiling is performed via a novel technique called Simultaneous Productive Micro-Profiling (SPMP). SPMP runs candidate kernels in parallel each on part of the device (hence, simultaneous). Each kernel executes on a small portion of the input (hence, micro) and its result contributes to the final output (hence, productive). The performance of each partial kernel execution is evaluated and the best kernel is picked to finish the job. This technique is described in detail in [9].

3. Results

Clang was modified to support parsing Tangram qualifiers, while containers and primitives were parsed as C++ template classes/functions. Kernels were generated in C, OpenMP, and CUDA and compiled with the ICC 15.0.2, OpenMP 4.0, and CUDA 7.0 respectively. The evaluation platforms are an i7-3820 Sandy Bridge CPU, a C2050 Fermi GPU, and a K20c Kepler GPU.

Five applications were implemented in Tangram: scan, spmv, dgemm, kmeans, and bfs. The references compared to are Thrust 1.9 for scan, MKL 11.2.2 and CUBLAS/CUSPARSE 7.0 for spmv (CSR) and dgemm, and Rodinia 3.0 [13] for kmeans and bfs.

The results are shown in Figure 2. Tangram's scan significantly outperforms the reference, benefitting particularly from Tangram's ability to fuse map iterators during composition. Even with fusion disabled (not shown in figure), it is still at least 50% faster due to better tuning. Tangram's spmv performs within 10% of all reference implementations. It succeeds at prefetching data during data placement which is key for good performance in this app. Tangram's dgemm performs within 30% of all reference implementations. This benchmark is bound by instruction throughput and references write assembly which is hard to compete with. Tangram's kmeans significantly outperforms the reference, mainly due to finding a better loop ordering for locality. Tangram's bfs performs within 10% of all reference implementations. It chooses to parallelize both vertex status checking and edge index fetching whereas the reference serializes the latter.

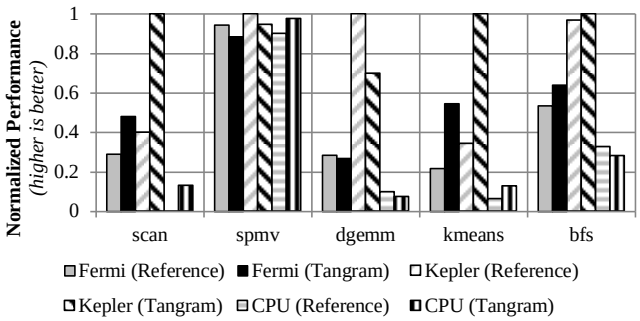

Figure 2: Tangram Performance Results

4. Acknowledgements

This work is supported by the Starnet Center for Future Architecture Research (C-FAR) and the DoE Vancouver Project (DE-FC0210ER26004/DE-SC0005515).

References

[1] B. Jang et al. Exploiting memory access patterns to improve memory performance in data-parallel architectures. *IEEE Trans. Parallel Distrib. Syst.*, 22(1):105–118, 2011.

[2] D. Merrill et al. Policy-based tuning for performance portability and library co-optimization. In *InPar*, pages 1–10, 2012.

[3] G. Blelloch. NESL: A nested data-parallel language. Technical report, Pittsburgh, PA, USA, 1992.

[4] G. Chen et al. PORPLE: An extensible optimizer for portable data placement on GPU. In *MICRO*, pages 88–100, 2014.

[5] H.-S. Kim et al. Locality-centric thread scheduling for bulk-synchronous programming models on cpu architectures. In *CGO*, pages 257–268, 2015.

[6] J. Ansel et al. Petabricks: A language and compiler for algorithmic choice. In *PLDI*, pages 38–49, 2009.

[7] R. Karrenberg and S. Hack. Improving Performance of OpenCL on CPUs. In *CC*, pages 1–20, 2012.

[8] L.-W. Chang et al. Tangram: a high-level language for performance portable code synthesis. In *In Programmability Issues for Heterogeneous Multicores*, 2015.

[9] L.-W. Chang et al. Dysel: Lightweight dynamic selection for kernel-based data-parallel programming model. In *ASPLOS*, 2016 (in press).

[10] M. Püschel et al. Spiral: A generator for platform-adapted libraries of signal processing alogorithms. *International Journal of High Performance Computing Applications*, 18(1):21–45, 2004.

[11] P. Jääskeläinen et al. pocl: A performance-portable OpenCL implementation, 2014.

[12] R. C. Whaley et el. Automated empirical optimizations of software and the atlas project. *Parallel Computing*, 27(1):3–35, 2001.

[13] S. Che et al. Rodinia: A benchmark suite for heterogeneous computing. In *IISWC*, pages 44–54, 2009.