Tangram: a High-level Language for Performance Portable Code Synthesis

Li-Wen Chang, Abdul Dakkak, Christopher I. Rodrigues, and Wen-mei Hwu

University of Illinois, Urbana, IL USA

Abstract. We propose Tangram, a general-purpose high-level language that achieves high performance across architectures. In Tangram, a program is written by synthesizing elemental pieces of code snippets, called codelets. A codelet can have multiple semantic-preserving implementations to enable automated algorithm and implementation selection. An implementation of a codelet can be written with tunable knobs to allow architecture-specific parameterization. The Tangram compiler produces highly optimized code by choosing and composing architecture-friendly codelets, and then tuning the knobs for the target architecture.
We demonstrate that Tangram's synthesized programs are comparable in performance to existing well-optimized codes on both CPUs and GPUs. The language is defined in a concise and maintainable way to improve debuggability and to enable progressive improvement. This strategy allows users to extend their applications and achieves higher performance on existing architectures and new architectures.

1 Introduction

Performance portability has become critical for the adoption of heterogeneous architectures. The current practice is largely a tight coupling of architectures and languages targeting those architectures. As the industry gains increasing momentum toward heterogeneity and parallelism, motivated by the desire to decrease power utilization while increasing performance, the cost of writing, maintaining and porting a code to multiple architectures becomes a substantial economic burden. Architecture-specific optimizations are not necessarily transferable to other architectures, forcing programmers to reinvest in performance optimization. A portable and high performance programming language is therefore desired by both the industry and the research community.

There have been several works to tackle the programming challenge for heterogeneous architectures. First, OpenCL provides an abstract computing model allowing programmers to write portable code. OpenCL is however biased to GPU style programs over other architectures. It has been shown that non-trivial transformations [6, 8–10, 16–19] are required to port OpenCL kernels to other architectures. Since OpenCL code often has embedded architecture-specific optimizations, automating the porting process becomes challenging.

Domain Specific Languages (DSLs) [4, 13, 14] take a different approach by specializing the language with certain types of computation patterns for a particular domain. A DSL compiler translates the high-level computation patterns

to a general language, and potentially can target different architectures with adaptive implementations. Their applicability however is largely confined to the intended domain.

General-purpose high-level languages using a library of algorithmic skeletons [2, 5, 7, 15] can also provide abstraction for computation and data, and rely on adaptive libraries to achieve performance portability. To fully support general purpose applications, the coverage of skeletons to architecture becomes an important issue.

In this paper, we present a general-purpose, high-level programming language, Tangram, that offers performance portability while maintaining productivity of the programmer. Tangram achieves this goal by decoupling architecture-dependent optimizations from high-level abstractions of computational patterns. Unlike algorithmic skeletons [5, 7, 15], which contain rich computation and data-structure libraries targeting a variety of architectures, Tangram only provides data-structure-based libraries. Transformations occur in Tangram either with data-structure-based libraries or by explicitly specifying an optimization rewriting rule. The rules are more aggressive than conventional compiler transformations, since the programmer gives the compiler more information from the application domain.

This paper makes following contributions:

- We propose a tuning-friendly high-level language that achieves high performance and performance portability on both CPUs and GPUs.
- We demonstrate that programs written in the proposed language are easy to maintain, debug, and incrementally improve, thus increasing efficiency of the programmer.

The rest of the paper is organized as follows: Section 2 describes Tangram's design. Section 3 presents features of Tangram. Section 4 shows selected examples and their performance evaluation. Section 5 discusses the current limitation and potential future work. Section 6 discusses related work, and Section 7 concludes.

2 Tangram Language

Performance portability is addressed in Tangram using a mixture of syntax to allow for performance tuned variables, abstract containers that simplify the analysis and development of algorithms, and loop manipulation primitives that enable transformations such as tiling, vectorization, and threading. Tangram admits multiple functionally equivalent implementations. This approach enlarges the design space and enables performance portable support for different hardware and runtime primitives. Type qualifiers and annotations allow programmers to override the choice of codelets and/or to specify the chosen transformation. This strategy allows programmers to bypass the tuning phase and provide customized algorithms for a specific algorithm or architecture.

```
#pragma tgm
int reduce(const Array<1,int> in) {
  int len = in.size();
  int accum = 0;
  for(int i=0; i<len; i++) {
    accum += in[i];
  }
  return accum;
}
```

(a) sequential atomic codelet

```
#pragma tgm tag(asso_tiled)
int reduce(const Array<1,int> in) {
  __tunable int p;
  int len = in.size();
  int tile_size= (len+p-1)/p;
  return reduce( map( reduce,
                      partition(in,
                                p,
                                sequence(0,tile_size,len),
                                sequence(1),
                                sequence(tile_size, tile_size, len+1)
                                )
                      )
                 );
}
```

(b) sequential compound codelet using adjacent tiling

```
#pragma tgm vector tag(kog)
int reduce(const Array<1,int> in) {
  __shared __tunable int vec_size;
  __shared Vector<1,int> vec(vec_size);
  __shared int tmp[vec_size];
  int len = in.size();
  int id = vec.id();
  tmp[id] = id < len ? in[id] : 0;
  int idle_len = 1;
  while(id >= idle_len) {
    tmp[id] += tmp[id-idle_len];
    idle_len *= 2;
  }
  if(id==0)
    return tmp[vector_size-1];
}
```

(c) vectorized atomic codelet

```
#pragma tgm tag(stride_tiled)
int reduce(const Array<1,int> in) {
  __tunable int p;
  int len = in.size();
  int tile_size= (len+p-1)/p;
  return reduce( map( reduce,
                      partition(in,
                                p,
                                sequence(0,1,p),
                                sequence(p),
                                sequence((p-1)*tile_size, 1, len+1)
                                )
                      )
                 );
}
```

(d) sequential compound codelet using strided tiling

Fig. 1. The spectrum of the reduce codelets contains two atomic implementations using different architectural components and two compound codelets that perform different tiling transformations

2.1 Codelet

Tangram adopts the codelet programming model [23] and extends it to facilitate performance portability, productivity, and maintainability. A *codelet* in Tangram is a coarse-grained code segment. Multiple codelets share the same name and function signature if they offer equivalent functionality for the current program. These codelets form a *spectrum*. Each codelet in a spectrum can be implemented either using different algorithms or the same algorithm but with different optimization techniques. Substituting one codelet for another in the same spectrum is considered a transformation of the program.

There are two kinds of codelets. *Atomic* codelets do not invoke other codelets in their implementations, while *compound* codelets do. Compound codelets may not be a scheduling quantum in execution, unlike the original codelet execution model. Leaving the decision of scheduling to either runtime or compiler allows the implementation to adapt across architectures. Multiple independent instances of an atomic codelet can form a *kernel*, which is similar to an OpenCL kernel. Furthermore, before scheduling is determined, a compound codelet can be decimated to multiple atomic codelets. The decimated codelets can then be fused. Fig. 1 shows an example of four codelets that implement the reduction pattern. The atomic codelet, shown in Fig. 1(a), performs the reduction sequen-

tially, while the compound codelets, shown in Fig. 1(b) and (d), partition the workload into multiple reductions and then performs another level of reduction.

To guarantee that each compound codelet can be transformed, each codelet spectrum contains at least one atomic codelet or a compound codelet invoking a spectrum which can terminate. Although a spectrum can be compiled to an unbounded set of possible codes due to recursion, in practice, the transformation step is pruned based on architectural limitations such as depth of the memory hierarchy and resource constraints.

2.2 Vectorized Codelet

A codelet can be also categorized based on its granularity. A *sequential* codelet executes sequentially. A *vectorized* codelet, on the other hand, is executed by a vector unit in semantic lock-step. The vectorized codelets are designed to describe a fine-grained communication pattern for vectorization, such as data shuffling across vector lanes. These codelets enlarge the design space by making use of the vector units prevalent in modern multi- and many-core architectures. Fig. 1(c) shows a vectorized codelet for the reduction where a tree-structure communication pattern, using the Kogge-Stone algorithm [11], is implemented.

Note that a vectorized codelet *without* communication across vector lanes is considered a group of sequential codelets with a particular scheduling, thus is regarded as redundant and is eliminated during compilation.

Tangram abstracts the architectural details, allowing vector length to be independent of the execution architecture constraints. Multiple instances, or parts of a vectorized codelet, can therefore be scheduled on an architectural vector unit.

2.3 Annotation and Qualifier

Sequential and vectorized codelets are distinguished by an annotation: `vector`. In a vectorized codelet, a `__shared` qualifier is used to label shared data across vector lanes, similar to OpenCL. Since codelets can be overloaded, a user may use a `tag` annotation to distinguish between them. The `tag` annotation is purely syntactical with no semantic meaning, but it may be used to differentiate between codelets during debugging. In Fig. 1(c), for example, the tag `tag(kog)` is used to document the fact that the codelet implements a Kogge-Stone reduction tree. The `env` annotation is used to label either architectural- or algorithmic-specific codelets, and provide strong hints to the compiler which can be used while synthesizing the kernel. Tuning-friendly qualifiers, such as `__tunable` and `__fixed`, can be used to label variables that are parameterized based on the granularity of the scheduling, the tiling factor, vector size, and cache line size of the target architecture.

2.4 Container

High-level languages provide containers to represent arrays. Containers allow programmers to reflect their intentions more clearly and avoid common bugs

found when using conventional linearized arrays. The container object also aids the compiler, since consecutive accesses along a high dimension can be inferred and proper optimizations applied.

Tangram codelets only accept scalars or containers as parameters. A wrapper function, which is not a codelet, is a user-defined function containing conventional linearized arrays and converters which map the linearized arrays to Tangram containers. In Tangram, conventional linearized arrays in a wrapper function are always assumed to not alias for consistency with sequential semantics in parallel execution.

2.5 Primitive

Tangram provides loop and data manipulating primitives which operate on containers. Table 1 lists the primitives and their definitions. Both `map` and `partition` are used to build compound codelets. The `map` primitive indicates independent workloads, which can be scheduled in parallel. The `partition` primitive indicates a data usage pattern which can be applied for better sharing or caching in parallel computing.

Unlike most library-based high-level languages, Tangram does not provide computational primitives, such as histogram or FFT. One main reason is that properties of a computational primitive might vary across applications or domains. A reduction, for example, in one domain may maintain both associativity and commutativity, while it may only maintain associativity in others. For a library designer, choosing these computational patterns and generality is not an easy task. Therefore, Tangram lets users build their own computational patterns using built-in loop and data manipulating primitives.

Another reason is that Tangram is not a pure library-based high-level language. Since high-level languages often rely on libraries which are commonly missing in emerging architectures, these languages tend to not be able to perform well and users are required to hand optimize until a new version of the underlying library is released or the language is updated. Therefore, Tangram focuses on supporting performance portability without relying on these traditional computational primitives. Further, Tangram potentially can become an internal library-building language for other library-based high-level languages.

3 Features

This section introduces the main features of Tangram, including transformation, performance tuning, performance portability, productivity, and maintainability.

3.1 Transformation

Compared to traditional compilers, which require recognition of loop structures through sophisticated analyses on flattened data structures, Tangram's use of

Table 1. Primitive examples in Tangram

Name	Definition
map(f,c)	given a function (or codelet) f and container c, the result is a container where f is applied to each element in c
partition(c,n,*start*,*end*,*inc*)	an input container c is split into n containers starting from a sequence *start* and ending at a sequence *end* with increment as a sequence *inc*, the result is a container of containers
zip($c1$,$c2$)	given two input containers of the same length, the output is a container of ($c1_i$, $c2_i$) pairs
iter_product($c1$,$c2$)	given two input containers, the output is the Cartesian product of the containers
sequence(*start*,*end*,*inc*)	outputs a container of integer sequence from *start* to *end* with increment *inc*
sequence(c)	outputs an infinite integer sequence of the constant c
container_sequence(c,*start*,*end*,*inc*)	given an integer container c with indices from *start* to *end* with increment *inc*, the result is a sequence converted from the container.

containers facilitates easy reasoning about transformations such as tiling, vectorization, privatization, blocking, coarsening, and thread-level parallelization [20]. The transformations are internally implemented within the `map` and `partition` primitives. Data layout transformation [22], locality-aware transformations, and scheduling can be achieved using strong locality hints implied by containers. Tiling exploiting architecture-specific memory, such as scratchpad or texture memory, can also be chosen during compilation.

Complicated transformations that are traditionally difficult in compiler frameworks can be described by programmers using substitute codelets. These transformations may include change of access pattern, application-specific transformations, or algorithm choices. Also, architecture- or environment-specific primitives can also be applied in substitute codelets with the `env` annotation. For example, a shuffle instructions in NVIDIA Kepler GPUs, SSE intrinsics in CPUs, or reduction primitives in MPI can be applied during synthesis in Tangram.

3.2 Performance Tuning and Portability

There are three tuning dimensions in Tangram. First, switching codelets can dramatically expand design spaces, compared to common high-level languages. For example, transforming a sequential to a vectorized codelet potentially can provide a finer-grained workload, which is infeasible in most refactoring tools. Furthermore, the `env` annotation enables tuning for specific environments or architectures while isolating the logic from these tuning parameters. Second, general loop transformations as described in the previous section can be tailored with proper granularities. Third, the `__tunable` qualifier allows strong hints for parameterization from programmers, enabling use of adaptive-grained codelets. It also provides a smoother tuning curve than the coarsening techniques commonly used in OpenCL compilers [8, 9, 16, 17], since the coarsening techniques tend to

serialize independent workloads originally scheduled in parallel, while Tangram has more flexible adjustment to schedule workloads based on the `__tunable` qualifier.

A tuning-friendly programming language with rich design spaces in multiple architectures potentially can achieve high performance across different architectures. Therefore, performance portability in Tangram is achieved by retuning the program to another architecture. Specialized intrinsics or built-in primitives of a given new architecture can be easily introduced in Tangram by adding substitute codelets if necessary.

3.3 Productivity

Tangram, like most high-level languages, provides abstractions to reduce application complexity. Also, unlike OpenCL, Tangram benefits from codelet reuse within a kernel and even across kernels. Considering performance portability, OpenCL may require kernel rewriting to effectively adapt to different architectures.

3.4 Maintainability

Tangram is designed with user debugging in mind. Since optimizations are decoupled from logic, users express high-level intent rather than coding architectural details, which may introduce bugs and limit portability. Since codelets within a spectrum are supposed to be functionally equivalent, codelet substitution can be used for verification. Output difference from substitute codelets potentially would only arise due to bugs in the codelet implementation. Small variations, such as rounding errors, may happen among substitute codelets, and tolerance can be introduced during testing.

Building substitute codelets in Tangram can be considered as incremental improvement for a program. Since substitute codelets provide the same functionality, one codelet per spectrum is the only necessary part to generate correct results. Other languages may require programmers to rewrite the kernel to exploit a new architectural feature. Tangram only requires adding substitute codelets that materializes the feature. In this process, minimal code changes are required to adapt to new architectures.

Even when the applied algorithm varies in a given program, codelets might still be reusable across algorithms. Also, specialized intrinsics or primitives for an architecture can be introduced by adding new substitute codelets. These would propagate up to the compound codelets garnering benefits.

4 Example and Evaluation

Tangram is currently implemented in C++ using a directive-based approach, with the compiler implemented using Clang as a source-to-source compiler. Access pattern analyses in containers consider only stride-one accesses in all dimensions. A kernel can be synthesized by inlining multiple instances of one codelet

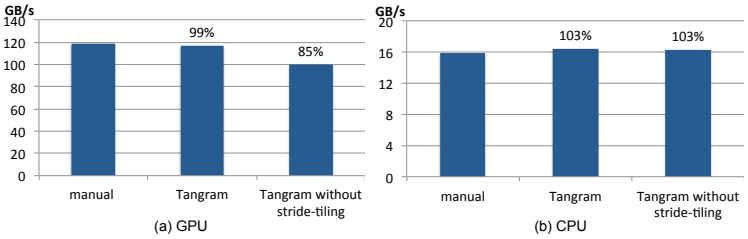

Fig. 2. Performance of 64M-integer Sum Reduction

with a map primitive. Codes across different domains are evaluated to demonstrate the features of Tangram. Tangram currently works in a shared memory model, and in terms of output source codes, Tangram currently supports C with OpenMP for a multicore CPU, and CUDA for NVIDIA GPUs. All evaluations are performed on a NVIDIA Tesla C2050 GPU and a quad-core Intel Xeon E5520 2.2Hz CPU, with -O3 flags in gcc 4.4.5, nvcc 4.2, and icc 13.0.1.

4.1 Reduction

Reduction is a simple example to illustrate Tangram codelets and the transformations performed. In Reduction, four substitute codelets, shown in Fig. 1, are written to represent the spectrum. Fig. 2(a) and (b) show synthesized results against manually optimized Reduction. Tangram can achieve almost 100% of performance against the manually optimized program in both architectures. For GPUs, Tangram selects a four-level hierarchical reduction, which is an adjacent-tiled reduction, shown in Fig. 1(b), containing stride-tiled reductions, shown in Fig. 1(d), as both inner and outer reductions. In each stride-tiled reduction, a vectorized and a sequential reduction form outer and inner reductions, respectively. On the other hand, for CPUs, Tangram selects a two-level hierarchical reduction, which is an adjacent-tiled reduction containing sequential reductions as both inner and outer reductions. Particularly, the adjacent-tiled reduction is parallel-friendly for multi-processors or multi-cores, and the stride-tiled reduction is vectorization-friendly for GPUs due to its stride-one memory access.

We also evaluate the results by removing the stride-tiled compound codelet from the spectrum. Tangram is robust enough to find a alternative kernel and achieves up to 85% of performance against the manually optimized code on GPUs. This demonstrates robustness of both built-in general loop transformation and codelet switching.

4.2 DGEMM

Double-precision General Matrix Multiplication (DGEMM) represents a common pattern of computation in scientific computing. It is chosen to demonstrate robustness of built-in tiling and coarsening transformation using high-level abstraction with containers. Fig. 3(a) shows synthesized GPU results against DGEMM in cuBLAS library, hand-optimized and naive implementations in the Parboil benchmark suite [21]. Tangram achieves up to 93.6% of performance against the well-tuned cuBLAS code, and outperforms the manually optimized and naive versions in Parboil by factors of 1.6× and 8.3×, respectively.

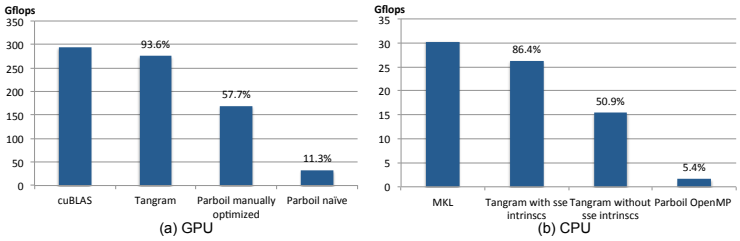

Fig. 3. Performance of DGEMM with two 2K-by-2K matrices

Fig. 3(b) shows the corresponding Tangram CPU results against (multi-threaded) DGEMM in MKL library, and the OpenMP implementation in Parboil. With an extra SSE-intrinsic codelet, Tangram achieves up to 86.4% of performance against the well-tuned MKL DGEMM and outperforms the OpenMP version by a factor of 15.9×. Without the SSE-intrinsic codelet, Tangram shows 50.9% of performance against MKL and outperforms the naive OpenMP version by a factor of 9.3×. This shows that incremental improvement by introducing SSE intrinsics in the codelets dramatically boost the performance, while codelets without SSE intrinsics still gain reasonable benefits from tiling and blocking in Tangram compiler, and auto-vectorization in icc.

4.3 SpMV

SpMV (Sparse Matrix-Vector multiplication) is one of the most important building blocks in sparse linear algebra. It is chosen to demonstrate how containers can simplify analyses. Two SpMV versions with CSR (Compressed Sparse Row) and ELL (ELLPACK) formats are implemented to demonstrate cache selection and data placement, instead of automatic SpMV format selection. Implementing two versions can evaluate code reuse in a software development process. The only codelet that is different (out of 7) between CSR and ELL SpMV is shown in Fig. 4. Both CSR and ELL SpMV codelets compute sparse dot products using the same codelet (`spvv_dotproduct`). However, due to different matrix formats, CSR and ELL SpMV codelets have different data partitions for accessing two containers holding matrix entries and column indices. Most information of memory accesses in CSR SpMV is data dependent and cannot be analyzed by the compiler due to indirect accesses, but abstraction of CSR SpMV still indicates critical information in `sequence(1)`, which means adjacent memory accesses within an instance (a row in this case) of a map primitive. In ELL, abstraction provides a critical hint in `sequence(...,1,...)`, which means adjacent memory accesses among adjacent instances of the map primitive. Tangram uses this information to synthesize different SpMV kernels, though both share almost the same codelets.

Tangram examines the lambda function (in the first two lines) and discerns that all instances of function applications share the same container `in_vec`. This implies that the data of `in_vec` should be cached and shared across the computation to garner a shorter latency and a higher bandwidth. Data may have irregular or indirect accesses, which can be inferred by analyzing the used codelet (`spvv_dotproduct`). Architectures may provide memory structures to support

```
#pragma tgm
Array<1, float> spmv_csr(…)
{
  return map([](const Array<1, float> val, const Array<1, int> idx )->float{
               return spvv_dotproduct(val,idx,in_vec);},
            zip( partition(in_M,
                           dim,
                           container_sequence(in_ptr,0,1, in_ptr.size()-1),
                           sequence(1), // <-locality
                           container_sequence(in_ptr,1,1, in_ptr.size())),
                 partition(in_idx,
                           dim,
                           container_sequence(in_ptr,0,1, in_ptr.size()-1) ,
                           sequence(1), // <-locality
                           container_sequence(in_ptr,1,1, in_ptr.size()))));
}
```

(a) SpMV CSR codelet

```
#pragma tgm
Array<1, float> spmv_ell(…)
{
  return map([](const Array<1, float> val, const Array<1, int> idx )->float{
               return spvv_dotproduct(val,idx,in_vec);},
            zip( partition(in_M,
                           dim,
                           sequence(0,1,cols_per_row),  // <- locality
                           sequence(cols_per_row),
                           sequence((cols_per_row-1)*dim,1, cols_per_row*dim)), // <-locality
                 partition(in_idx,
                           dim,
                           sequence(0,1,cols_per_row),  // <-locality
                           sequence(cols_per_row),
                           sequence((cols_per_row-1)*dim,1, cols_per_row*dim)))); // <-locality
}
```

(b) SpMV ELL codelet

Fig. 4. Codelets of SpMV

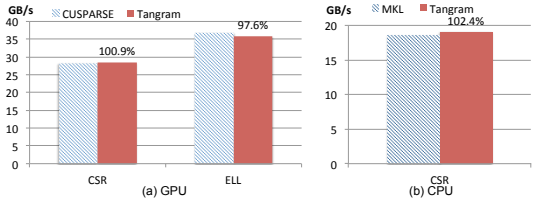

Fig. 5. Performance of SpMV with BCSSTK18 matrix [1]

this irregular accesses. In this case, texture memory in GPUs potentially can be applied.

Fig. 5(a) and (b) show generated Tangram results against CUSparse and MKL libraries. Tangram can achieve almost 100% of performance against the existing libraries.

5 Current Limitation and Future Work

As shown in Section 4.1 and 4.2, a spectrum of codelets might impact performance of a program. Building a complete and robust spectrum may require expert's efforts or thorough understanding of applications. It might therefore be challenging to a novice to come up with a minimal set of codelets that cover the algorithm and result in speeds that compete with the optimal libraries.

Redundant codelets within a spectrum might reduce maintainability and increase complexity of autotuning. Redundant codelet removal could alleviate such overhead using techniques such as AST-level preprocessing.

The current autotuning process is built in Tangram. Future work would introduce a description language for autotuning, so this would allow Tangram to either interact with other autotuning frameworks or support new architectures easily.

6 Related Work

Tangram is inspired by multiple high-level languages [2, 5, 7, 15]. Triolet [15] for example provides a similar language design based on list comprehension, but it adopts Python, instead of C++, and does not allow fine-tuned modification of the implementation syntax as can be done using codelets.

DSLs [4, 12–14] potentially can achieve all features Tangram can provide in their own domains. However, they are not general to cover applications from other domains. Rewriting-based languages [2] or libraries [13] have similar abilities to Tangram. They heavily rely on rewriting rules to seek performance gains. Tangram keeps balanced between rewriting rules and traditional optimizations, including loop transformations, thread scheduling, granularity adaptation, and data placement, and then achieves both high performance and productivity.

Conventional OpenCL compilers [6, 8–10, 16–19] rely on code patterns and use sophisticated analyses and transformations to optimize for specific architectures. Tangram inherits a certain degree of those transformations but simplifies analyses using abstractions and containers. PEPPHER [3] maintains multiple implementations of kernels across architectures, while Tangram maintains both architecture-dependent and architecture-independent codelets to synthesize kernels across architectures.

7 Conclusion

We present Tangram, a general-purpose, high-level language to support future parallel programming on multi-/many-core architectures. It achieves high performance across multiple architectures and provides productivity and maintainability by separating architecture-specific optimizations from computational pattern design, and embracing tuning-friendly interface, abstract containers, and codelet substitution. Tangram's synthesized programs are comparable in performance to existing well-optimized counterparts on both CPUs and GPUs.

8 Acknowledgements

This work is partly supported by the Starnet Center for Future Architecture Research (C-FAR), and the UIUC CUDA Center of Excellence.

References

1. The Matrix Market. `http://math.nist.gov/MatrixMarket/`
2. Ansel, J., et al.: PetaBricks: A language and compiler for algorithmic choice. In: Proceedings of the 2009 ACM SIGPLAN Conference on Programming Language Design and Implementation. pp. 38–49 (2009)
3. Benkner, S., et al.: PEPPHER: Efficient and productive usage of hybrid computing systems. IEEE Micro 31(5), 28–41 (2011)

4. Brown, K.J., et al.: A heterogeneous parallel framework for domain-specific languages. In: Parallel Architectures and Compilation Techniques, 2011 International Conference on. pp. 89–100 (2011)
5. Catanzaro, B., Garland, M., Keutzer, K.: Copperhead: compiling an embedded data parallel language. In: ACM SIGPLAN Notices. vol. 46, pp. 47–56 (2011)
6. Gummaraju, J., et al.: Twin peaks: a software platform for heterogeneous computing on general-purpose and graphics processors. In: Proceedings of the 19th International Conference on Parallel Architectures and Compilation Techniques. pp. 205–216 (2010)
7. Hoberock, J., Bell, N.: Thrust: A parallel template library. Online at http://thrust.googlecode. com 42, 43 (2010)
8. Jääskeläinen, P., et al.: pocl: A performance-portable opencl implementation. International Journal of Parallel Programming pp. 1–34 (2014)
9. Karrenberg, R., Hack, S.: Improving performance of OpenCL on CPUs. In: Compiler Construction. pp. 1–20 (2012)
10. Kim, J., Seo, S., Lee, J., Nah, J., Jo, G., Lee, J.: SnuCL: an OpenCL framework for heterogeneous CPU/GPU clusters. In: Proceedings of the 26th ACM International Conference on Supercomputing. pp. 341–352 (2012)
11. Kogge, P.M., et al.: A parallel algorithm for the efficient solution of a general class of recurrence equations. Computers, IEEE Transactions on 22(8), 786–793 (1973)
12. Membarth, R., et al.: Generating device-specific GPU code for local operators in medical imaging. In: Parallel and Distributed Processing Symposium, 2012 IEEE 26th International. pp. 569–581 (2012)
13. Puschel, M., et al.: SPIRAL: Code generation for DSP transforms. Proceedings of the IEEE 93(2), 232–275 (2005)
14. Ragan-Kelley, J., et al.: Halide: a language and compiler for optimizing parallelism, locality, and recomputation in image processing pipelines. ACM SIGPLAN Notices 48(6), 519–530 (2013)
15. Rodrigues, C., Jablin, T., Dakkak, A., Hwu, W.M.: Triolet: a programming system that unifies algorithmic skeleton interfaces for high-performance cluster computing. In: Proceedings of the 19th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming. pp. 247–258 (2014)
16. Rotem, N.: Intel OpenCL implicit vectorization module (2011)
17. Stratton, J.A., Kim, H.S., Jablin, T.B., Hwu, W.M.W.: Performance portability in accelerated parallel kernels. Tech. rep. (2013)
18. Stratton, J.A., et al.: MCUDA: An efficient implementation of CUDA kernels for multi-core CPUs. In: Languages and Compilers for Parallel Computing, pp. 16–30 (2008)
19. Stratton, J.A., et al.: Efficient compilation of fine-grained SPMD-threaded programs for multicore CPUs. In: Proceedings of the 8th annual IEEE/ACM International Symposium on Code Generation and Optimization. pp. 111–119 (2010)
20. Stratton, J.A., et al.: Algorithm and data optimization techniques for scaling to massively threaded systems. Computer 45(8), 26–32 (2012)
21. Stratton, J.A., et al.: Parboil: A revised benchmark suite for scientific and commercial throughput computing. Tech. rep., University of Illinois at Urbana-Champaign, Center for Reliable and High-Performance Computing (2012)
22. Sung, I.J., et al.: Data layout transformation exploiting memory-level parallelism in structured grid many-core applications. In: Proceedings of the 19th International Conference on Parallel Architectures and Compilation Techniques (2010)
23. Zuckerman, S., et al.: Position paper: Using a codelet program execution model for exascale machines. In: EXADAPT Workshop (2011)