Efficient Kernel Synthesis for Performance Portable Programming

Li-Wen Chang*, Izzat El Hajj*, Christopher Rodrigues†, Juan Gómez-Luna‡ and Wen-mei Hwu*
*University of Illinois at Urbana-Champaign
†Huawei America Research Lab
‡Universidad de Córdoba
{lchang20, elhajj2}@illinois.edu, christopher.rodrigues@huawei.com, el1goluj@uco.es, w-hwu@illinois.edu

***Abstract*—The diversity of microarchitecture designs in heterogeneous computing systems allows programs to achieve high performance and energy efficiency, but results in substantial software re-development cost for each type or generation of hardware. To mitigate this cost, a performance portable programming system is required.**

One fundamental difference between architectures that makes performance portability challenging is the hierarchical organization of their computing elements. To address this challenge, we introduce TANGRAM, a kernel synthesis framework that composes architecture-neutral computations and composition rules into high-performance kernels customized for different architectural hierarchies.

TANGRAM is based on an extensible architectural model that can be used to specify a variety of architectures. This model is coupled with a generic design space exploration and composition algorithm that can generate multiple composition plans for any specified architecture. A custom code generator then compiles these plans for the target architecture while performing various optimizations such as data placement and tuning.

We show that code synthesized by TANGRAM for different types and generations of devices achieves no less than 70% of the performance of highly optimized vendor libraries such as Intel MKL and NVIDIA CUBLAS/CUSPARSE.

I. Introduction

Heterogeneity is becoming ubiquitous in modern computing systems, ranging from low-power mobile devices to high-performance supercomputers. Ideally, applications for heterogeneous systems should exhibit performance portability. That is, they should achieve high performance on different device architectures and on existing and future generations without software re-development.

Various techniques have been developed in portability-focused tools to automatically apply architecture-specific optimizations to architecture-neutral code. OpenCL compilers apply varying levels of coarsening [1], [2], [3], [4], [5] and locality-centric scheduling [5] to achieve good performance for data-parallel workloads on CPU architectures. Data placement tools [6], [7] automatically assign data structures to the most suitable type of memory depending on the characteristics of the memory subsystem. Autotuning [8], [9], [10], [11] finds optimal parameter settings to adapt programs to resource constraints that vary across architectures such as cache sizes and occupancy.

978-1-5090-3508-3/16/$31.00

These portability techniques can be very effective, but they are often limited by the fact that their source languages assume some particular architectural hierarchy. For example, OpenCL dictates a fixed hierarchical arrangement (work-items and work-groups). Using more (or fewer) levels with OpenCL is laborious for the programmer and hard for the compiler. For this reason, we propose an architecture-neutral programming model that is oblivious to the hierarchy of target architectures. In this model, programmers write computations and composition rules which are then synthesized by our generic composition framework into architecture-specific composition plans based on a device specification.

While programming based on composition rules has been proposed by other languages [12], [13], [14], it has often been focused on algorithmic choice, adaptation to varying input characteristics, or even adaptation to architectural hierarchies. Existing methodologies mainly relied on the high-quality implementation of base rules, like calling existing high-performance libraries internally. Unlike those existing methodologies, we take advantage of this programming model to explore the design space of architectural optimizations for code synthesis, by assigning computations to different levels of the architectural hierarchy based on the computational capabilities at each level.

In this paper, we present the design, implementation, and evaluation of TANGRAM, a kernel synthesis framework that generates highly optimized architecture-specific kernels from generic and reusable code fragments. At the core of TANGRAM's approach is an extensible architectural model that can be used to specify a variety of architectures. The TANGRAM language allows users to express architecture-neutral computations and composition rules in terms of composable, interchangeable, and tunable building blocks called codelets. Then, for any specified architecture, TANGRAM's generic composition algorithm explores the design space to generate multiple composition plans while pruning the search. Each composition plan can then be compiled to the target architecture by an architecture-specific code generator that applies various optimizations such as data placement and parameter tuning.

Figure 1 shows the flow of TANGRAM framework and also the organization of this paper. We make the following contributions:

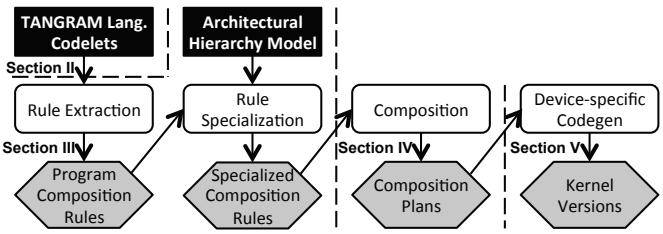


Fig. 1. Organization of the TANGRAM Framework

- We present a programming language (Section II) that supports specification of architecture-neutral computations and composition rules.
- We define a simple architectural hierarchy model that can be used to specify different architectures with different hierarchical organizations, and show how this model is useful in generating architecture-specific composition rules from our architecture-neutral language (Section III).
- We design a generic composition algorithm (Section IV) that can be used to compose architecture-specific kernels based on our abstractions.
- We implement a holistic kernel synthesis framework that leverages our generic composition algorithm, and couples it with other portability techniques during code generation and optimization (Section V) such as data placement and tuning, to synthesize highly optimized processor-specific kernels.
- We demonstrate that kernels synthesized from the same description achieve 70% performance (in the worst observed case) to multiple times performance compared to vendor hand-tuned data-parallel libraries (Section VI).

II. TANGRAM Language

A. Language Design

The TANGRAM language is designed with the following objectives:

1) Express equivalent computations interchangeably to expose algorithmic choice
2) Express compositions and computations interchangeably to enable variable levels of composition that best fit the device level specifications of the architectural hierarchy model
3) Express data parallelism
4) Ease the analysis of data flow and memory access patterns and the transformation of memory accessing logic
5) Provide tuning knobs

The programming model is built around *spectrums* and *codelets*. A *spectrum* represents a unique computation with a defined set of inputs, outputs, and side effects. A *codelet* represents a specific implementation of a spectrum. A spectrum can have many codelets that implement it. These codelets all have the same name and function signature, but can be implemented using different algorithms or the same algorithm with different optimization techniques. The interchangeability of the codelets in a spectrum serves Objective 1.

Codelets are classified into *compound* and *atomic* codelets. Compound codelets compose work by invoking primitives and other spectrums (including their own). Atomic codelets are self-contained: they compute without further decomposing work or invoking other spectrums. The interchangeability of compound and atomic codelets serves Objective 2. Atomic codelets are classified into *autonomous* and *cooperative* codelets. Computations in autonomous codelets are oblivious to other lanes in the same data parallel computation, whereas computations in cooperative codelets can explicitly exchange data with other lanes.

The TANGRAM language is an extension of C++ summarized in Table I. The `__codelet` qualifier is used to designate function declarations as spectrums and function definitions as codelets. The `__coop` qualifier labels cooperative codelets and the `__shared` qualifier labels data structures that are shared by all lanes of a cooperative codelet.[1] The `__tag` qualifier is optional and used to distinguish different codelets with the same function signature in debugging. The `__env` qualifier enables the user to write device-specific codelets. It is not intended as the main usage model but is included for completeness. The results we report do not use this feature, but we provide users with the option of using it if they wish (particularly if they want to write device-specific intrinsics or assembly).

The language comes with several built-in primitives. A `map` primitive is used to express data parallelism by applying a codelet to all elements of a data container, serving Objective 3. The `partition` primitive is used to express the pattern used for data partitioning. The three `sequence` primitives are used as arguments to `partition` to express different patterns. These primitives along with the `Array` container and the `__mutable` qualifier are used to facilitate memory access and data flow analysis serving Objective 4. The `__tunable` qualifier labels parameters that the compiler can tune serving Objective 5.

B. Codelet Examples

Figure 2 shows an example of four codelets implementing a spectrum for computing a summation. All codelets have the same function signature and are marked with the `__codelet` qualifier. Figure 2(a) shows an atomic autonomous codelet where each lane performs the summation sequentially. Figure 2(b) shows an atomic cooperative codelet with the `__coop` qualifier that performs a tree-based summation among lanes of parallel execution. The codelet contains multiple variables and arrays that are shared across the lanes and are marked with the `__shared` qualifier. Special functions `coopIdx()` and `coopDim()` are used to obtain the lane ID and the width of the cooperative codelet respectively.

Figures 2(c) and 2(d) show compound codelets using different tiling strategies. They both contain a tunable variable `p` that controls the number of partitions in the recursive call. In

[1]The `__shared` qualifier is different from the shared memory in CUDA. The data structure can be placed in global memory, shared memory in CUDA, or registers with shuffle instructions in SSE/AVX/CUDA.

Qualifiers	__codelet	Designates that a function declaration is a spectrum, or a function definition is a codelet
	__coop	Designates that a codelet is a cooperative codelet
	__tag	Designates the codelet's tag (optional, useful for debugging)
	__env	If a codelet is specific to a particular device(s), designates which device(s)
	__mutable	Designates that a variable or container is mutable (everything is immutable by default)
	__tunable	Designates that a variable can be tuned
	__shared	Designates that a variable is shared across lanes of a cooperative codelet (for cooperative codelets only)
Primitives	map(*f*, *c*)	Returns a container where each element results from applying spectrum *f* to each element in container *c*
	partition(*c*, *n*, *start*, *inc*, *end*)	Returns *n* sub-containers c_i of *c* where c_i goes from *start[i]* to *end[i]* with increment *inc[i]*
	sequence(*a*)	Returns an integer sequence of the value of *a* (argument to partition)
	sequence(*start*, *inc*, *end*)	Returns a sequence of integers from *start* to *end* with increment *inc* (argument to partition)
	sequence(*c*, *start*, *inc*, *end*)	Returns a sequence of integers from values of *c* at indexes *start* to *end* with increment *inc* (arg. to partition)
	Array<*n*, *type*>	A *n*-dimensional container of values of type *type*

TABLE I
QUALIFIERS, PRIMITIVES, AND CONTAINERS OF THE TANGRAM LANGUAGE

```
__codelet
int sum(const Array<1,int> in) {
  unsigned len = in.size();
  int accum = 0;
  for(unsigned i=0; i < len; ++i) {
    accum += in[i];
  }
  return accum;
}
```

(a) Atomic autonomous codelet

```
__codelet __coop __tag(kog)
int sum(const Array<1,int> in) {
  __shared int tmp[coopDim()];
  unsigned len = in.size();
  unsigned id = coopIdx();
  tmp[id] = (id < len)? in[id] : 0;
  for(unsigned s=1; s<coopDim(); s *= 2) {
    if(id >= s)
      tmp[id] += tmp[id - s];
  }
  return tmp[coopDim()-1];
}
```

(b) Atomic cooperative codelet

```
__codelet  __tag(asso_tiled)
int sum(const Array<1,int> in) {
  __tunable unsigned p;
  unsigned len = in.size();
  unsigned tile = (len+p-1)/p;
  return sum( map( sum, partition(in,
      p,sequence(0,tile,len),sequence(1),sequence(tile,tile,len+1))));
}
```

(c) Compound codelet using adjacent tiling

```
__codelet __tag(stride_tiled)
int sum(const Array<1,int> in) {
  __tunable unsigned p;
  unsigned len = in.size();
  unsigned tile = (len+p-1)/p;
  return sum( map( sum, partition(in,
      p,sequence(0,1,p),sequence(p),sequence((p-1)*tile,1,len+1))));
}
```

(d) Compound codelet using strided tiling

Fig. 2. Codelet examples for `sum` spectrum

Figure 2(c), the array is partitioned into adjacent contiguous tiles that start `tile` elements apart and have an internal stride of one. Such a partitioning is suitable for distributing data to workers with different caches such as different CPU threads or GPU thread blocks. In Figure 2(d), the array is partitioned into interleaved tiles that are staggered to start one element apart and have an internal stride of `p`. Such a partitioning is suitable for distributing data to workers that execute together and in the same cache such as CPU vector lanes or GPU threads.

Each codelet is intended to represent a fundamentally different algorithm or composition. Thus, it is unlikely that there will be more than a handful of atomic codelets in each spectrum. We do not expect the number of codelets to increase significantly over the lifetime of the code base.

III. FROM ARCHITECTURA HIERARCHYL MODEL TO COMPOSITION RULES

This section defines our architectural hierarchy model and abstract composition rules (Section III-A), and shows how these abstractions are used to extract program composition rules from codelets (Section III-B). It then describes how architectures are specified using our model and how the composition rules are specialized for different architectures (Section III-C).

A. Architectural Hierarchy Model and Abstract Rules

TANGRAM's approach builds on the observation that a key differentiating factor between devices is their architectural hierarchy. Different devices come with different architectural levels. For example, CPUs may be modeled as two-level devices (process, thread)[2] while GPUs may be modeled as three-level devices (grid, block, thread).

Each architectural level may have the ability to execute scalar or vector code. Such a computational capability is represented by C in Figure 3. Furthermore, each level of the hierarchy having a level beneath it has the capability to synchronize across the elements of that level. For example, a process can undergo barrier synchronization among all its threads. The subordinate level and synchronization capability are denoted by (ℓ, S).

[2]The CPU SIMD unit is omitted for brevity and clarity, but discussed later in Section III-C3.

Architectural Hierarchy Model:

$L := C_L , (\ell_L , S_L)$

L	:	level
C	:	computational capability (possible values: *SE* – scalar execution, *VE* – vector execution)
(ℓ , S)	:	(subordinate level , capability to synchronize subordinate level)

Abstract Composition Rules:

Select:	*compose(s , L)*	→	*compose(c , L)*	// *s* : spectrum, *c* : codelet of spectrum *s*
Compute:	*compose(c, L)*	→	*compute(c , C_L)*	// *c* : atomic codelet
Devolve:	*compose(s, L)*	→	*S_L , devolve(ℓ_L) , compose(s, ℓ_L)*	// *s* : spectrum
Cascade:	*compose(f(g(…), L)*	→	*compose(g(…), L) , compose(f(…), L)*	// *f, g* : primitives or spectrum invocations
Regroup:	*compose(partition(…, p), L)*	→	*S_L , regroup(p, L)*	// *p* : a partitioning scheme
Distribute:	*compose(map(s, …), L)*	→	*distribute(ℓ_L) , compose(s, ℓ_L)*	// *s* : spectrum

Fig. 3. Architectural Hierarchy Model and Abstract Composition Rules

Program Composition Rules: (for the sum example)

Rule 1:	*compose(sum , L)*	→	*S_L , devolve(ℓ_L) , compose(sum, ℓ_L)*	// Derived from Devolve
Rule 2:	*compose(sum , L)*	→	*compute(c_a , SE_L)*	// Derived from codelet a (c_a)
Rule 3:	*compose(sum , L)*	→	*compute (c_b , VE_L)*	// Derived from codelet b (c_b)
Rule 4:	*compose(sum , L)*	→	*S_L , regroup(p_c, L) , distribute(ℓ_L) , compose(sum, ℓ_L) , compose(sum, L)*	// Derived from codelet c (c_c)
Rule 5:	*compose(sum , L)*	→	*S_L , regroup(p_d, L) , distribute(ℓ_L) , compose(sum, ℓ_L) , compose(sum, L)*	// Derived from codelet d (c_d)

Example for Deriving Composition Rules from Compound Codelets: (using codelet c as an example)

compose(sum , L)	→	*compose(c_c , L)*	// Rule: Select
	→	*compose(sum(map(sum, partition(…, p_c))), L)*	// Expand c_c
	→	*compose(map(sum, partition(…, p_c)) , L) , compose(sum, L)*	// Rule: Cascade
	→	*compose(partition(…, p_c) , L) , compose(map(sum, …) , L) , compose(sum , L)*	// Rule: Cascade
	→	*S_L , regroup(p_c, L) , distribute(ℓ_L) , compose(sum, ℓ_L) , compose(sum, L)*	// Rules: Regroup, Distribute

Fig. 4. Extracted Spectrum Composition Rules

Based on this architectural hierarchy model, we define six abstract composition rules that are used to extract program composition rules from the codelets:

1) Select: A spectrum is composed at a level by selecting a codelet of that spectrum and composing it at that level.
2) Compute: An atomic codelet is composed at a level by assigning that atomic codelet to that level's computational capability (if possible).
3) Devolve: A spectrum is composed at a level by synchronizing then delegating the spectrum to a single worker of that level's subordinate level. For example, a master thread may perform a task on behalf of all threads in a process. Commas ',' on the right-hand side of a rule represent a concatenation of two code fragments. To keep the examples concise, we do not represent the flow of data between code fragments in the notation.
4) Cascade: Cascaded primitive or spectrum invocations are composed at a level by composing them sequentially at that level.
5) Regroup: A `partition` primitive is composed at a level by synchronizing to ensure the data is ready, then regrouping the data at that level according to the partitioning scheme.
6) Distribute: A `map` primitive is composed at a level by spawning multiple workers of the subordinate level and distributing the spectrum to those workers.

In practice, the `map` primitive is transformed with two additional rules: Serialize and Split. Serialize creates a loop at the current level (if it has a computational capability) to serialize the `map` operation at that level. Split breaks the `map` into a composition of two `maps` to extend the reach of the `map` to a lower subordinate level. Due to space constraints, we omit these two rules from the examples in this section.

B. Program Composition Rule Extraction

The first step in TANGRAM's flow is to extract the composition rules of a program by applying the abstract rules to the codelets until none can be applied deterministically anymore. One composition rule is extracted per codelet. Figure 4 shows the rules extracted from the codelets in Figure 2. Rule 1 is basically the Devolve rule. Rules 2 and 3 show how atomic codelets generate rules that assign those codelets to computational capabilities (autonomous to scalar, cooperative to vector). Rules 4 and 5 show how compound codelets generate more complex rules corresponding to their functionality. Rule 1 is extracted from the devolve abstract rule which requires no codelets.

An example of how rules are extracted from a compound codelet is also shown in Figure 4 for codelet *c*. In the resulting rule, the data is first regrouped according to the scheme

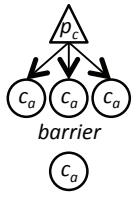
Device Specification:

P := C_P = none, (ℓ_P , S_P) = (T , barrier/join) // P : process

T := $C_T = SE_T$, (ℓ_T , S_T) = none // T : thread

Specialized Composition Rules:

P rules:	P1:	compose(sum , P)	→	S_P , devolve(T) , compose(sum, T)
	P4:	compose(sum , P)	→	S_P , regroup(p_c, P) , distribute(T) , compose(sum, T) , compose(sum, P)
	P5:	compose(sum , P)	→	S_P , regroup(p_d, P) , distribute(T) , compose(sum, T) , compose(sum, P)
T rules:	T2:	compose(sum , T)	→	compute(c_a , SE_T)



Composition Example:

compose(sum , P) → S_P , regroup(p_c, P) , distribute(T) , compose(sum, T) , compose(sum, P) // P4

→ S_P , regroup(p_c, P) , distribute(T) , compute(c_a , SE_T) , S_P , devolve(T) , compose(sum, T) // T2 , P1

→ S_P , regroup(p_c, P) , distribute(T) , compute(c_a , SE_T) , S_P , devolve(T) , compute(c_a , SE_T) // T2

Fig. 5. Rule Specialization and Composition (CPU Example). For the composition plan diagram on the right side, a triangle represents distribution of work according to the partition pattern from the indicated codelet and a circle represents scalar compute according to the indicated codelet.

specified in codelet c (p_c). The inner `sum` (parameter to the `map` primitive) is then distributed to multiple workers of the subordinate level. Finally, the outer `sum` is performed at the original level.

We note that up until this point, the composition rules have been generated from the codelets without any consideration for the target device. This indicates the architecture-neutrality of the programming model.

C. Composition Rule Specialization

In this subsection, we show how architectures are specified and how the program composition rules are specialized for those architectures. We use a CPU and a GPU example to assist with the explanation.

1) CPU Example: Figure 5 shows an example of how a basic CPU can be specified and how the composition rules can be specialized for the specified architecture. In this example, the CPU is treated as a two-level device (without SIMD units): the first level being the process (P) and the second being the thread (T). The process does not have a computational capability, but has subordinate threads and the ability to synchronize those threads via a barrier/join operation. The thread has scalar execution capability and has no subordinate levels.

Based on this architecture specification, the extracted rules 1 through 5 in Figure 4 can be specialized for the CPU architecture. For the process level, only rules 1, 4, and 5 can be specialized. Rules 2 and 3 cannot because the process does not have a compute capability; it can only distribute work to its subordinate (thread) level. For the thread level, only rule 2 can be specialized. Rules 1, 4, and 5 cannot because the thread does not have a subordinate level in the specification, and rule 3 cannot because the thread does not have SIMD units (vector computational capability) needed to execute cooperative codelets. The resulting specialized rules are P1, P4, and P5 for the process level and T2 for the thread level in Figure 5.

To assist the reader with understanding the application of these rules, Figure 5 also shows an example of one possible composition plan that they can be used to derive. This plan takes three steps to create. First, P4 is applied to distribute the `sum` to the different threads and then to sum up the partial sums at the process level. Next, T2 is used to perform the `sum` in each thread. Also, because the process cannot perform computations, P1 is used to delegate one of its threads to perform the final `sum` of partial sums. Finally, T2 is used to perform that delegated `sum` using a single thread. The created composition plan is illustrated with the diagram on the right side of Figure 5. The final code generated from this composition plan is shown in Figure 10 which will be discussed later in Section V. The actual composition algorithm to generate this and other composition plans is discussed in Section IV.

2) GPU Example: Figure 6 shows an example of how a basic GPU can be specified and how the composition rules can be specialized. In this example, the GPU is treated as a three-level device. The grid (G) level has no computational capability, but has a subordinate block level (B) and can perform a barrier synchronization across blocks via kernel termination and launch of a new kernel. The block level has a vector execution capability, a thread (T) subordinate level, and can synchronize subordinate threads using `__syncthreads()`. Finally, the thread level has scalar execution capability and no subordinate level.

Similar to the CPU example, rules 1, 4, and 5 can only be assigned to levels with subordinate levels, rule 2 to levels with scalar execution capability, and rule 3 to levels with vector execution capability. Accordingly, we can specialize rules 1, 4, and 5 for the grid level, rules 1, 3, 4, and 5 for the block level, and rule 2 for the thread level.

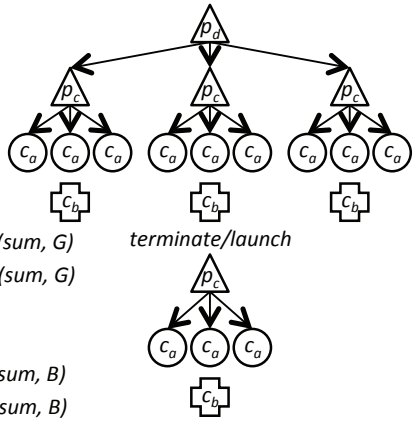
Figure 6 shows an example of one possible composition plan that these specialized rules can be used to create. We omit the detailed explanation of this composition for brevity since the process is similar to that shown in the CPU example. The final code generated from this composition is shown in

Device Specification:

$G := C_G = none,\ (\ell_G, S_G) = (B, terminate/launch)$ // G : grid

$B := C_B = VE_B,\ (\ell_B, S_B) = (T, \_\_syncthreads())$ // B : block

$T := C_T = SE_T,\ (\ell_T, S_T) = none$ // T : thread



Specialized Composition Rules:

G rules: G1: $compose(sum, G) \rightarrow S_G, devolve(B), compose(sum, B)$

G4: $compose(sum, G) \rightarrow S_G, regroup(p_c, G), distribute(B), compose(sum, B), compose(sum, G)$

G5: $compose(sum, G) \rightarrow S_G, regroup(p_d, G), distribute(B), compose(sum, B), compose(sum, G)$

B rules: B1: $compose(sum, B) \rightarrow S_B, devolve(T), compose(sum, T)$

B3: $compose(sum, B) \rightarrow compute(c_b, VE_B)$

B4: $compose(sum, B) \rightarrow S_B, regroup(p_c, B), distribute(T), compose(sum, T), compose(sum, B)$

B5: $compose(sum, B) \rightarrow S_B, regroup(p_d, B), distribute(T), compose(sum, T), compose(sum, B)$

T rules:T2: $compose(sum, T) \rightarrow compute(c_a, SE_T)$

Composition Example:

$compose(sum, G)$

$\rightarrow S_G, regroup(p_c, G), distribute(B), compose(sum, B), compose(sum, G)$ // G4

$\rightarrow S_G, regroup(p_c, G), distribute(B), S_B, regroup(p_d, B), distribute(T), compose(sum, T), compose(sum, B), S_G, devolve(B), compose(sum, B)$ // B5 , G1

$\rightarrow S_G, regroup(p_c, G), distribute(B), S_B, regroup(p_d, B), distribute(T), compute(c_a, SE_T), compute(c_b, VE_B), S_G, devolve(B), S_B, regroup(p_d, B), distribute(T), compose(sum, T), compose(sum, B)$ // T2 , B3, B5

$\rightarrow S_G, regroup(p_c, G), distribute(B), S_B, regroup(p_d, B), distribute(T), compute(c_a, SE_T), compute(c_b, VE_B), S_G, devolve(B), S_B, regroup(p_d, B), distribute(T), compute(c_a, SE_T), compute(c_b, VE_B)$ // T2 , B3

Fig. 6. Rule Specialization and Composition (GPU Example). In addition to the triangles and circles introduced in the previous diagram, a cross represents vector compute according to the indicated codelet.

Figure 11 which will be discussed later in Section V.

3) Model Extensibility: To keep the previous examples simple, we have modeled CPUs as two-level devices and GPUs as three-level devices. However, the architectural model is extensible as we show in the following examples.

CPU SIMD Unit. The SIMD (vector) unit of a CPU can be added by giving the thread level vector execution capability as well as a subordinate level consisting of a vector lane with scalar execution capability.

GPU Warp. Warp-centric mapping [15] on GPUs can be achieved by treating warps as a separate level between blocks and threads and giving the warp level a vector execution capability. Doing so enables more optimal code generation for cooperative codelets featuring warp-centric mapping optimization techniques. Such techniques include avoiding the use of `__syncthreads()` as well as using shuffle instructions and registers instead of scratchpad memory to store shared data.

Instruction-level Parallelism (ILP). On both CPUs and GPUs, ILP can be achieved via a subordinate level to the thread level that executes a serialized `map` loop that is unrolled. In this case, the subordinate level to the thread is the iteration of an unrolled loop and the synchronization happens by closing the loop.

GPU Dynamic Parallelism. Dynamic parallelism on GPUs can be achieved by assigning the grid level as a subordinate level to the thread level. This enables threads to decompose work and delegate to subordinate grids through new kernel launches which creates a cycle in the architecture hierarchy. Optimizations [16] involving dynamic parallelism can be considered alternative choice of composition.

In our experiments, we model CPUs as four-level devices (process, thread with SIMD unit, SIMD lane, and ILP) and GPUs as five-level devices (grid, block, warp, thread, and ILP). We leave support for dynamic parallelism as future work.

D. Discussion

The proposed architectural model mainly focuses at the difference in the architectural hierarchies, and guides the synthesis of the algorithmic structure for a particular computation. In this sense, the proposed model might be too "coarse-grained" and be deficient in finer architectural details, such as resource sizes, including core numbers, SIMD width, cache sizes, etc. However, as brightly mentioned in Section I, the architectural hierarchies mainly determine the algorithmic structure, while the resource sizes mainly impact the fine-grained optimizations, such as data placement and parameter tuning. These fine-grained optimizations will be discussed in Section V. Most of these techniques exist in the literature and are not the major contribution of this paper.

IV. Creating Composition Plans

This section describes how the specialized rules are used by our generic composition algorithm to create composition plan candidates specific to the target architecture. This phase

s_0 is the spectrum subject to kernel synthesis
L is the top level in the device being targeted
N is the number of iterations to search
candidates(i) is the set of composition candidates at iteration *i*
rules(s, ℓ) is the set specialized rules for spectrum *s* at level *ℓ*
prune(rules, i) sorts and prunes *rules* for iteration *i*
prune(candidates, i) sorts and prunes *candidates* for iteration *i*

```
candidates(0) := { compose(s0, L) }
for iteration i from 1 to N do
  forall c ∈ candidates(i-1) do
    if no calls compose(s, ℓ) in c then
      candidates(i) ← c      // propagate complete candidates
    else
      forall compose(s, ℓ) in c do
        forall r in prune(rules(s, ℓ), i) do
          mark r as a candidate rule for compose(s, ℓ) in c
      B := all combinations of candidate rules for c
      forall b in B do
        candidates(i) ← c with all rules in b applied
  candidates(i) := prune(candidates(i), i)
```

Fig. 7. Composition Algorithm

determines the overall structure of the kernel including its hierarchical organization, work decomposition, and algorithmic choice. After this phase, the composition candidates are passed onto the code-generator which is described in Section V.

A. Composition Algorithm

The algorithm for creating composition plans for a spectrum targeting a specific architecture is shown in Figure 7. It begins by composing the spectrum at the top level in the architectural hierarchy then proceeds to explore the design space via a *breadth-first* search. At each point in the search space, if the candidate has no more *compose* invocations to expand, it is considered complete and is passed to the next iteration as is. Otherwise, the algorithm selects the set of rules to expand each invocation of *compose* and generates a new candidate for every combination of rules.

The search iterates for N iterations where N must be at least the number of architectural levels in order for the composition plans to reach the lowest level. Typically, N is a bit larger to enable a wider search.

B. Pruning

To avoid explosion of the search space, pruning takes place throughout the process when specialized rules are being selected as well as in between iterations. The prune function in Figure 7 sorts rules or candidates according to their expected benefit, and then drops the lowest ones. The strictness of pruning can be set by the user and determines how many candidates to keep or drop.

The pruning policy currently used is *parallelism first* whereby rules extracting more parallelism are prioritized. The criteria for comparing two rules according to their benefit is shown in Figure 8. Note that the compared rules (r_1 and r_2) are already specialized for the level (ℓ) before entering the composition process, so they are all applicable in this level. If

```
compare(r1, r2, ℓ, i):
    # Comparing rules r1 and r2 (of level ℓ) for composing at iteration i
    if i is not the last iteration then
        prefer(distributes)
        if r1 distributes and r2 distributes then
            prefer(partitioning matches ℓ)
        else      # neither distributes
            prefer(vectorizes)
    else          # i is the last iteration (distribute is undesirable)
        prefer(computes)
        if r1 computes and r2 computes then
            prefer(vectorizes)
    prefer(came from tunable codelet)
    return both are the same
```

prefer(cond): if one rule satisfies *cond* and the other doesn't, return the rule that does, otherwise continue with the execution

Fig. 8. Comparing Composition Rules for Pruning

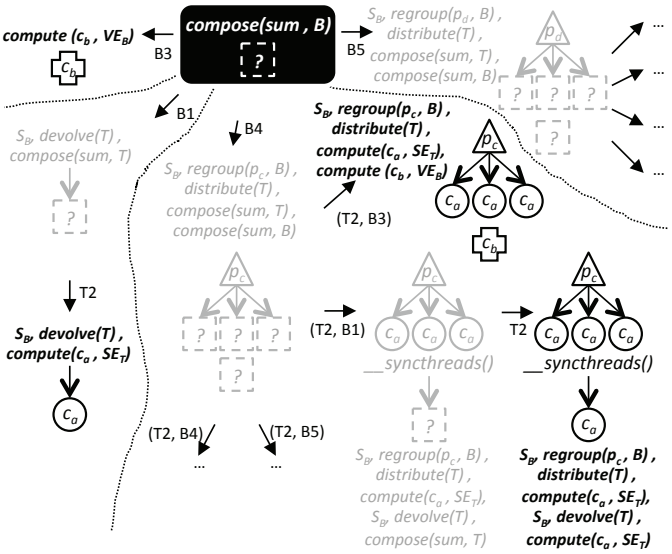


Fig. 9. Composition Plan Creation Example (Showing Four Possible Composition Plans for the Block and Thread Levels)

we are not in the last iteration, rules that generate a *distribute* are preferred because they extract more parallelism. Among rules that distribute, the partitioning schemes are analyzed and used to determine which rule has more favorable *locality* for the level in question. The preference of the level is determined by the architecture specification via an additional entry for each level that specifies whether it prefers adjacent or strided tiling. If neither rule distributes, then rules generating vector execution are preferred over those generating scalar execution. In the last iteration, rules that compute are preferred over those that distribute because there will be no more iterations to expand the distributed *compose* invocations. Among compute rules, those that generate vector execution are preferred. When all is equal, rules that come from codelets having tuning knobs are preferred because they give the compiler more optimization opportunities. Candidates' plans are compared via pairwise comparison of the rules applied to each.

S_P, regroup(p_c, P), distribute(T), compute(c_a, SE_T), S_P, devolve(T), compute(c_a, SE_T)

```
S_P                 :  // No sync needed at the beginning
regroup(p_c, P)     :  unsigned p_c = omp_get_num_threads();
regroup(p_c, P)     :  unsigned len_c = in_size;
regroup(p_c, P)     :  unsigned tile_c = (len_c+p-1)/p_c;
distribute(T)       :  #pragma omp parallel
distribute(T)       :  {
distribute(T)       :    unsigned j = omp_get_thread_num();
compute(c_a, SE_T)  :    unsigned len_a1 = tile;
compute(c_a, SE_T)  :    int accum_a1 = 0;
compute(c_a, SE_T)  :    for(int i = 0; i < len_a1; ++i) {
compute(c_a, SE_T)  :      accum_a1 += in[j*tile_c + i];
compute(c_a, SE_T)  :    }
compute(c_a, SE_T)  :    ret_a1[j] = accum_a1;
S_P                 :  } // Join omp threads
devolve(T)          :  // No spawn (only master executes)
compute(c_a, SE_T)  :  unsigned len_a2 = p;
compute(c_a, SE_T)  :  int accum_a2 = 0;
compute(c_a, SE_T)  :  for(int i = 0; i < len_a2; ++i)
compute(c_a, SE_T)  :    accum_a2 += ret_a1[i];
compute(c_a, SE_T)  :  ret_a2 = accum_a2;
```

Fig. 10. Codegen for CPU Example in Figure 5

C. GPU Composition Plan Example

An example of the composition process for the `sum` spectrum on a *single block* in a GPU is shown in Figure 9. This example demonstrates how our algorithm generates different possibilities from the same codelets. By applying B3, the `sum` is performed by the cooperative codelet. By applying B1 then T2, the entire `sum` is performed by a single thread in the block. By applying B4 then T2 and B3, the `sum` is distributed to the individual threads by the partition in codelet c, each thread then calculates partial sums, and then the partial sums are aggregated with the cooperative codelet. If instead of T2 and B3, we apply T2 and B1 then T2, the partial sums are aggregated by a single thread in the thread block.

V. Code Generation

After the composition plan candidates have been created, code is generated for the target architecture as shown in the examples in Figures 10 and 11. During code generation, TANGRAM built-in optimizations also take place such as data placement and parameter tuning. Most of these techniques exist in the literature and are not the main focus or contribution of this paper. We briefly explain how they are integrated into our system.

Data Placement. A data placement decision must be made for each data container in the composition candidate. A heuristic-based GPU data placement algorithm, similar to [7], is currently applied, but our framework can also support model-based data placement tools [6]. For CPUs, we have *copying* and *caching* data placement. While copying loads data into another local array in the beginning, caching directly accesses the data in the original data structure. Additionally, transposition can be further applied when copying data to a local array for a CPU and a scratchpad memory for a GPU, if necessary.

S_G, regroup(p_c, G), distribute(B), S_B, regroup(p_d, B), distribute(T), compute(c_a, SE_T), compute(c_b, VE_B) S_G, devolve(B), S_B, regroup(p_d, B), distribute(T), compute(c_a, SE_T), compute(c_b, VE_B)

First kernel

```
S_G                 :  // No sync needed at beginning
regroup(p_c, G)     :  unsigned p_c = gridDim.x;
regroup(p_c, G)     :  unsigned len_c = in_size;
regroup(p_c, G)     :  unsigned tile_c = (len_c+p_c-1)/p_c;
distribute(B)       :  unsigned k = blockIdx.x;
S_B                 :  // No sync needed at beginning
regroup(p_d, B)     :  unsigned p_d = blockDim.x;
regroup(p_d, B)     :  unsigned len_d = tile_c;
regroup(p_d, B)     :  unsigned tile_d = (len_d+p_d-1)/p_d;
distribute(T)       :  unsigned j = threadIdx.x;
compute(c_a, SE_T)  :  unsigned len_a = tile_d;
compute(c_a, SE_T)  :  int accum_a = 0;
compute(c_a, SE_T)  :  for(unsigned i=0; i < len_a; ++i) {
compute(c_a, SE_T)  :    accum_a += in[k*tile_c + j + p_d*i];
compute(c_a, SE_T)  :  }
compute(c_a, SE_T)  :  ret_a = accum_a;
compute(c_b, VE_B)  :  __shared__ int tmp[blockDim.x];
compute(c_b, VE_B)  :  unsigned len_b = p_d;
compute(c_b, VE_B)  :  unsigned id = threadIdx.x;
compute(c_b, VE_B)  :  tmp[id] = ret_a;
compute(c_b, VE_B)  :  __syncthreads();
compute(c_b, VE_B)  :  for(unsigned s=1; s<blockDim.x; s *= 2) {
compute(c_b, VE_B)  :    if(id >= s)
compute(c_b, VE_B)  :      tmp[id] += tmp[id - s];
compute(c_b, VE_B)  :    __syncthreads();
compute(c_b, VE_B)  :  }
compute(c_b, VE_B)  :  ret_b[k] = tmp[blockDim.x-1];
S_G                 :  return; // Terminate kernel
```

Second kernel

```
devolve(B)          :  if(blockIdx.x == 0)
S_B until end       :      ... // Similar to first kernel
```

Fig. 11. Codegen for GPU Example in Figure 6

Parameter Tuning. The tuning process determines the value for each variable marked with the `__tunable` qualifier. Tunable variables are usually SIMD unit widths or partition sizes. SIMD widths are determined from the architecture specification. Partition sizes after code generation become tile sizes, work-group sizes, and coarsening factors whose values depend on properties of each architectural level such as cache/scratchpad sizes and occupancy [17]. These properties can also be determined from the architecture specification. The tuning process can also produce multiple candidate kernels from each original candidate composition plan and use profiling methods to identify the best ones, as discussed below.

Candidate Selection. After candidate kernels are generated, they need to be profiled so that the best one is selected for execution. Selection can be done via offline profiling [8], [11], [14] or online profiling [18]. Pruning reduces the final number of candidates for the selection process into a reasonable amount. In the evaluation, we rely on offline profiling for benchmarks with regular memory accesses (because datasets can be automatically generated) and online profiling for benchmarks with irregular accesses.

Benchmark	Reference	Dataset	Number of Codelets (Input Code)
Scan	Thrust	A 16M integer array	4, with 2 exclusive scan and 4 reduction
SGEMV-TS	MKL & CUBLAS	A 512K-by-128 (Tall-and-Skinny) matrix	1, with 1 dot-product, and 2 reduction
SGEMV-SF	MKL & CUBLAS	A 128-by-512K (Short-and-fat) matrix	1, with 1 dot-product, and 2 reduction
DGEMM	MKL & CUBLAS	A non-transposed 4K-by-4K matrix & a transposed 4K-by-4K matrix	1, with 2 dot-product and 1 reduction
SpMV	MKL & CUSPARSE	`bcsstk18` [19] (CSR format)	1, with 1 sparse scalar-multiply, 1 sparse dot-product, and 4 reduction
KMeans	Rodinia	`kdd_cup` (default in Rodinia)	1, with 1 difference, 4 reduction, 1 minima selection, and 1 gemm-like operation
BFS	Rodinia	`graph1MW_6` (default in Rodinia)	1, with 1 edge visiting, and 1 vertex visiting

TABLE II
BENCHMARKS

VI. Evaluation

A. Setup

The TANGRAM language is implemented as an extension of C++. We modify Clang [20] 3.6 to support parsing TANGRAM's qualifiers. Containers and primitives are parsed as C++ template classes. Code generation is implemented as a Clang traversal of the AST which generates C, OpenMP, and CUDA kernel code. The generated kernels are then compiled using the Intel C compiler (icc) version 16.0.0, OpenMP version 4.0, and the NVIDIA CUDA compiler (nvcc) version 7.0 respectively. The compiled programs are evaluated on an i7-3820 Sandy Bridge CPU, a C2050 Fermi GPU, and a K20c Kepler GPU.

B. Benchmarks

Table II summaries the applications implemented in TANGRAM: `Scan`, `SGEMV` (with 2 datasets, called TS and SF), `DGEMM`, `SpMV`, `KMeans`, and `BFS`, and the corresponding datasets and numbers of codelets. We compare each of our generated kernels to a reference, or the best performing implementation available to us: Thrust [21] version 1.9, MKL [22] version 12.0, CUBLAS/CUSPARSE [23], [24] version 7.0, and and Rodinia [25] 3.0. In selecting the reference for each benchmark, we chose CUBLAS/CUSPARSE and Thrust for GPUs and MKL for CPUs where possible. Particularly, MKL, CUBLAS, and CUSPARSE come with their own offline tuning and then heuristic version selections for parameterization of different architectures. When using Rodinia, we chose the best known hand-optimized version from the benchmark suite. For example, for the Rodinia CPU references, we pick the best result among the OpenMP version (with icc -O3) and the OpenCL version on top of the Intel and AMD OpenCL CPU stacks.

For the benchmarks with regular memory access patterns and no data-dependent control flow, such as `Scan`, `SGEMV`, and `DGEMM`, we use offline profiling. For iterative applications, such as `SpMV` and `KMeans`, we apply online profiling using techniques similar to [18] and only profile the first iteration. For irregular but non-iterative applications, such as `BFS`, we use offline profiling on synthetic random graphs.

C. Performance Results

Figure 12 shows the performance of the six benchmarks on the three evaluation architectures, comparing the TANGRAM implementation to the reference implementation for each. In presenting the results, performance is normalized to the best performing implementation for each benchmark (highest bar), thereby showing the relative performance of the implementations being compared.

Scan. TANGRAM's `Scan` consistently outperforms Thrust's for all devices. TANGRAM's `Scan` is expressed with codelets of different simple scan algorithms, including sequential scan, tree-structure scan, and recursive scan. Particularly, each scan call in a recursive scan codelet can be mapped to different scan codelets to fit the architectural hierarchy and to further enable high performance portability. Given a target architecture, offline profiling can be used to select the best version from a range of competitive ones.

Besides selecting appropriate compositions, coarsening factors, and tiling factors, TANGRAM also benefits from fusing `map`s. `Scan` is commonly written as either a scan-scan-add or a reduce-scan-scan algorithm [26]. In either case, the middle scan lacks parallelism and blocks fusion. However, the three stages can be fused by implementing the middle scan in a streaming or sliding fashion using atomic operations [27], [28]. In TANGRAM, this sliding scan is implemented as a codelet in the scan spectrum using `map`, enabling TANGRAM to automatically generate a composition that fuses the three stages.

Figure 13 compares TANGRAM's scan to the reference with and without the sliding codelet (which enables fusion). Even without the extra codelet, TANGRAM still outperforms Thrust, due to better choice of `partition` parameters, which are labeled as tunable in TANGRAM. The addition of the sliding codelet has more impact on GPUs than CPUs because GPU cachelines have shorter lifetimes than CPU ones so applying fusion is more critical in order to avoid reloading intermediate data. Note that Thrust's CPU results[3] is 47.9-49.4x slower than TANGRAM's due to an inefficient CPU implementation in Thrust version 1.9.

SGEMV. Parallelism of `SGEMV` highly depends on the height of input matrix. Therefore, a tall-and-skinny (TS) matrix is used as a test case with high parallelism and a short-and-fat (SF) matrix as a test case with low parallelism.

In the TS matrix, TANGRAM's `SGEMV` surprisingly outperforms MKL's on the CPU by a factor of 2.18x while

[3]Note that Thrust CPU `Scan` is confirmed as multi-threaded.

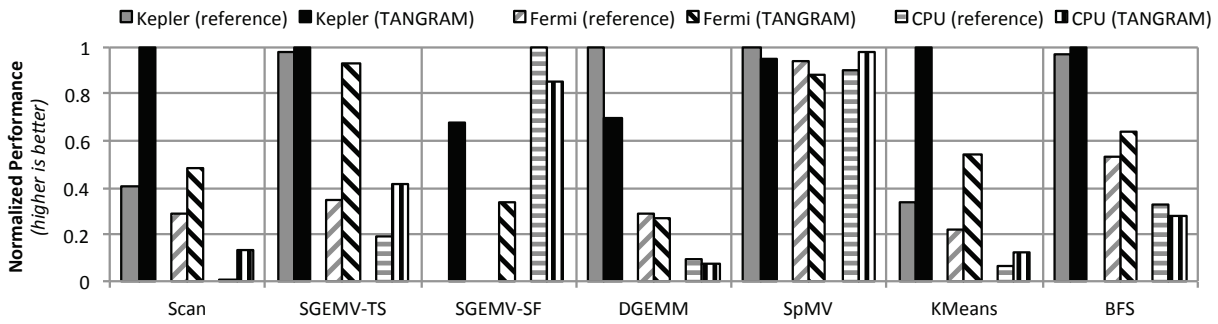


Fig. 12. TANGRAM Performance Results

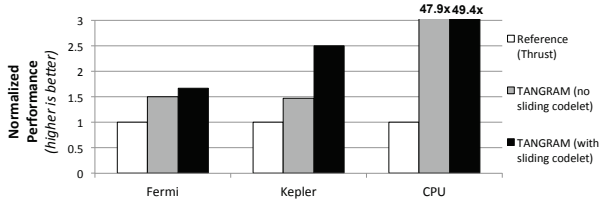


Fig. 13. `Scan` Results with or without the Sliding Codelet (Normalized to the Corresponding Thrust Results)

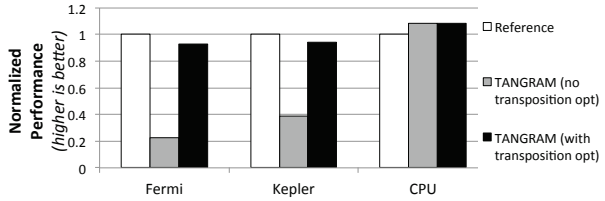


Fig. 15. `SpMV` Results with and without Transposition Optimization (Normalized to the Corresponding References)

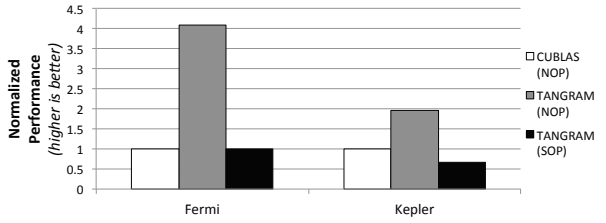


Fig. 14. SOP and NOP `SGEMV`-SF Results on GPUs (Normalized to the Corresponding CUBLAS Results)

delivering comparable performance to CUBLAS' on Kepler and outperforms CUBLAS' on Fermi by a factor of 2.69x. In this particular evaluation, since MKL's `SGEMV` delivers only less than half of the memory bandwidth, we believe it is mistuned. A similar conclusion is also applied to Fermi's `SGEMV`. This demonstrates that the current industry practice falls short in keeping performance critical libraries well tuned for each generation of hardware.

In the SF matrix, we discover[4] CUBLAS does not implement the "standard" `SGEMV` (denoted as SOP, sequential-order-preserving), which preserves the sequential order of the dot-product. A "non-standard" `SGEMV` (denoted as NOP, non-order-preserving) generates different rounding error from the SOP one, and might impact some applications. The SOP `SGEMV` only allows the dot-product following the sequential order, so NOP reduction codelets (like Figure 2 (b), (c), and (d) but using float) must be excluded. Two reduction codelets used in the SOP `SGEMV` are sequential reduction codelet (similar to Figure 2 (a)) and a sliding fashion [28] sequential reduction codelet. Therefore, the only difference between SOP and NOP is the reduction codelets used in the dot-product. This also shows the productivity of TANGRAM framework without kernel redevelopment.

Figure 14 compares the SOP and NOP `SGEMV` with the SF matrix on the GPUs. For NOP `SGEMV`, TANGRAM's code outperforms CUBLAS' on Kepler and Fermi by a factor of 1.97x and 4.09x respectively. It is worth mentioning that TANGRAM's SOP `SGEMV` has comparable performance of CUBLAS' NOP `SGEMV` on Fermi.

DGEMM. TANGRAM's `DGEMM` performs within 30% difference of all reference implementations. The presented TANGRAM CPU result is based on generic codelets, though, as mentioned in Section II, the TANGRAM framework allows easy integration of intrinsics through __env. The version using AVX intrinsics can gain another 7% performance improvement.

One difficulty in achieving good performance in `DGEMM` is that it is bounded by instruction throughput. The current implementation of TANGRAM relies on the backend C (icc) or CUDA compiler (nvcc) to generate good quality code. Therefore, our TANGRAM Kepler result (717 GFLOPS) achieves 70% performance of CUBLAS (1,027 GFLOPS). In the future, we will likely employ more optimization passes in our compiler and provide a code generation path to PTX or assembly code to better control these low-level factors. For example, one of such factors is register bank conflicts [29], which is hard to address at the source-code level.

SpMV. TANGRAM's `SpMV` delivers comparable performance (within 10%) to all reference implementations, doing

[4]by examining the output rounding errors of specially designed matrices

slightly worse than CUSPARSE on the GPUs, and slightly better than MKL on the CPU. In SpMV, two candidate kernels are generated for each architecture. Online profiling [18] is applied to the first iteration to select the best version, and the overhead of online profiling is less than 0.8%.

Traditional implementations [30], [31] only consider the warp-centric dot-product and the scalar dot-product. The former tends to have a better memory access pattern but less parallelism than the latter. Compared to the traditional implementations, TANGRAM explores more combination of compositions with built-in optimizations, such as transposition on GPU scratchpad memory.

Figure 15 shows how built-in optimizations impact the final performance. In this evaluation, the result with TANGRAM's optimizations is a version very similar to the scalar dot-product. As mentioned in Section IV, the TANGRAM's composition process prefer higher parallelism by preferring the rules that generate a *distribute*. Although the scalar dot-product might have a worse memory access pattern, the TANGRAM's optimizers can still improve its performance by applying proper optimizations, such as transposition on GPU scratchpad, or selecting a proper tile size for CPU caches. Particularly, TANGRAM's GPU implementation is similar to [32]. In the end, the results show the built-in optimizations significantly improve performance of SpMV in TANGRAM by up to 4.08x on GPUs. Note the CPU results are not sensitive to TANGRAM's optimizers, because CPUs tend to have larger caches to tolerate different tiling sizes.

KMeans. TANGRAM's KMeans consistently outperforms Rodinia's for all devices (the best performing CPU version among Rodinia implementations was OpenMP).

For KMeans, TANGRAM generates seven candidates (with different coarsening factors and data placements) for GPUs and four candidates for CPUs. The online profiling is applied to the first iteration, and the total overhead is less than 2%.

As mentioned in Section IV, TANGRAM applies the rules that have favorable locality among those generating a *distribute*, and consequently outperforms Rodinia's, all of which access the loops (the feature loop and the cluster loop) in a suboptimal order.

BFS. TANGRAM's BFS performs within 10% difference of all reference implementations, doing slightly better than Rodinia's on the GPU, and slightly worse on the CPU (the best performing CPU version of Rodinia BFS is the OpenCL version on top of Intel's stack). The evaluated BFS only includes the same vertex-based algorithm that Rodinia uses for fair comparison.

The vertex status checking (for g_graph_mask) and edge index fetching (for g_graph_nodes) of BFS are parallelizable. While Rodina's parallelizes the former one but serialize the latter one, TANGRAM's parallelizes both, since TANGRAM tends to apply a rule with higher parallelism.

D. Discussion

TANGRAM language enables expression for interchangeable codelets, allowing recursive calls to adapt different architectural hierarchies and tunable qualifiers for parameterization tuning to adapt resource sizes. Therefore, TANGRAM compiler potentially can choose alternative algorithms or optimizations for a particular computation to achieve better performance.

Algorithms. In our evaluation, TANGRAM's DGEMM, KMeans, and BFS use the same algorithms as the references. TANGRAM's SGEMV uses the same algorithm as the standard BLAS and MKL, while CUBLAS uses a different algorithm for the SF matrix. For Scan and SpMV, we cannot confirm whether the algorithms are the same, since the references are close-sourced. However, particularly for Scan, we believe TANGRAM synthesized a different combination of scan algorithms compared to Thrust.

Synthesized Kernels. Most differences among the synthesized kernels for different types of architectures (for example, CPUs and GPUs) are either algorithmic or loop structure due to different hierarchies. For the same type of architectures (for example, Fermi and Kepler GPUs), the differences mainly come from different parameters or data placement. The only exception happens in Scan: its algorithmic combination changes from Fermi to Kepler, due to the high efficiency of Kepler's shuffle instructions.

Reasons for High Performance. We summarize the major reasons why TANGRAM can achieve better or comparable performance to the references. TANGRAM potentially can deliver better algorithmic combination to match the architectural hierarchy (Scan), more parallelism (SGEMV, SpMV and BFS), better locality (Kmeans), better parameters (Scan) or better data placement (Kmeans and SpMV). For DGEMM, TANGRAM did not outperform MKL or CUBLAS, since the references are written in assembly.

VII. Related Work

Performance portability from a single source has been an area of great interest. **High-level languages** [33], [34] have been proposed for targeting CPUs and GPUs. Surge [33] provides collective primitives for the users, but does not support hierarchical composition of codelets like TANGRAM does. Steuwer et al. [34] proposed a language using built-in rewrite rules of map and reduce to generate high-performance BLAS routines, while TANGRAM enables user-defined composition rules through codelets.

Libraries of algorithms and data structures for heterogeneous computing such as Thrust [21] and PEPPHER [35], [36] have also been used to target CPUs and GPUs from a single API. However, libraries are limited to the library developer's ability to anticipate architectures and tune to them, and do not have TANGRAM's ability to automatically synthesize new kernels given new device specification.

Policy-based tuning on GPU [10] provides tunable coarsening and hierarchy mapping for different generations, but only works for the same fixed algorithms of kernels, while TANGRAM allows different algorithms for a given computation and can work on different numbers of hierarchies using recursive decomposition.

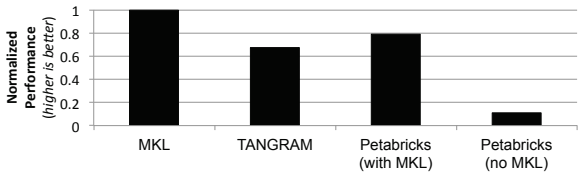


Fig. 16. Comparison between TANGRAM and Petabricks using DGEMM on CPU

Kim et al. [5] and Lee et al. [37] studied hierarchy mapping of nested parallel patterns for CPUs and GPUs respectively. Both can be considered as specialized cases of the composition process in TANGRAM. Neither of them considers different algorithms, or different hierarchies of architectures.

CHiLL [38] provides the capability of hierarchy mapping for loop transformations mainly on CPUs. TANGRAM encompasses a wider array of optimization techniques, including different algorithms.

Multiple domain libraries or languages [9], [11], [39], [40] employ similar composition processes. Delite [41] supports performance portability from a single source by providing a **metaprogramming** framework for creating domain-specific languages. TANGRAM is a general purpose language, and TANGRAM's device specification and static pruning process potentially can be also applied to them.

LMS [42] provides a metaprogramming framework to enable users to define custom rewriting rules. TANGRAM potentially can be implemented through LMS, and the concept of device specification and static pruning in TANGRAM can be also extended to the languages or libraries using LMS.

Languages [12], [13], [14] with composition rules can potentially provide functionality of adaptation to architectural hierarchy similar to what TANGRAM does. In contrast to their heavy reliance on the high quality implementation of base rules, TANGRAM can directly generate high-performance code.

A. Comparison to Petabricks

Petabricks [14] is the most similar work to TANGRAM, allowing the user to define **codelet**-like functions (called *transforms* and *rules*), supporting **composition** and parameter tuning, and trying to achieve performance portability on CPUs and GPUs [43].

The major difference between TANGRAM and Petabricks is **architectural optimization**. TANGRAM introduces architectural hierarchy models and corresponding rules to guide composition and optimization processes, and focuses on architectural optimizations themselves. Compared to TANGRAM, Petabricks directly relied on autotuning (using evolutionary algorithms, particularly) for design space search, and focused on task scheduling, and selection of proper algorithms or libraries for particular input data. Lack of architectural hierarchy models could obstruct possible composition and potential architectural optimization for the target architecture, then prevent exploration of certain versions, and possibly lead to a suboptimal result. Meanwhile, lack of general architectural optimizations could cause catastrophic performance degradation for generated code.

To demonstrate this difference, a common benchmark, DGEMM, is evaluated. Figure 16 shows Petabricks can achieve 79% and 11% of MKL performance, with and without calling MKL DGEMM internally[5] respectively, while TANGRAM can achieve 70% of MKL performance (without calling MKL DGEMM internally). These results imply that Petabricks highly relied on high-performance base *rules* (atomic codelets in TANGRAM). We also observe that the released package of Petabricks did not optimize function inlining, thread spawning, and branch divergences of version selection, so it achieved only 79% of MKL performance even with internal MKL DGEMM calls. This evaluation demonstrates architectural optimizations are crucial to achieve high performance.

Other important differences include TANGRAM's support for cooperative codelets, which is crucial for better utilization of SIMD execution on modern architectures. TANGRAM also introduces static pruning in composition to select competitive candidates before profiling and potentially can enable dynamic profiling in runtime for certain applications.

VIII. Conclusion

In this paper, we present TANGRAM, a kernel synthesis framework that supports performance portability through composition of user-defined architecture-neutral code into high-performance kernels customized for different architectural hierarchies. We provide comprehensive description from the architectural hierarchy model, the composition mechanism, to the code generation. Our results show that TANGRAM can achieve a performance of at least 70%, and in some cases multiple times, of various well-known reference implementations on various architectures.

Acknowledgment

This material is based upon work supported by the Department of Energy, National Nuclear Security Administration, under Award Number DE-NA0002374. This work is also supported by the Starnet Center for Future Architecture Research (C-FAR), the DoE Vancouver Project (DE-SC0005515): Designing a Next Generation Software Infrastructure for Heterogeneous Exascale Computing, the Huawei Project (YB2015120003): High Performance Algorithm Compilation for Heterogeneous Systems, and the NVIDIA GPU Center of Excellence at UIUC.

References

[1] N. Rotem, "Intel OpenCL implicit vectorization module," 2011.

[2] J. Gummaraju, L. Morichetti, M. Houston, B. Sander, B. R. Gaster, and B. Zheng, "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.

[5]The DGEMM in Petabricks calls MKL by default and disables all other rules. For fair comparison, we re-enable all of the rules and optionally enable MKL DGEMM.

[3] R. Karrenberg and S. Hack, "Improving Performance of OpenCL on CPUs," in *Proceedings of the 21st International Conference on Compiler Construction*, pp. 1–20, 2012.

[4] P. Jääskeläinen, C. S. de La Lama, E. Schnetter, K. Raiskila, J. Takala, and H. Berg, "pocl: A performance-portable opencl implementation," *International Journal of Parallel Programming*, vol. 43, no. 5, pp. 752–785, 2015.

[5] H.-S. Kim, I. El Hajj, J. Stratton, S. Lumetta, and W.-M. Hwu, "Locality-centric thread scheduling for bulk-synchronous programming models on CPU architectures," in *Proceedings of the 13th Annual IEEE/ACM International Symposium on Code Generation and Optimization*, pp. 257–268, 2015.

[6] G. Chen, B. Wu, D. Li, and X. Shen, "PORPLE: An extensible optimizer for portable data placement on GPU," in *Proceedings of the 47th Annual IEEE/ACM International Symposium on Microarchitecture*, pp. 88–100, 2014.

[7] B. Jang, D. Schaa, P. Mistry, and D. Kaeli, "Exploiting memory access patterns to improve memory performance in data-parallel architectures," *IEEE Trans. Parallel Distrib. Syst.*, vol. 22, no. 1, pp. 105–118, 2011.

[8] R. C. Whaley, A. Petitet, and J. J. Dongarra, "Automated empirical optimizations of software and the atlas project," *Parallel Computing*, vol. 27, no. 1, pp. 3–35, 2001.

[9] M. Püschel, J. M. Moura, B. Singer, J. Xiong, J. Johnson, D. Padua, M. Veloso, and R. W. Johnson, "Spiral: A generator for platform-adapted libraries of signal processing alogorithms," *International Journal of High Performance Computing Applications*, vol. 18, no. 1, pp. 21–45, 2004.

[10] D. Merrill, M. Garland, and A. Grimshaw, "Policy-based tuning for performance portability and library co-optimization," in *Innovative Parallel Computing*, pp. 1–10, 2012.

[11] J. Ragan-Kelley, C. Barnes, A. Adams, S. Paris, F. Durand, and S. Amarasinghe, "Halide: a language and compiler for optimizing parallelism, locality, and recomputation in image processing pipelines," *ACM SIGPLAN Notices*, vol. 48, no. 6, pp. 519–530, 2013.

[12] G. E. Blelloch, "NESL: A nested data-parallel language.(version 3.1)," tech. rep., DTIC Document, 1995.

[13] K. Fatahalian, D. R. Horn, T. J. Knight, L. Leem, M. Houston, J. Y. Park, M. Erez, M. Ren, A. Aiken, W. J. Dally, and P. Hanrahan, "Sequoia: Programming the memory hierarchy," in *Proceedings of the 2006 ACM/IEEE conference on Supercomputing*, ACM, 2006.

[14] J. Ansel, C. Chan, Y. L. Wong, M. Olszewski, Q. Zhao, A. Edelman, and S. Amarasinghe, "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.

[15] S. Hong, S. K. Kim, T. Oguntebi, and K. Olukotun, "Accelerating CUDA graph algorithms at maximum warp," in *ACM SIGPLAN Notices*, vol. 46, pp. 267–276, 2011.

[16] I. El Hajj, J. Gómez-Luna, C. Li, L.-W. Chang, D. Milojicic, and W. mei Hwu, "KLAP: Kernel launch aggregation and promotion for optimizing dynamic parallelism," in *Proceedings of the 49th Annual IEEE/ACM International Symposium on Microarchitecture*, 2016 (in press).

[17] NVIDIA, "CUDA C best practices guide v. 7.0," 2015.

[18] L.-W. Chang, H.-S. Kim, and W.-m. Hwu, "DySel: Lightweight dynamic selection for kernelbased data-parallel programming model," in *Proceedings of the Twenty-First International Conference on Architectural Support for Programming Languages and Operating Systems*, pp. 667–680, ACM, 2016.

[19] "The Matrix Market." http://math.nist.gov/MatrixMarket/.

[20] C. Lattner and V. Adve, "LLVM: A compilation framework for lifelong program analysis & transformation," in *Code Generation and Optimization, International Symposium on*, pp. 75–86, 2004.

[21] N. Bell and J. Hoberock, "Thrust: A productivity-oriented library for CUDA," *GPU Computing Gems Jade Edition*, p. 359, 2011.

[22] "Intel Math Kernel Library." http://software.intel.com/en-us/articles/intel-mkl/.

[23] NVIDIA, *CUBLAS Library User Guide*. NVIDIA, v7.0 ed., Oct. 2015.

[24] NVIDIA, *CUDA CUSPARSE Library*, Aug. 2015.

[25] S. Che, M. Boyer, J. Meng, D. Tarjan, J. W. Sheaffer, S.-H. Lee, and K. Skadron, "Rodinia: A benchmark suite for heterogeneous computing," in *Workload Characterization, 2009, IEEE International Symposium on*, pp. 44–54, 2009.

[26] Y. Dotsenko, N. K. Govindaraju, P.-P. Sloan, C. Boyd, and J. Manferdelli, "Fast scan algorithms on graphics processors," in *Proceedings of the 22Nd Annual International Conference on Supercomputing*, pp. 205–213, 2008.

[27] S. Yan, G. Long, and Y. Zhang, "StreamScan: Fast scan algorithms for GPUs without global barrier synchronization," in *Proceedings of the 18th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pp. 229–238, 2013.

[28] J. Gómez-Luna, L.-W. Chang, I.-J. Sung, W.-M. Hwu, and N. Guil, "In-place data sliding algorithms for many-core architectures," in *Parallel Processing, 2015 44th International Conference on*, pp. 210–219, IEEE, 2015.

[29] J. Lai and A. Seznec, "Performance upper bound analysis and optimization of sgemm on Fermi and Kepler GPUs," in *Code Generation and Optimization, 2013 IEEE/ACM International Symposium on*, pp. 1–10, 2013.

[30] N. Bell and M. Garland, "Efficient sparse matrix-vector multiplication on CUDA," NVIDIA Technical Report NVR-2008-004, NVIDIA Corporation, 2008.

[31] A. Danalis, G. Marin, C. McCurdy, J. S. Meredith, P. C. Roth, K. Spafford, V. Tipparaju, and J. S. Vetter, "The scalable heterogeneous computing (SHOC) benchmark suite," in *Proceedings of the 3rd Workshop on General-Purpose Computation on Graphics Processing Units*, pp. 63–74, 2010.

[32] J. L. Greathouse and M. Daga, "Efficient sparse matrix-vector multiplication on GPUs using the CSR storage format," in *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, pp. 769–780, IEEE, 2014.

[33] S. Muralidharan, M. Garland, B. Catanzaro, A. Sidelnik, and M. Hall, "A collection-oriented programming model for performance portability," in *Proceedings of the 20th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pp. 263–264, 2015.

[34] M. Steuwer, C. Fensch, S. Lindley, and C. Dubach, "Generating performance portable code using rewrite rules: from high-level functional expressions to high-performance OpenCL code," in *Proceedings of the 20th ACM SIGPLAN International Conference on Functional Programming*, pp. 205–217, 2015.

[35] S. Benkner, S. Pllana, J. L. Träf, P. Tsigas, U. Dolinsky, C. Augonnet, B. Bachmayer, C. Kessler, D. Moloney, and V. Osipov, "PEPPHER: Efficient and productive usage of hybrid computing systems," *IEEE Micro*, vol. 31, no. 5, pp. 28–41, 2011.

[36] U. Dastgeer, L. Li, and C. Kessler, "The PEPPHER composition tool: Performance-aware dynamic composition of applications for GPU-based systems," in *High Performance Computing, Networking, Storage and Analysis, 2012 SC Companion:*, pp. 711–720, 2012.

[37] H. Lee, K. J. Brown, A. K. Sujeeth, T. Rompf, and K. Olukotun, "Locality-aware mapping of nested parallel patterns on GPUs," in *Proceedings of the 47th Annual IEEE/ACM International Symposium on Microarchitecture*, pp. 63–74, IEEE Computer Society, 2014.

[38] C. Chen, J. Chame, and M. Hall, "CHiLL: A framework for composing high-level loop transformations," tech. rep., 2008.

[39] K. Goto and R. A. v. d. Geijn, "Anatomy of high-performance matrix multiplication," *ACM Transactions on Mathematical Software*, vol. 34, pp. 12:1–12:25, May 2008.

[40] D. Merrill, "CUB:kernel-level software reuse and library design," in *GPU Technology Conference Presentation*, 2013.

[41] A. K. Sujeeth, K. J. Brown, H. Lee, T. Rompf, H. Chafi, M. Odersky, and K. Olukotun, "Delite: A compiler architecture for performance-oriented embedded domain-specific languages," *ACM Trans. Embed. Comput. Syst.*, vol. 13, no. 4s, pp. 134:1–134:25, 2014.

[42] T. Rompf and M. Odersky, "Lightweight modular staging: A pragmatic approach to runtime code generation and compiled DSLs," in *Proceedings of the Ninth International Conference on Generative Programming and Component Engineering*, pp. 127–136, 2010.

[43] P. M. Phothilimthana, J. Ansel, J. Ragan-Kelley, and S. Amarasinghe, "Portable performance on heterogeneous architectures," in *Proceedings of the Eighteenth International Conference on Architectural Support for Programming Languages and Operating Systems*, vol. 48, pp. 431–444, ACM, 2013.