TANGRAM
Efficient Kernel Synthesis for Performance Portable Programming

Li-Wen Chang¹, Izzat El Hajj¹, Christopher Rodrigues², Juan Gómez-Luna³, Wen-Mei Hwu¹
¹University of Illinois, ²Huawei America Research Lab, ³Universidad de Córdoba
lchang20@illinois.edu
Performance Portability

• Maintaining optimized programs for different devices is costly

• Ideally, programs written once should run difference devices with performance
Performance Portability: OpenCL SGEMM

- Tesla GPU (GTX 280)
- Fermi GPU (C2050)
- Sandy Bridge CPU (i7-3820)

- Parboil (default naïve OpenCL version)
- Parboil (OpenCL version optimized for Tesla GPU)
- Reference (MKL for CPU, CUBLAS for GPU)
Composition-based Programming Language

• NESL, Sequoia, Petabricks

• Highly adaptive to hierarchies
  – Through composition

• Usually scaling well

• Performance relies on base-rule implementations/libraries
Performance Sensitivity in Base Rule: DGEMM

Normalized Performance
(higher is better)

- MKL
- TANGRAM
- Petabricks (with MKL)
- Petabricks (no MKL)

20% loss
11% of MKL

Sandy Bridge i7-3820
TANGRAM

• Composition-based language
• Focus at high-performance code synthesis within a node
  – Remove dependence of high-performance base-rule implementations/libraries
• Provide a representation for better SIMD utilization
• Provide an architectural hierarchy model to guide composition
TANGRAM Language

(a) Atomic autonomous codelet

```c
__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;
}
```

(b) Atomic cooperative codelet

```c
__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;
}
```

(c) Compound codelet using adjacent tiling

```c
__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))));
}
```

(d) Compound codelet using strided tiling

```c
__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))));
}
```
TANGRAM Language

(a) Atomic autonomous codelet

```c
__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;
}
```

(b) Atomic cooperative codelet

```c
__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];
}
```

(c) Compound codelet using adjacent tiling

```c
__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))));
}
```

(d) Compound codelet using strided tiling

```c
__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))));
}```
TANGRAM Language

(a) Atomic autonomous codelet

```cpp
__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;
}
```

(b) Atomic cooperative codelet

```cpp
__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];
}
```

(c) Compound codelet using adjacent tiling

```cpp
__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))));
}
```

(d) Compound codelet using strided tiling

```cpp
__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))));
}
```
TANGRAM Language

(a) Atomic autonomous codelet

```cpp
__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;
}
```

(b) Atomic cooperative codelet

```cpp
__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];
}
```

(c) Compound codelet using adjacent tiling

```cpp
__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))));
}
```

(d) Compound codelet using strided tiling

```cpp
__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))));
}
```
TANGRAM Language

(a) Atomic autonomous codelet

```c
__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;
}
```

(b) Atomic cooperative codelet

```c
__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];
}
```

(c) Compound codelet using adjacent tiling

```c
__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))));
}
```

(d) Compound codelet using strided tiling
(a) Atomic autonomous codelet

```c
__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;
}
```

(b) Atomic cooperative codelet

```c
__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];
}
```

(c) Compound codelet using adjacent tiling

```c
__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)) ) );
}
```

(d) Compound codelet using strided tiling

```c
__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)) ) );
}
```
TANGRAM Language

(a) Atomic autonomous codelet

```cpp
__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;
}
```

(b) Atomic cooperative codelet

```cpp
__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;
}
```

(c) Compound codelet using adjacent tiling

```cpp
__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))));
}
```

(d) Compound codelet using strided tiling

```cpp
__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))));
}```
Rule Extraction

• TANGRAM parser
  – Clang 3.5
  – Customized TANGRAM AST builder

• Output a set of TANGRAM ASTs

Program Composition Rules: (sum)
Rule 1:  \( \text{compose}(\text{sum}, L) \rightarrow S_L, \text{devolve}(\mathcal{E}_L), \text{compose}(\text{sum}, \mathcal{E}_L) \)
Rule 2:  \( \text{compose}(\text{sum}, L) \rightarrow \text{compute}(c_a, SE_L) \)
Rule 3:  \( \text{compose}(\text{sum}, L) \rightarrow \text{compute}(c_b, VE_L) \)
Rule 4:  \( \text{compose}(\text{sum}, L) \rightarrow S_L, \text{regroup}(p_o L), \text{distribute}(\mathcal{E}_L), \text{compose}(\text{sum}, \mathcal{E}_L), \text{compose}(\text{sum}, L) \)
Rule 5:  \( \text{compose}(\text{sum}, L) \rightarrow S_L, \text{regroup}(p_o L), \text{distribute}(\mathcal{E}_L), \text{compose}(\text{sum}, \mathcal{E}_L), \text{compose}(\text{sum}, L) \)

Example for Deriving Composition Rules from Compound Codelets: (codelet c)
\( \text{compose}(\text{sum}, L) \rightarrow \text{compose}(c_c, L) \)
\( \rightarrow \text{compose}(\text{sum}(\text{map}(\text{sum}, \text{partition}(\ldots, p_j)), L) \)
\( \rightarrow \text{compose}(\text{map}(\text{sum}, \text{partition}(\ldots, p_j)), \text{compose}(\text{sum}, L) \)
\( \rightarrow \text{compose}(\text{partition}(\ldots, p_j), L) \), \text{compose}(\text{map}(\text{sum}, \ldots), L) \), \text{compose}(\text{sum}, L) \)
\( \rightarrow S_L, \text{regroup}(p_o L), \text{distribute}(\mathcal{E}_L), \text{compose}(\text{sum}, \mathcal{E}_L), \text{compose}(\text{sum}, L) \)
Architectural Hierarchy Model

- Define a “level”
  - Computational capability
    - Scalar or vector execution
    - Capability to synchronize across the subordinate level of that level

**Device Specification:**

\[
G := C_G = \text{none} , \ ( \ell_G, S_G ) = ( B, \text{terminate/launch} ) \quad // \ G : \text{grid}
\]

\[
B := C_B = \text{VE}_B , \ ( \ell_B, S_B ) = ( T, \_\_ \text{syncthreads}() ) \quad // \ B : \text{block}
\]

\[
T := C_T = \text{SE}_T , \ ( \ell_T, S_T ) = \text{none} \quad // \ T : \text{thread}
\]

- Extensible
  - CPU SIMD, GPU warp, ILP, even GPU dynamic parallelism
Rule Specialization

- TANGRAM analyzer
  - AST traverser
- Output a lookup table
  - Legal codelets for each level
  - Also prioritize them

**Specialized Composition Rules:**

**G rules:**
- G1: $\text{compose}(\text{sum}, G) \rightarrow S_G, \text{devolve}(B), \text{compose}(\text{sum}, B)$
- G4: $\text{compose}(\text{sum}, G) \rightarrow S_G, \text{regroup}(p_c, G), \text{distribute}(B), \text{compose}(\text{sum}, B), \text{compose}(\text{sum}, G)$
- G5: $\text{compose}(\text{sum}, G) \rightarrow S_G, \text{regroup}(p_d, G), \text{distribute}(B), \text{compose}(\text{sum}, B), \text{compose}(\text{sum}, G)$

**B rules:**
- B1: $\text{compose}(\text{sum}, B) \rightarrow S_B, \text{devolve}(T), \text{compose}(\text{sum}, T)$
- B3: $\text{compose}(\text{sum}, B) \rightarrow \text{compute}(c_p, VE_B)$
- B4: $\text{compose}(\text{sum}, B) \rightarrow S_B, \text{regroup}(p_c, B), \text{distribute}(T), \text{compose}(\text{sum}, T), \text{compose}(\text{sum}, B)$
- B5: $\text{compose}(\text{sum}, B) \rightarrow S_B, \text{regroup}(p_d, B), \text{distribute}(T), \text{compose}(\text{sum}, T), \text{compose}(\text{sum}, B)$

**T rules:**
- T2: $\text{compose}(\text{sum}, T) \rightarrow \text{compute}(c_a, SE_T)$
Composition

• TANGRAM planner
  – AST traverser/builder
  – Selection of codelets or map policies
  – Pruning

• Output ASTs for codegen
Codegen

• TANGRAM codegen
  – AST traversers
  – Conventional optimizations

• Output C/CUDA source code
tile = (len + gridDim.x - 1)/gridDim.x;
sub_tile = (tile + blockDim.x - 1)/blockDim.x;
accum = 0
#pragma unroll
for(unsigned i = 0; i < sub_tile; ++i) {
   accum += in[blockIdx.x*tile + i*blockDim.x + threadIdx.x];
}
tmp[threadIdx.x] = accum;
__syncthreads();
for(unsigned s=1; s<blockDim.x; s *= 2) {
   if(id >= s)
      tmp[threadIdx.x] +=
      tmp[threadIdx.x - s];
   __syncthreads();
}
partial[blockIdx.x] = tmp[blockDim.x-1];
return; // Launch new kernel to sum up partial
Experimental Results

- TANGRAM delivers 70% or higher performance compared to highly-optimized libraries, such as Intel MKL, NVIDIA CUBLAS, CUSPARSE, or Thrust, or experts’ optimized benchmarks, Rodinia.
FAQ1

• Why TANGRAM is better than other composition-based languages?
  – TANGRAM provides an architectural hierarchy model to guide composition
  – TANGRAM provides a representation of cooperative codelets for better SIMD utilization
    • Especially shuffle instructions and scratchpad
FAQ2

• Where optimizations happen?
  – Selection of codelets or map policies in Composition
  – Conventional optimizations in Codegen
  – Optimizations in backend compilers
FAQ3

• What? Multiple versions?
  – We did NOT ask users to write multiple versions of kernels
  – Codelets can be used to synthesize different versions of kernels
  – Codelets can be reused multiple times within one kernel, across kernels in a device, across kernels for different devices
Takeaways of TANGRAM

• Performance portability
  – 70% or higher performance compared to highly-optimized libraries

• Extensible architectural hierarchical model
  – Support CPU SIMD, GPU warp, ILP, even GPU dynamic parallelism

• Native description for algorithmic design space
  – Perfect for domain users
Questions