Efficient Kernel Synthesis for Performance Portable Programming

Li-Wen Chang[1], Izzat El Hajj[1], Christopher Rodrigues[2], Juan Gómez-Luna[3], Wen-Mei Hwu[1]

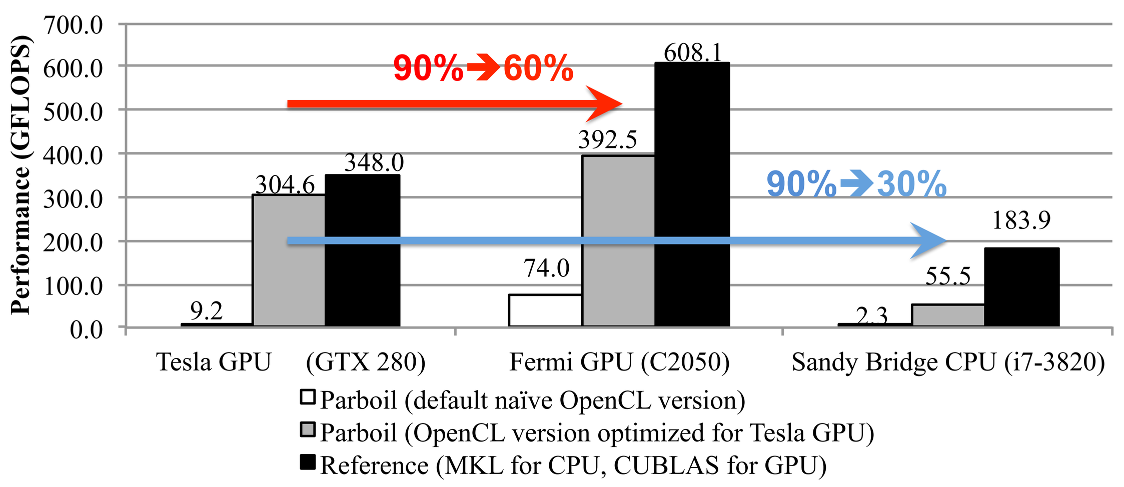
[1]University of Illinois at Urbana-Champaign, [2]Huawei America Research Lab, [3]Universidad de Córdoba

Motivation

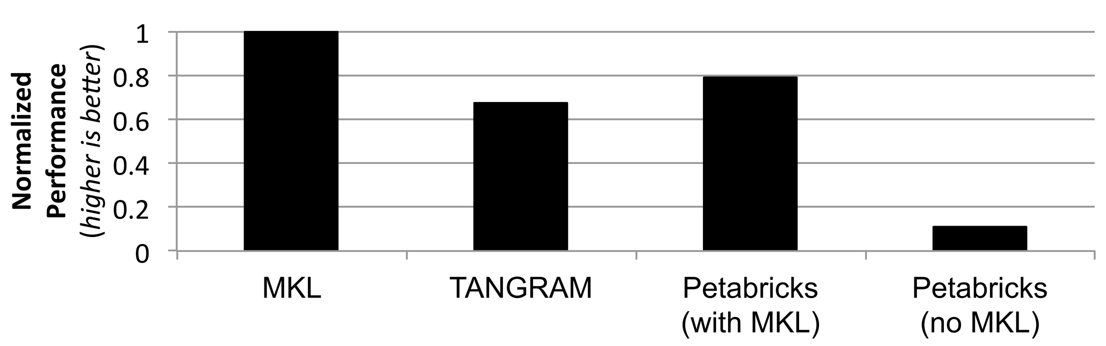
- Maintaining optimized programs for different devices is costly
- Programs written once should run on difference devices with performance, which is known performance portability

Limitations of Current Practice

- OpenCL is not performance portable



- Composition-based languages highly relying on high-performance base-rule implementations



TANGRAM Platform

- TANGRAM adopts codelet programming model
  - A codelet is defined as a code snippet reusable for one or many kernels
- Users write interchangeable alternative codelets, and corresponding composition and partition rules for a computation pattern (called spectrum)
  - We do **Not** ask users to write multiple versions of kernels
- TANGRAM supports recursive composition to adapt to different hierarchies of devices and cooperative codelets for SIMD architectures
- TANGRAM also provides performance tuning annotation to enable parameterization

TANGRAM Workflow

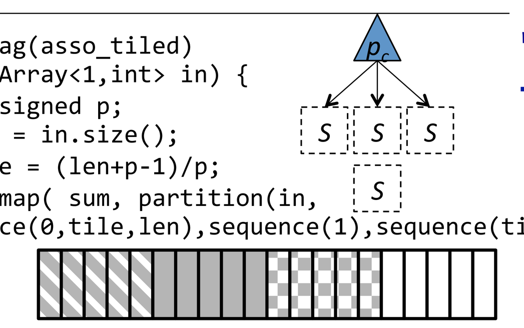
```
__codelet
int sum(const Array<1,int> in) {
  unsigned len = in.size();
  int accum = 0;
  for(unsigned i=0; i < len; ++i) {
    accum += in[i];
  }
  return accum;
}
```

(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];
}
```

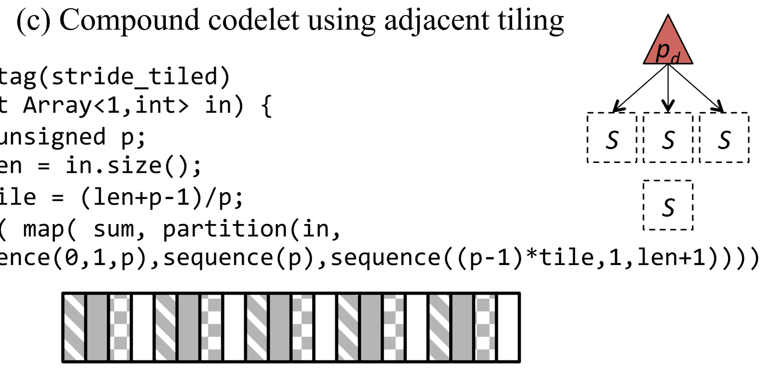
(b) Atomic cooperative codelet

```
__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

Program Composition Rules: (sum)

Rule 1: *compose(sum , L)* → S_L , *devolve*(ℓ_L) , *compose*(*sum*, ℓ_L)

Rule 2: *compose(sum , L)* → *compute*(c_a , SE_L)

Rule 3: *compose(sum , L)* → *compute* (c_b , VE_L)

Rule 4: *compose(sum , L)* → S_L , *regroup*(p_c, *L*) , *distribute*(ℓ_L) , *compose*(*sum*, ℓ_L) , *compose(sum, L)*

Rule 5: *compose(sum , L)* → S_L , *regroup*(p_d, *L*) , *distribute*(ℓ_L) , *compose*(*sum*, ℓ_L) , *compose(sum, L)*

Example for Deriving Composition Rules from Compound Codelets: (codelet c)

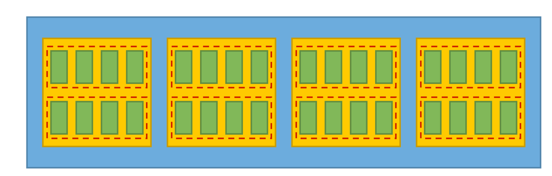
compose(sum , L) → *compose*(c_c , *L*)
→ *compose(sum(map(sum, partition(…,* p_c*))), L)*
→ *compose(map(sum, partition(…,* p_c*)) , L) , compose(sum, L)*
→ *compose(partition(…,* p_c*) , L) , compose(map(sum, …) , L) , compose(sum , L)*
→ S_L , *regroup*(p_c, *L*) , *distribute*(ℓ_L) , *compose*(*sum*, ℓ_L) , *compose(sum, L)*

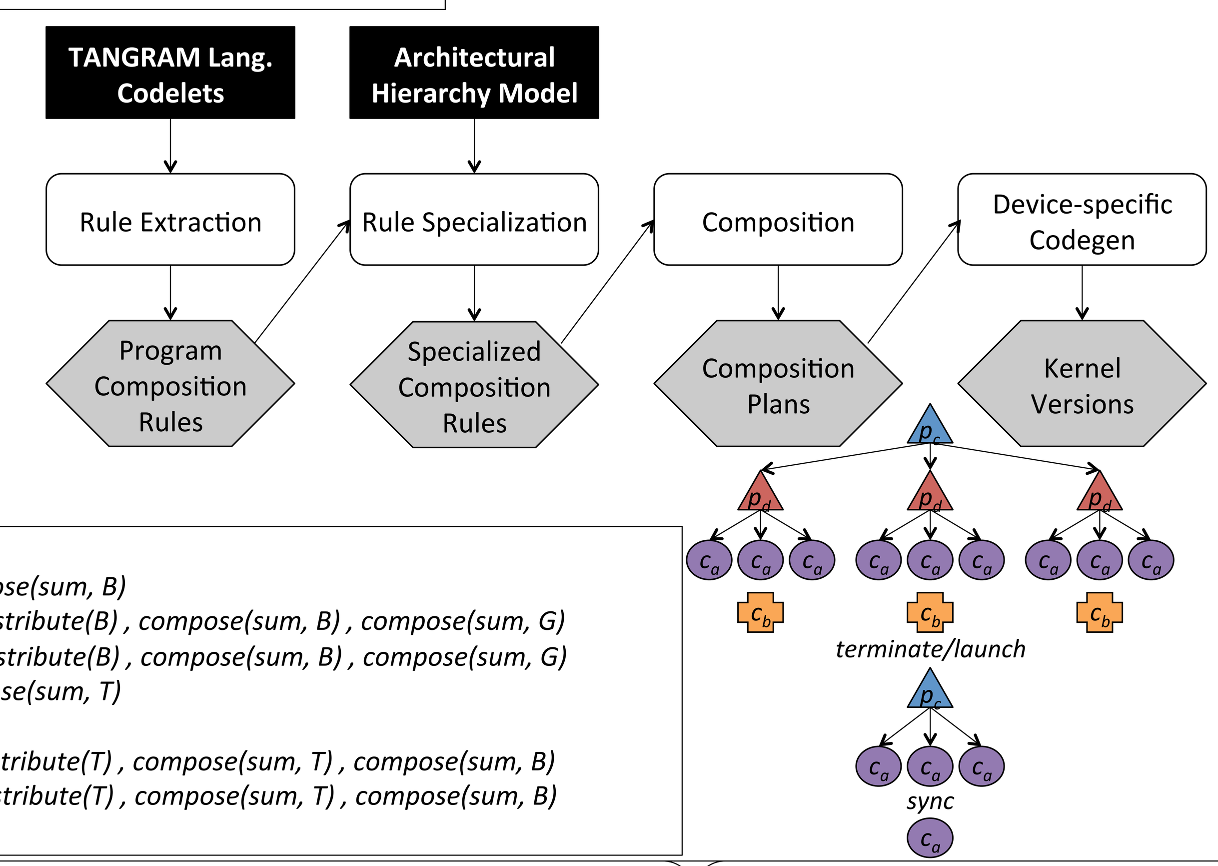
Specialized Composition Rules:

G rules: G1: *compose(sum , G)* → S_G , *devolve(B)* , *compose(sum, B)*
G4: *compose(sum , G)* → S_G , *regroup*(p_c, *G*) , *distribute(B)* , *compose(sum, B)* , *compose(sum, G)*
G5: *compose(sum , G)* → S_G , *regroup*(p_d, *G*) , *distribute(B)* , *compose(sum, B)* , *compose(sum, G)*
B rules: B1: *compose(sum , B)* → S_B , *devolve(T)* , *compose(sum, T)*
B3: *compose(sum , B)* → *compute* (c_b , VE_B)
B4: *compose(sum , B)* → S_B , *regroup*(p_c, *B*) , *distribute(T)* , *compose(sum, T)* , *compose(sum, B)*
B5: *compose(sum , B)* → S_B , *regroup*(p_d, *B*) , *distribute(T)* , *compose(sum, T)* , *compose(sum, B)*
T rules: T2: *compose(sum , T)* → *compute*(c_a , SE_T)

Device Specification:

$G := C_G = none$, (ℓ_G , S_G) = (*B* , *terminate/launch*) // grid
$B := C_B = VE_B$, (ℓ_B , S_B) = (*T* , *__syncthreads()*) //block
$T := C_T = SE_T$, (ℓ_T , S_T) = *none* // thread





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)

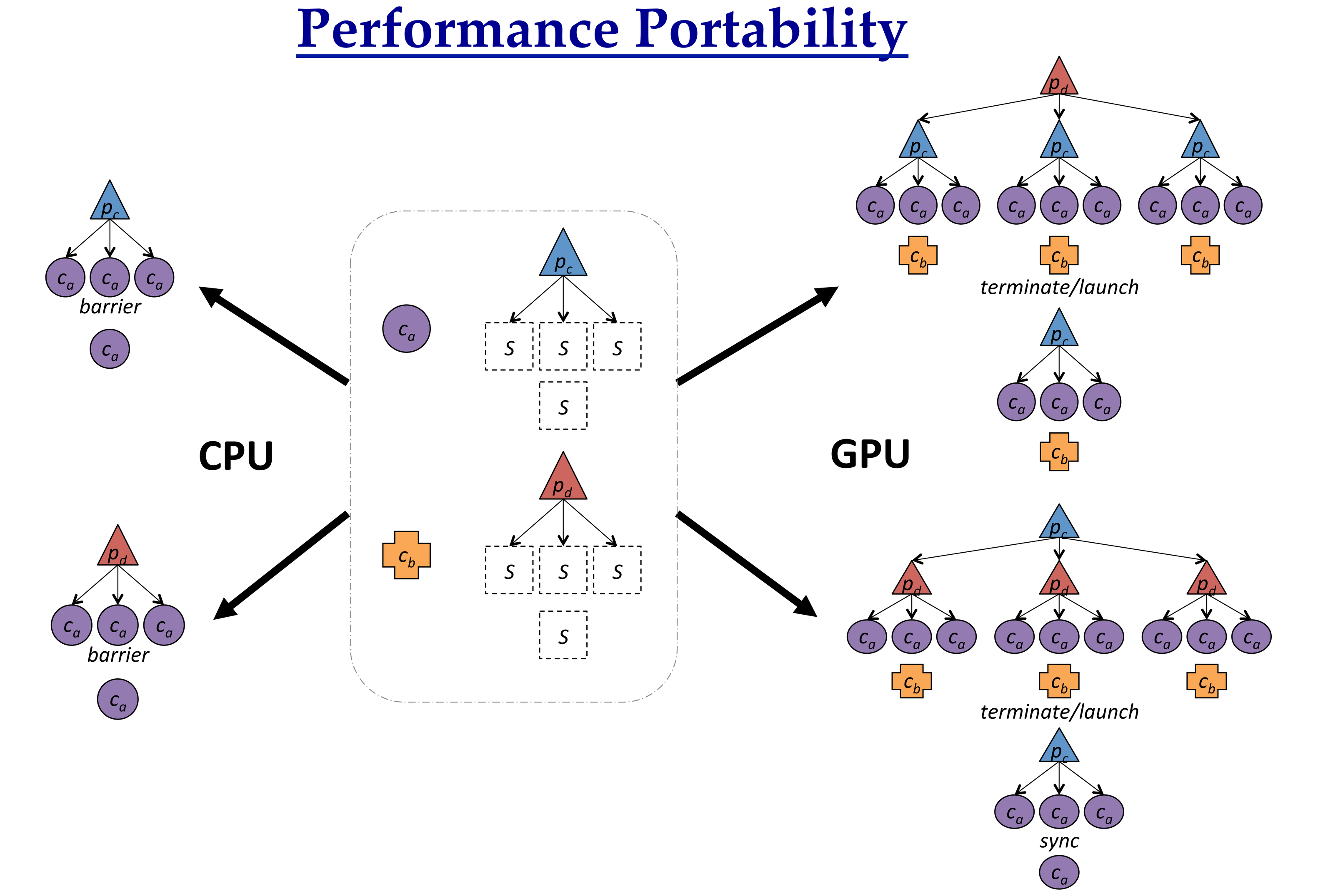
First kernel

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`

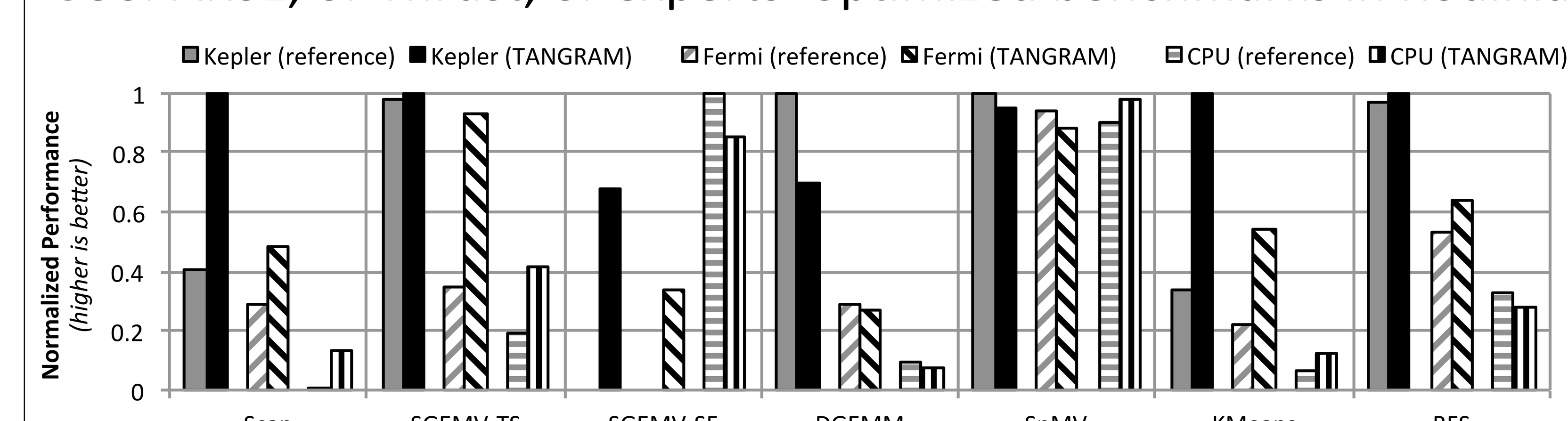
Performance Portability



- TANGRAM's device specification model is highly extensible to support CPU SIMD unit, GPU Warp, ILP, and GPU Dynamic Parallelism

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 in Rodinia



Conclusion

- We propose TANGRAM, a programming system for performance portability across devices
- Our results show TANGRAM can achieve promising performance across devices