TANGRAM
Efficient Kernel Synthesis for Performance Portable Programming

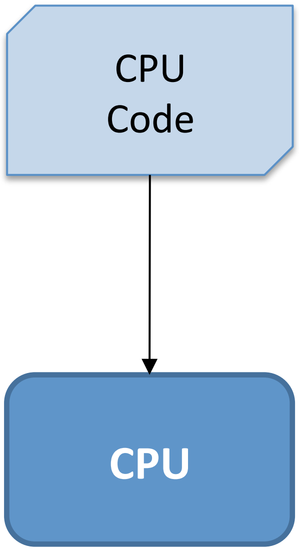
Li-Wen Chang[1], Izzat El Hajj[1], Christopher Rodrigues[2], Juan Gómez-Luna[3], Wen-Mei Hwu[1]
[1]University of Illinois, [2]Huawei America Research Lab, [3]Universidad de Córdoba
lchang20@illinois.edu

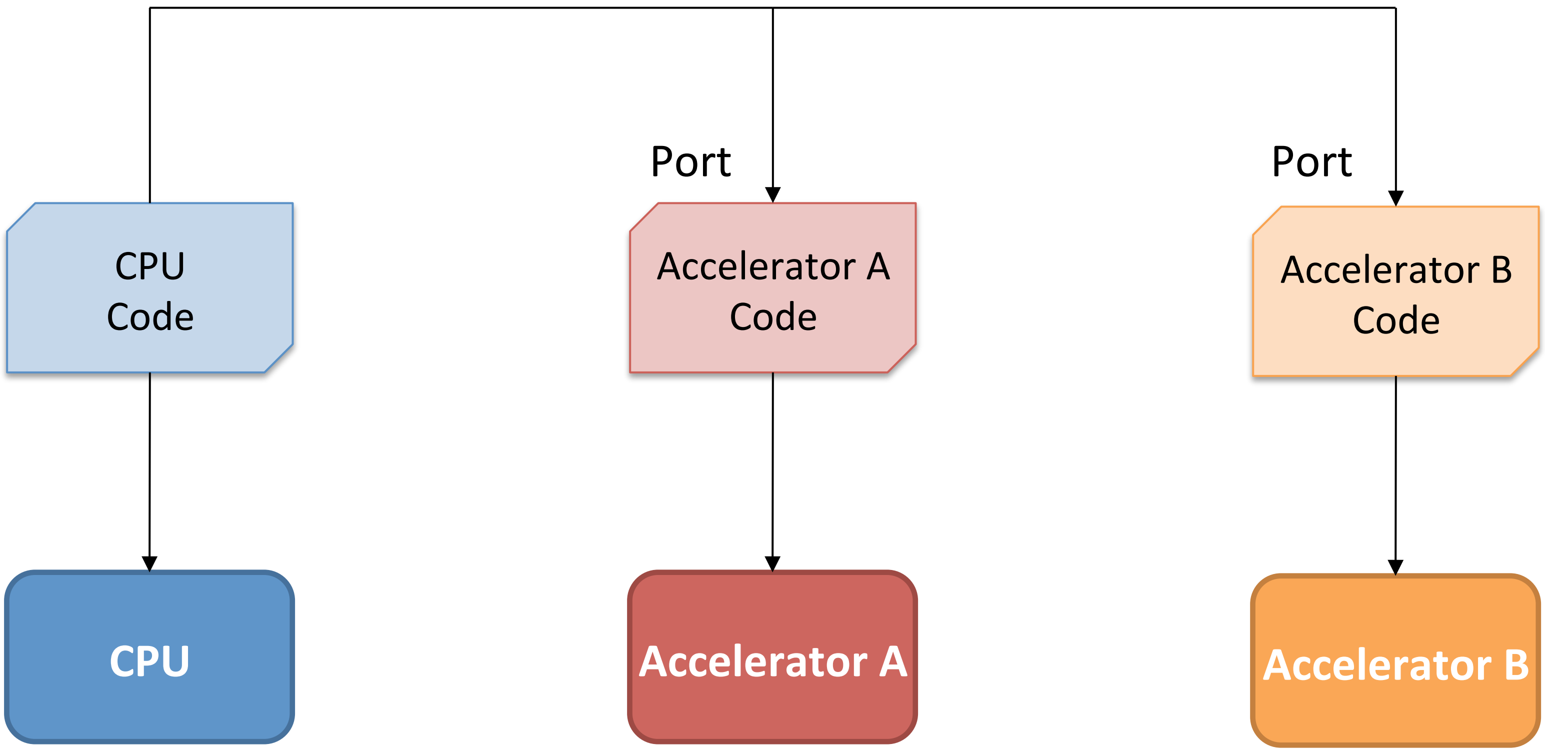
ECE ILLINOIS

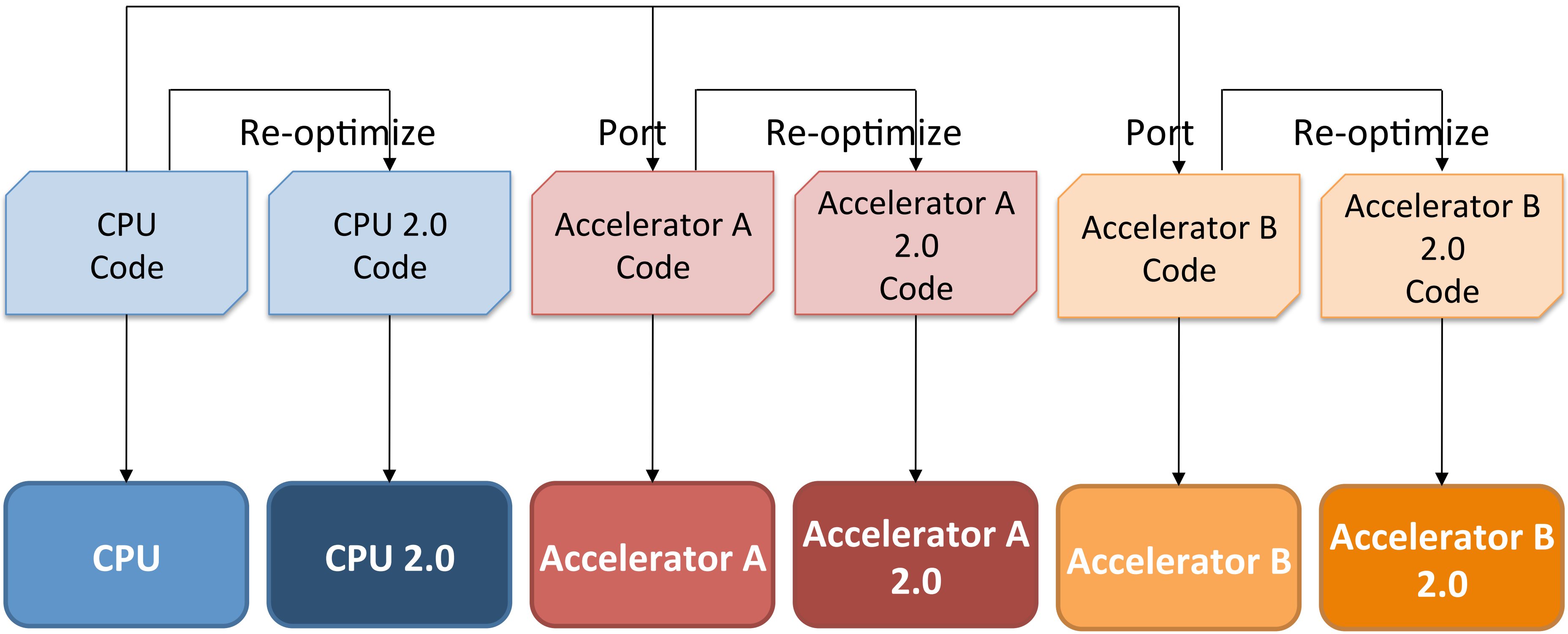
Performance Portability

- Maintaining optimized programs for different devices is costly

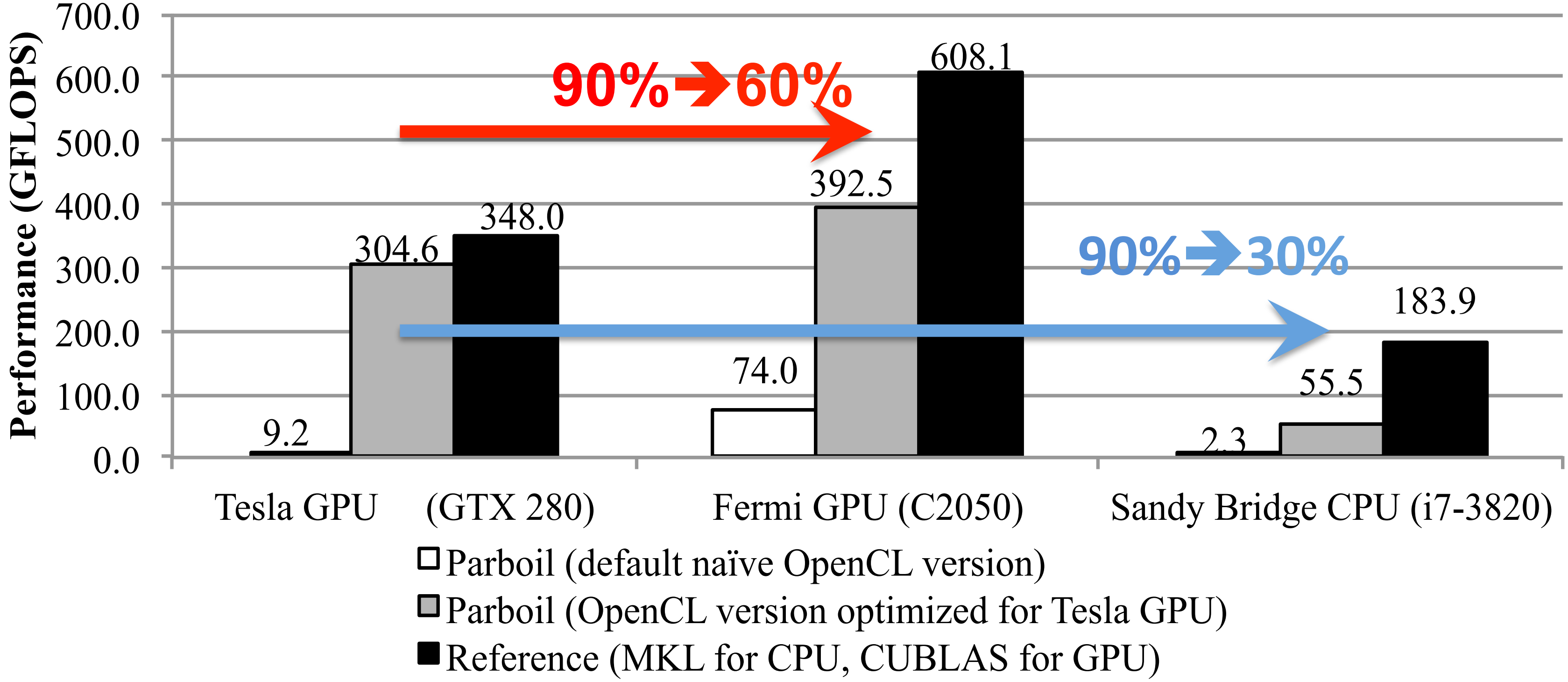
- Ideally, programs written once should run difference devices with performance

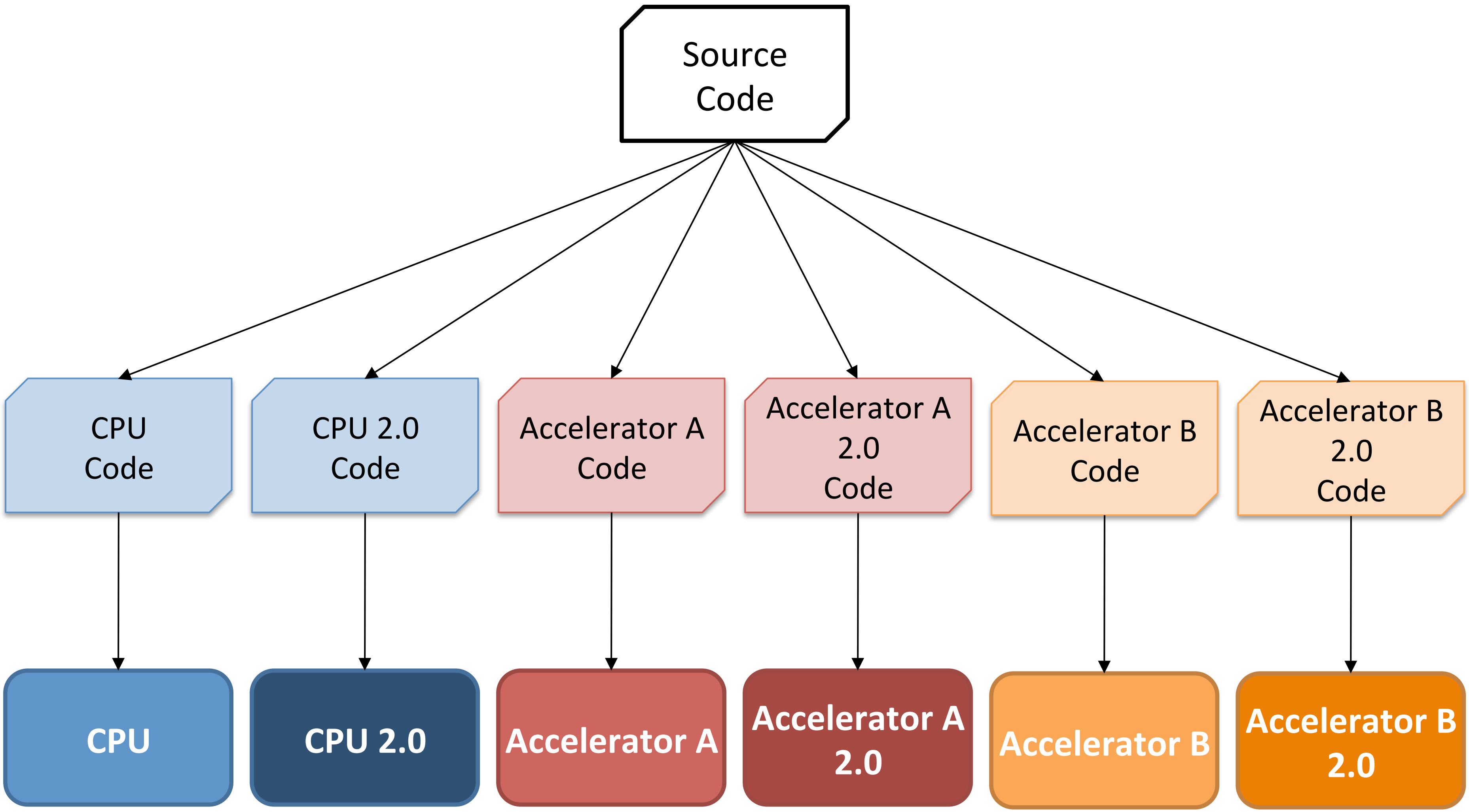

CPU
Code
CPU


Port
Port
CPU Code
Accelerator A Code
Accelerator B Code
CPU
Accelerator A
Accelerator B


Re-optimize
Port
Re-optimize
Port
Re-optimize
CPU Code
CPU 2.0 Code
Accelerator A Code
Accelerator A 2.0 Code
Accelerator B Code
Accelerator B 2.0 Code
CPU
CPU 2.0
Accelerator A
Accelerator A 2.0
Accelerator B
Accelerator B 2.0

Performance Portability: OpenCL SGEMM



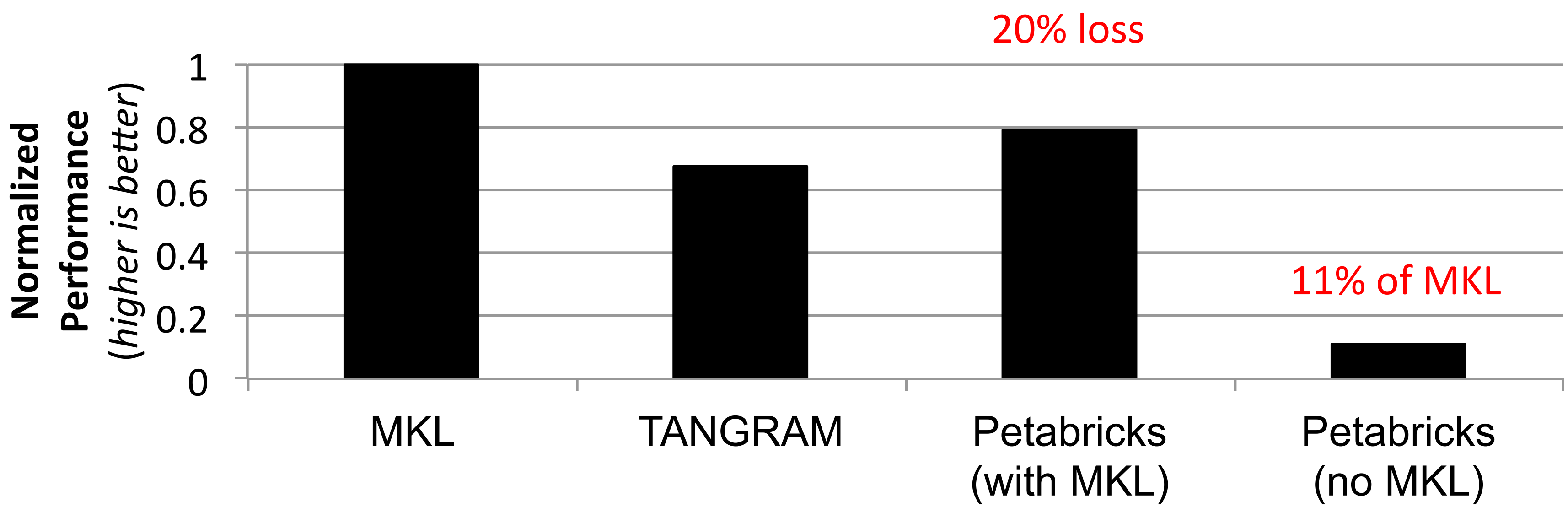


Source Code
CPU Code
CPU 2.0 Code
Accelerator A Code
Accelerator A 2.0 Code
Accelerator B Code
Accelerator B 2.0 Code
CPU
CPU 2.0
Accelerator A
Accelerator A 2.0
Accelerator B
Accelerator B 2.0

Composition-based Programing 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



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

```
__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

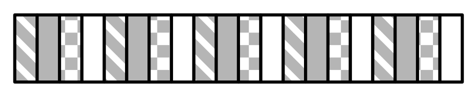
```
__codelet __coop __tag(kog)
int sum(const Array<1,int> in) {
  __shared int tmp[coopDim()];
  unsigned len = in.size();
  unsigned id = coopIdx();
  tmp[id] = (id < len)? in[id] : 0;
  for(unsigned s=1; s<coopDim(); s *= 2) {
    if(id >= s)
      tmp[id] += tmp[id - s];
  }
  return tmp[coopDim()-1];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

(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

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

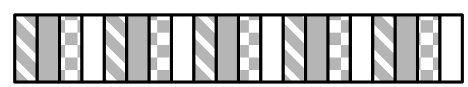
```
__codelet __coop __tag(kog)
int sum(const Array<1,int> in) {
  __shared int tmp[coopDim()];
  unsigned len = in.size();
  unsigned id = coopIdx();
  tmp[id] = (id < len)? in[id] : 0;
  for(unsigned s=1; s<coopDim(); s *= 2) {
    if(id >= s)
      tmp[id] += tmp[id - s];
  }
  return tmp[coopDim()-1];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

(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

TANGRAM Language

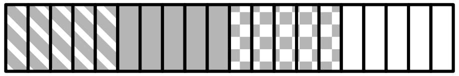
```
__codelet
int sum(const Array<1,int> in) {
  unsigned len = in.size();
  int accum = 0;
  for(unsigned i=0; i < len; ++i) {
    accum += in[i];
  }
  return accum;
}
```

(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];
}
```

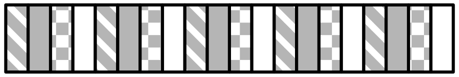
(b) Atomic cooperative codelet

```
__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

TANGRAM Language

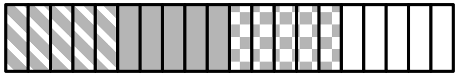
```
__codelet
int sum(const Array<1,int> in) {
  unsigned len = in.size();
  int accum = 0;
  for(unsigned i=0; i < len; ++i) {
    accum += in[i];
  }
  return accum;
}
```

(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];
}
```

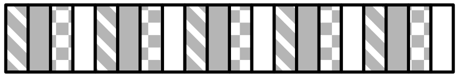
(b) Atomic cooperative codelet

```
__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

TANGRAM Language

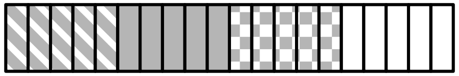
```
__codelet
int sum(const Array<1,int> in) {
  unsigned len = in.size();
  int accum = 0;
  for(unsigned i=0; i < len; ++i) {
    accum += in[i];
  }
  return accum;
}
```

(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];
}
```

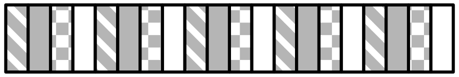
(b) Atomic cooperative codelet

```
__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

TANGRAM Language

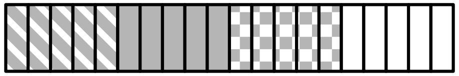
```
__codelet
int sum(const Array<1,int> in) {
  unsigned len = in.size();
  int accum = 0;
  for(unsigned i=0; i < len; ++i) {
    accum += in[i];
  }
  return accum;
}
```

(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];
}
```

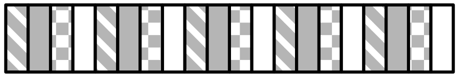
(b) Atomic cooperative codelet

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



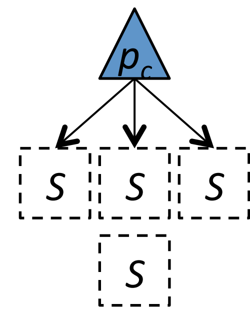
(c) Compound codelet using adjacent tiling

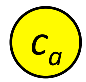
```
__codelet __tag(stride_tiled)
int sum(const Array<1,int> in) {
  __tunable unsigned p;
  unsigned len = in.size();
  unsigned tile = (len+p-1)/p;
  return sum( map( sum, partition(in,
      p,sequence(0,1,p),sequence(p),sequence((p-1)*tile,1,len+1))));
}
```



(d) Compound codelet using strided tiling

TANGRAM Language



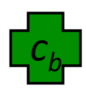



```
__codelet
int sum(const Array<1,int> in) {
  unsigned len = in.size();
  int accum = 0;
  for(unsigned i=0; i < len; ++i) {
    accum += in[i];
  }
  return accum;
}
```

(a) Atomic autonomous codelet

```
__codelet  __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))));
}
```

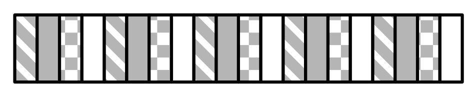
(c) Compound codelet using adjacent tiling



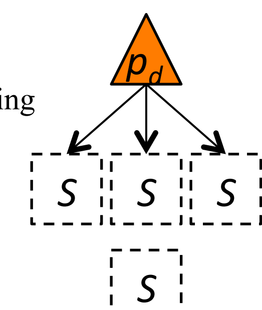

```
__codelet __coop __tag(kog)
int sum(const Array<1,int> in) {
  __shared int tmp[coopDim()];
  unsigned len = in.size();
  unsigned id = coopIdx();
  tmp[id] = (id < len)? in[id] : 0;
  for(unsigned s=1; s<coopDim(); s *= 2) {
    if(id >= s)
      tmp[id] += tmp[id - s];
  }
  return tmp[coopDim()-1];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

(b) Atomic cooperative codelet

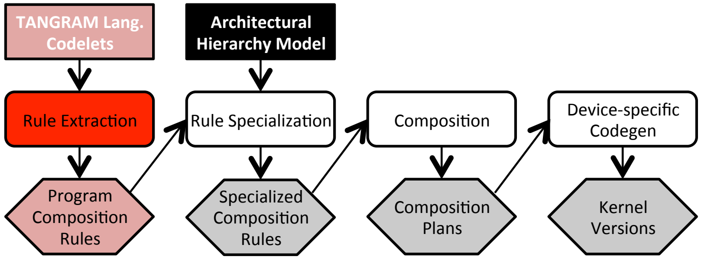
```
__codelet __tag(stride_tiled)
int sum(const Array<1,int> in) {
  __tunable unsigned p;
  unsigned len = in.size();
  unsigned tile = (len+p-1)/p;
  return sum( map( sum, partition(in,
      p,sequence(0,1,p),sequence(p),sequence((p-1)*tile,1,len+1))));
}
```



(d) Compound codelet using strided tiling



Rule Extraction



- TANGRAM parser
  - Clang 3.5
  - Customized TANGRAM AST builder
- Output a set of TANGRAM ASTs

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)*

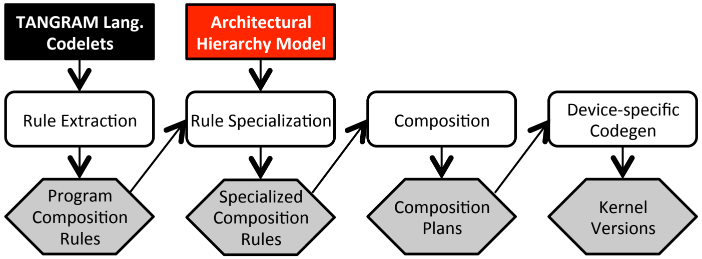
→ *compose(sum(map(sum, partition(…,* p_c*))), L)*

→ *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)*

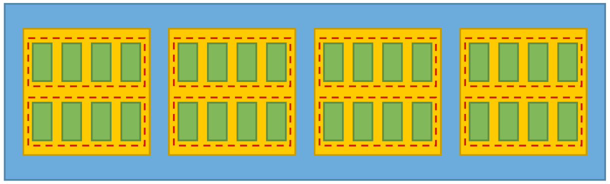
Architectural Hierarchy Model



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

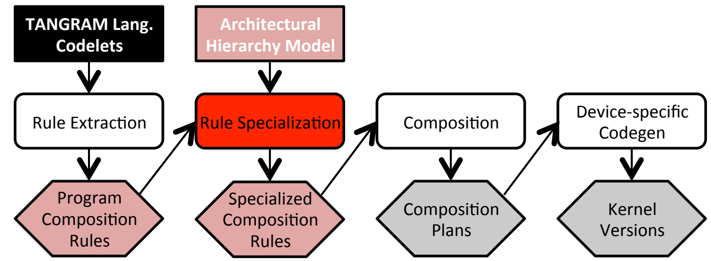
Device Specification:

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



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

Rule Specialization

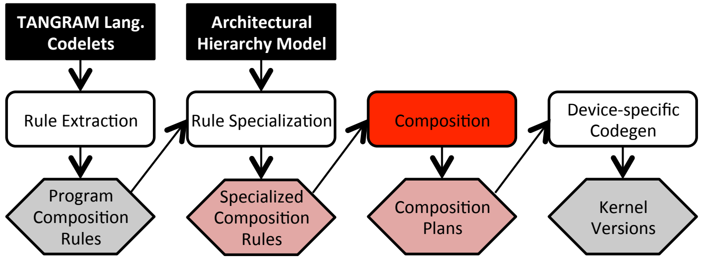


- TANGRAM analyzer
  - AST traverser
- Output a lookup table
  - Legal codelets for each level
  - Also prioritize them

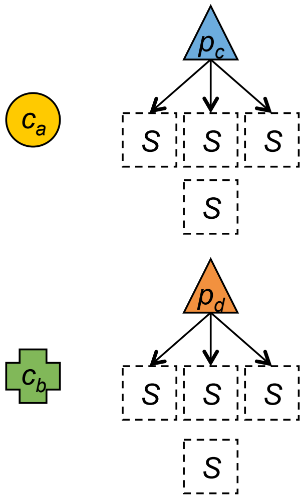
Specialized Composition Rules:

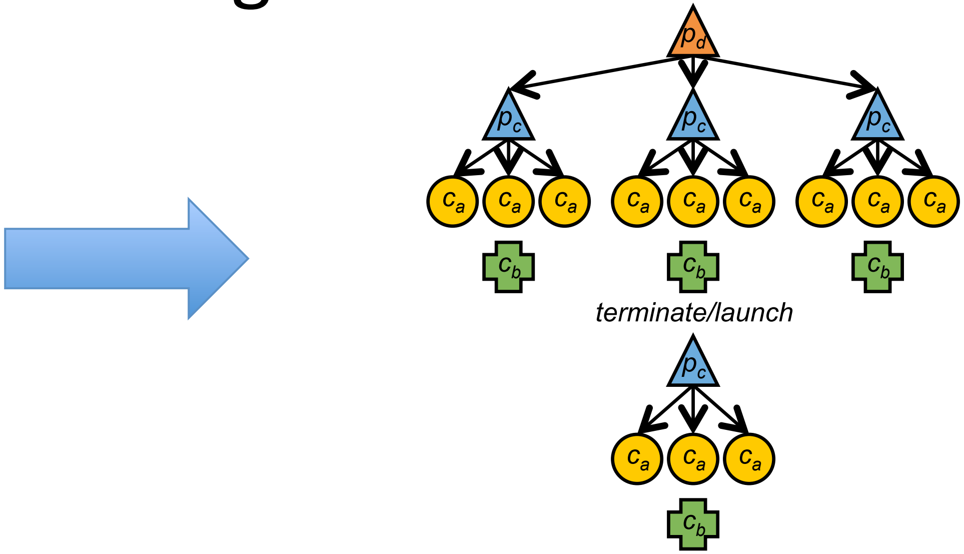
G rules:	G1: *compose(sum , G)* →		S_G , *devolve(B)* , *compose(sum, B)*
	G4: *compose(sum , G)* →		S_G , *regroup*(p_c , *G*) , *distribute(B)* , *compose(sum, B)* , *compose(sum, G)*
	G5: *compose(sum , G)* →		S_G , *regroup*(p_d , *G*) , *distribute(B)* , *compose(sum, B)* , *compose(sum, G)*
B rules:	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)

Composition

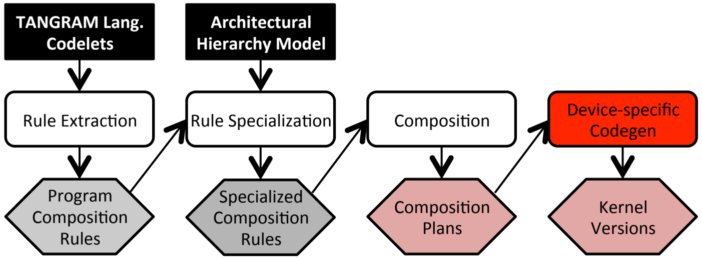


- TANGRAM planner
  - AST traverser/builder
  - Selection of codelets or map policies
  - Pruning
- Output ASTs for codegen





Codegen



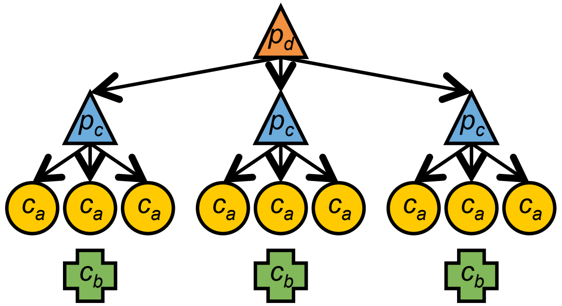
- TANGRAM codegen
  - AST traversers
  - Conventional optimizations
- Output C/CUDA source code

GPU Codegen Example

```
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
```

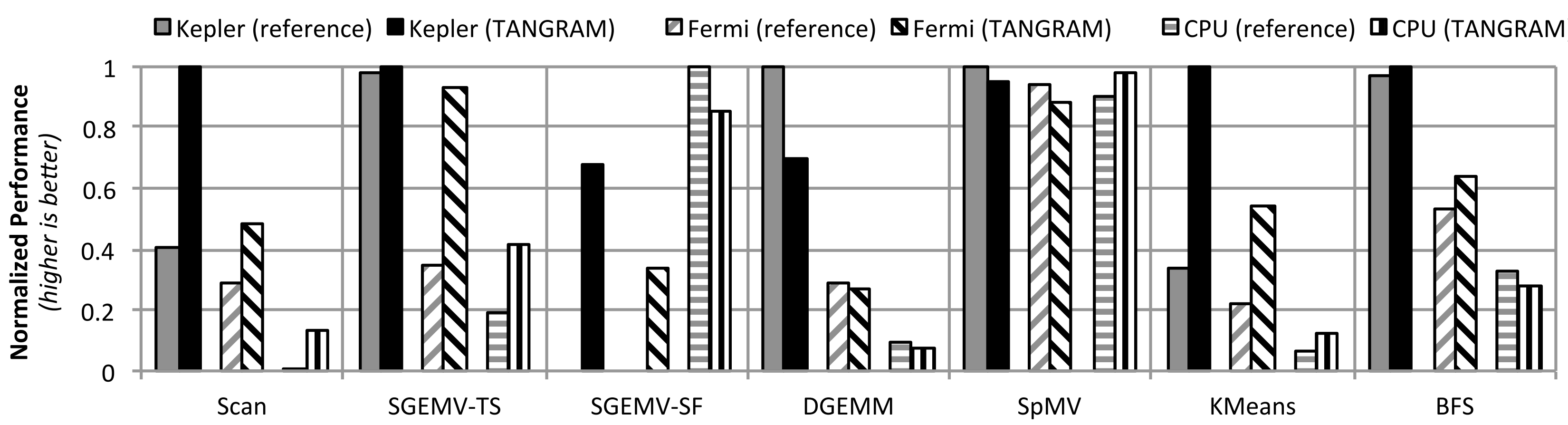
GPU

1. Grid
2. Block
3. Thread



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