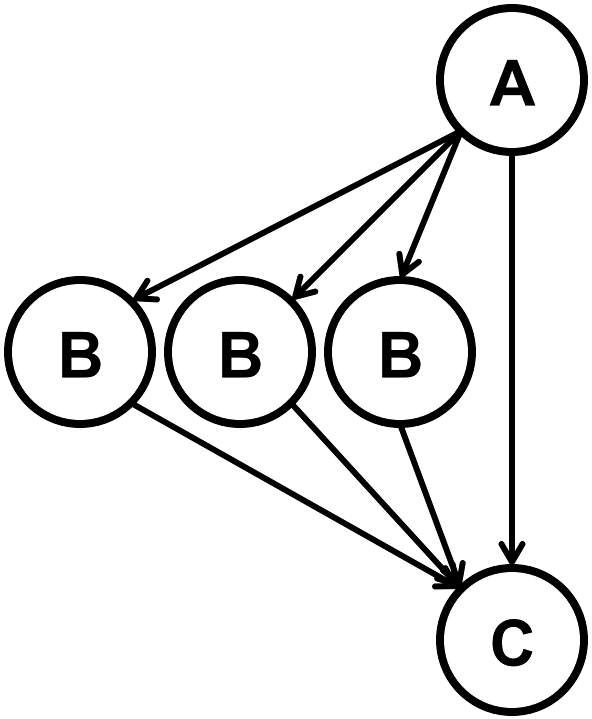
Regent

Elliott Slaughter

Regent

- A language for the Legion programming model
- Implicit parallelism, sequential semantics
- Tasks + automatic discovery of dependences
- Automatic data movement

```
A(r)
for i = 0, 3 do
  B(p[i])
end
C(r)
```



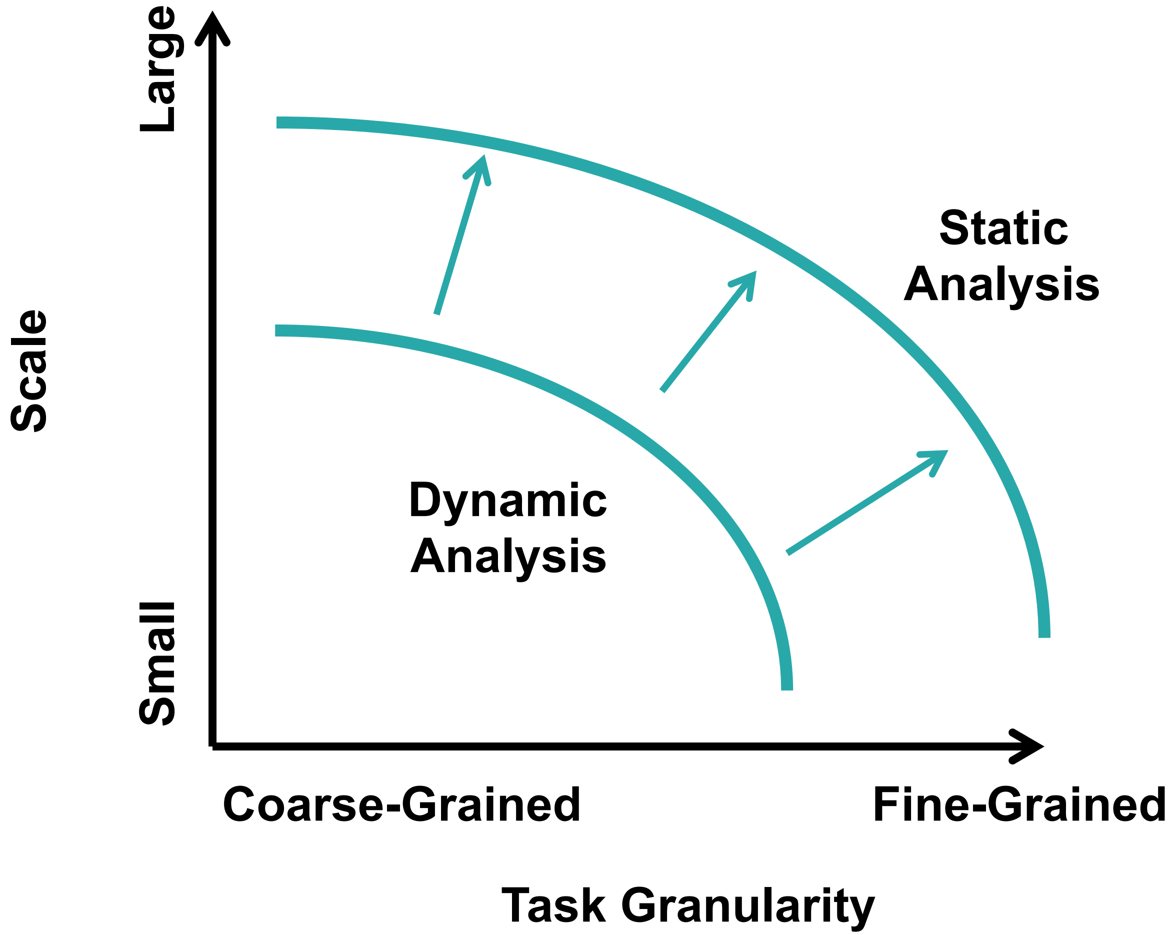
Regent vs Legion API

```
A(r)
for i = 0, 3 do
  B(p[i])
end
C(r)
```

```
runtime->unmap_region(ctx, physical_r);
TaskLauncher launcher_A(TASK_A, TaskArgument());
launcher_A.add_region_requirement(
  RegionRequirement(r, READ_WRITE, EXCLUSIVE, r));
launcher_A.add_field(0, FIELD_X);
launcher_A.add_field(0, FIELD_Y);
runtime->execute_task(ctx, launcher_A);
Domain domain = Domain::from_rect<1>(
  Rect<1>(Point<1>(0), Point<1>(2)));
IndexLauncher launcher_B(TASK_B, domain,
  TaskArgument(), ArgumentMap());
launcher_B.add_region_requirement(
  RegionRequirement(p, 0 /* projection */,
    READ_WRITE, EXCLUSIVE, r));
launcher_B.add_field(0, FIELD_X);
runtime->execute_index_space(ctx, launcher_B);
TaskLauncher launcher_C(TASK_A, TaskArgument());
launcher_C.add_region_requirement(
  RegionRequirement(r, READ_ONLY, EXCLUSIVE, r));
launcher_C.add_field(0, FIELD_X);
launcher_C.add_field(0, FIELD_Y);
runtime->execute_task(ctx, launcher_C);
runtime->map_region(ctx, physical_r);
```

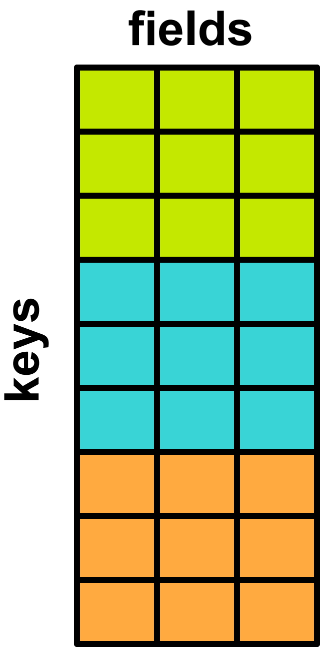
- **Regent simplifies Legion prog. model**
- **Regent achieves performance identical to hand-tuned Legion**

Pushing the Performance Envelope with Compilation

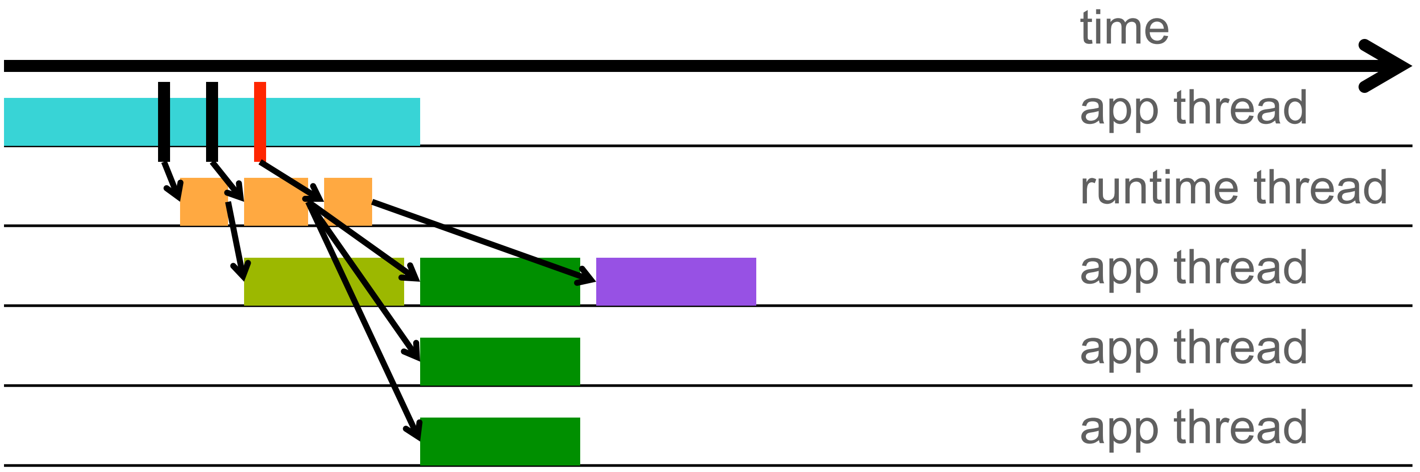


Data Model

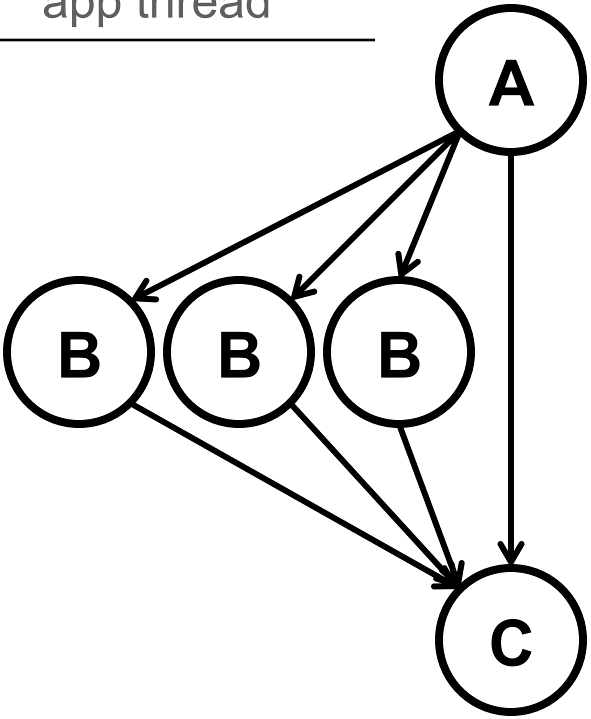
```
task A(r : region(…)) where writes(r.{x, y}) do … end
task B(r : region(…)) where reads writes(r.x) do … end
task C(r : region(…)) where reads(r.{x, y}) do … end

task main()
  var r = region(…)
  var p = partition(equal, r, …)
  A(r)
  for i = 0, 3 do
    B(p[i])
  end
  C(r)
end
```



Execution Model




```
var r = region(…)
var p = partition(disjoint, r, …)
A(r)
for i = 0, 3 do
  B(p[i])
end
C(r)
```



Regions

```
fspace point { x : int, y : int, z : int }
fspace node(list : region(node)) {
  idx : int2d,
  next : ptr(node(list), list),
}

task main()
  var bag = ispace(ptr, 28)
  var grid = ispace(int2d, {x = 4, y = 7})
  var points = region(grid, point)
  var list = region(bag, node(list))
  …
```

Fills and Copies

```
task main()
  var grid, points, list = …
  fill(points.{x, y, z}, 0)
  copy(points.{x, y}, list.idx.{x, y})
  …
```

Tasks

```
task init_pointers(grid : ispace(int2d),
                   points : region(grid, point),
                   list : region(node(list)))
where reads(points), reads writes(list.{idx, next}) do
  …
end

task main()
  var grid, points, list = …
  init_pointers(grid, points, list)
  …
```

Control

```
task main()
  var grid, points, list = …
  if c1 then … elseif c2 then … else … end
  while c do … end
  for idx = 0, n do … end
  for idx in grid do … end
  for elt in list do … end
  …
```

Pointers

```
task main()
  var grid, points, list = …
  var last = null(ptr(node(list), list))
  for idx in grid do
    var elt = new(ptr(node(list), list))
    elt.next = last
    last = elt
    elt.point = idx
    points[idx].{x, y, z} += 1
  end
  …
```

Vectorization

```
task inc(grid : ispace(int2d), points : region(grid, point),
           list : region(node(list)))
where reads(list), reduces+(points) do
  __demand(__vectorize)
  for elt in list do
    points[elt.idx].{x, y, z} += 1
  end
end
```

CUDA

```
__demand(__cuda)
task inc(grid : ispace(int2d), points : region(grid, point),
              list : region(node(list)))
where reads(list), reduces+(points) do
  for elt in list do
    points[elt.idx].{x, y, z} += 1
  end
end
```

C Functions

```
local cstdio = terralib.includec("stdio.h")
local cmath = terralib.includec("math.h")

task main()
  cstdio.printf("Hello, %f\n", cmath.sin(1.0))
  …
```

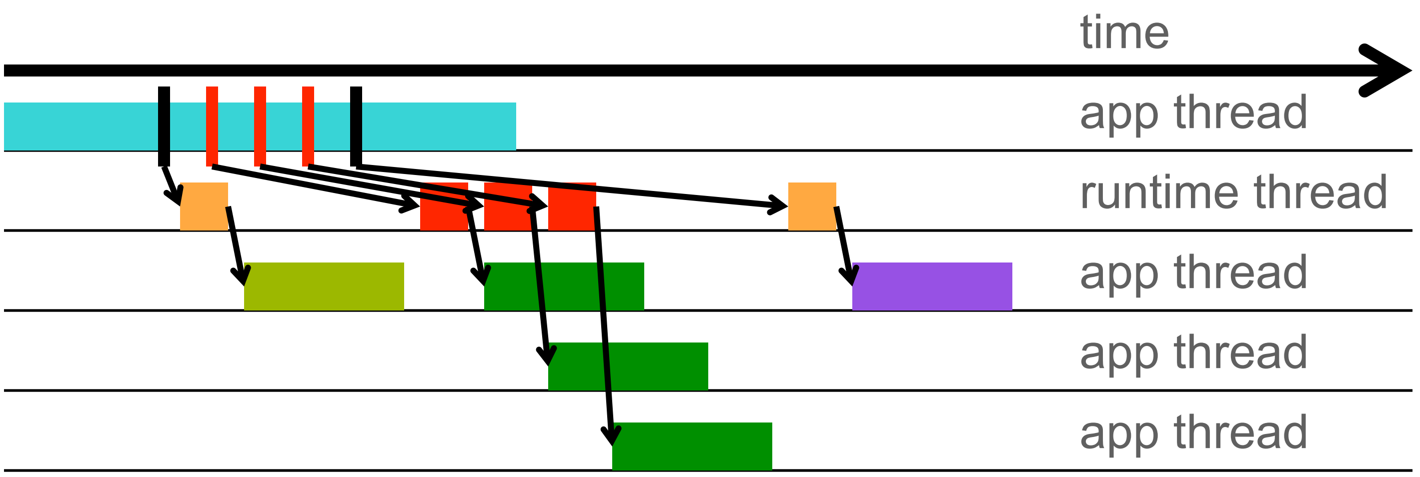
Legion Interop

```
terralib.linklibrary(“my.so”)
local my = terralib.includec(“my.h”)

task main()
  my.legion_task(__runtime(), __context())
  …
```

Metaprogramming

```
function make_inc(t, v)
  local task inc(r : region(t)) where reads writes(r) do
    for x in r do x += v end
  end
  return inc
end
local inc1 = make_inc(int, 1)

task main()
  var r = …
  inc1(r)
  …
```

Optimization: Index Launches (Before)



var r = region(…)

var p = partition(disjoint, r, …)

A(r)

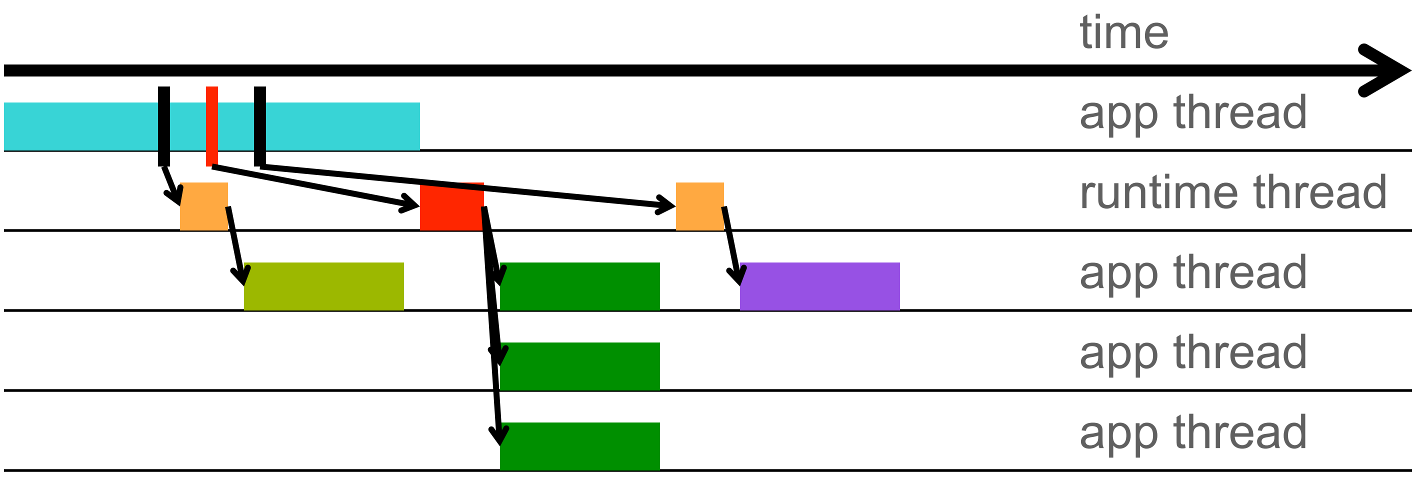
for i = 0, 3 do

  B(p[i])

end

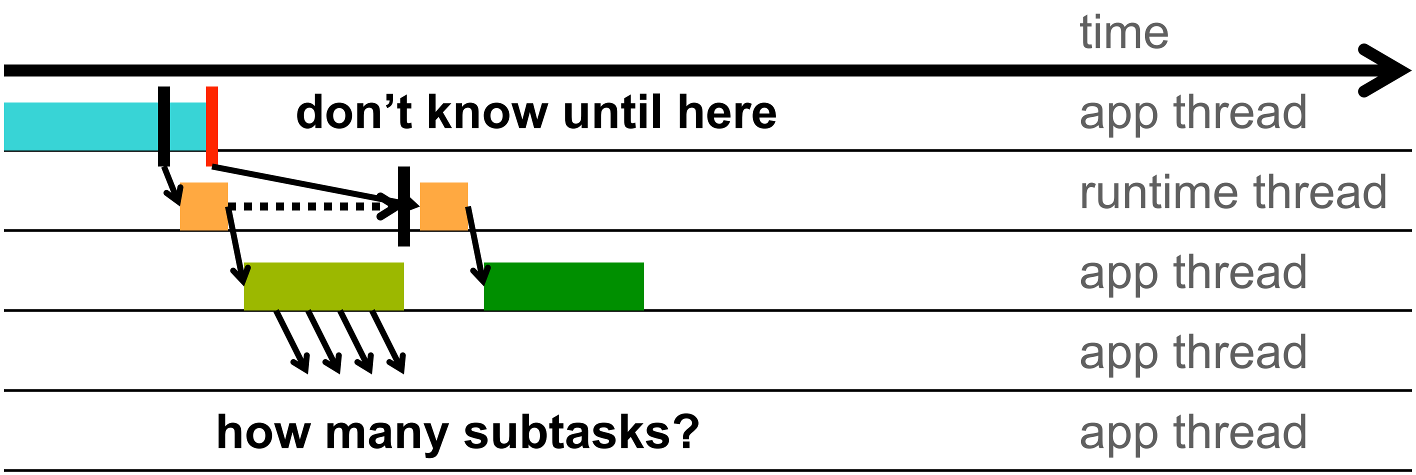
C(r)

Optimization: Index Launches (After)



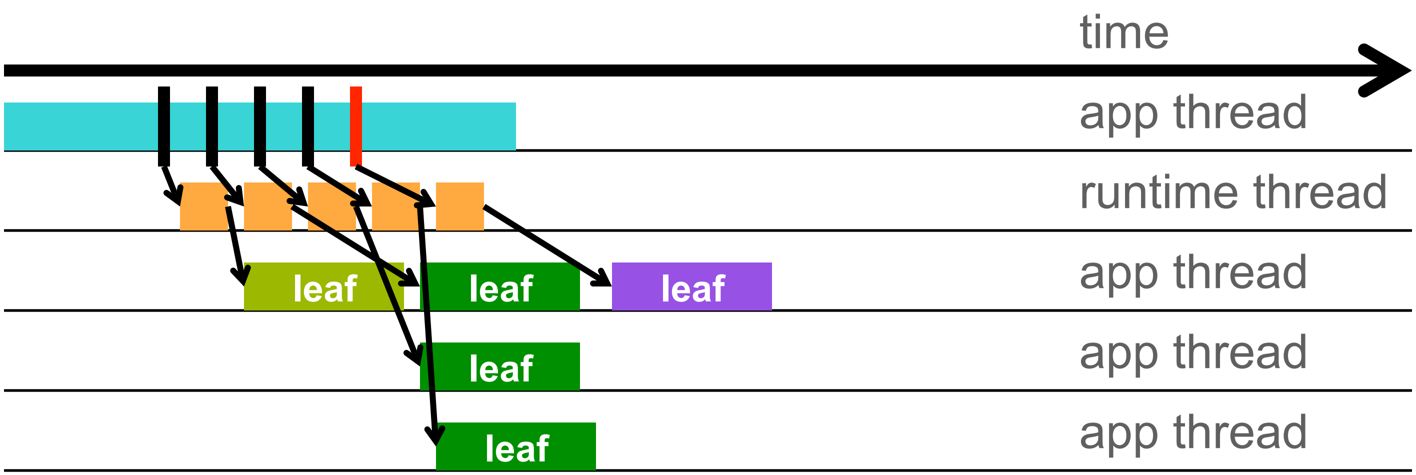

```
var r = region(…)
var p = partition(disjoint, r, …)
A(r)
for i = 0, 3: B(p[i])
C(r)
```

Optimization: Leaf Tasks (Before)



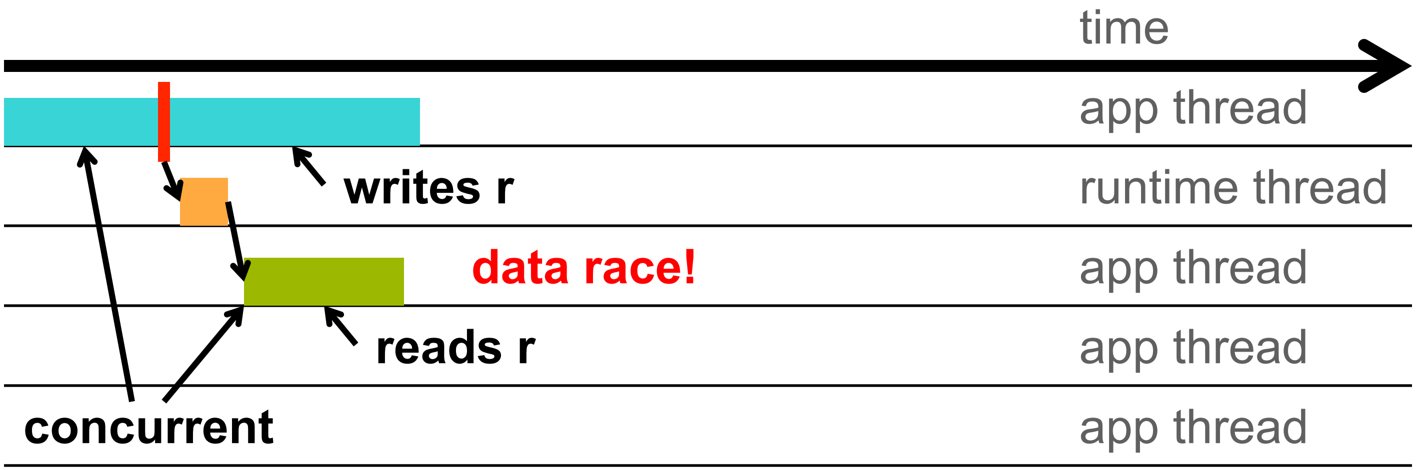

```
var r = region(…)
var p = partition(disjoint, r, …)
A(r)
for i = 0, 3 do
  B(p[i])
end
C(r)
```

Optimization: Leaf Tasks (After)



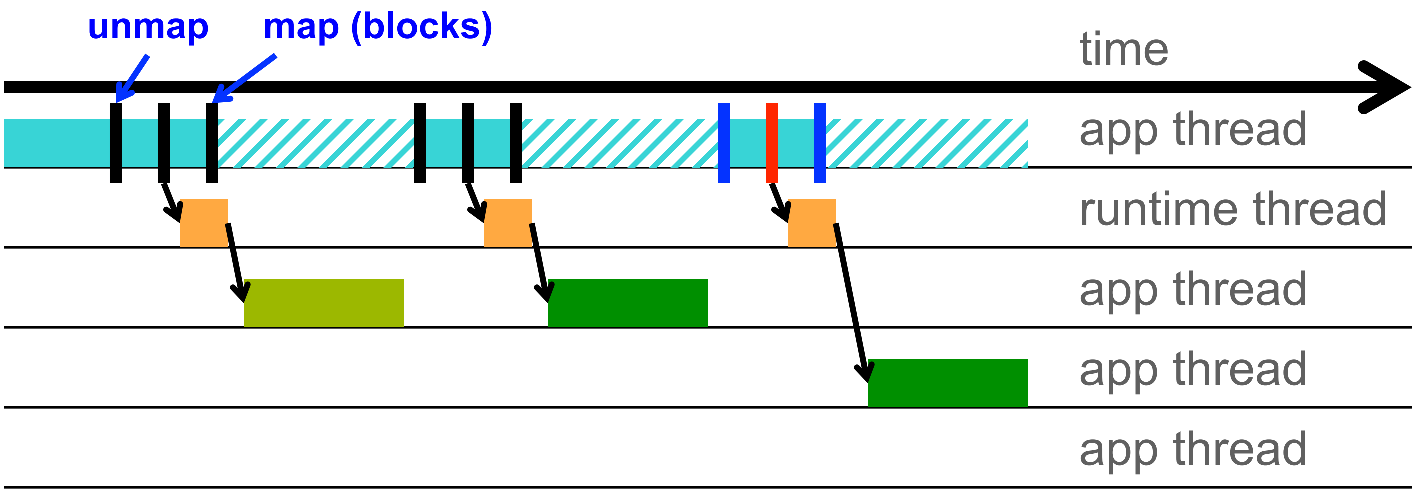

```
var r = region(…)
var p = partition(disjoint, r, …)
A(r)
for i = 0, 3 do
  B(p[i])
end
C(r)
```

Optimization: Mapping (Before)



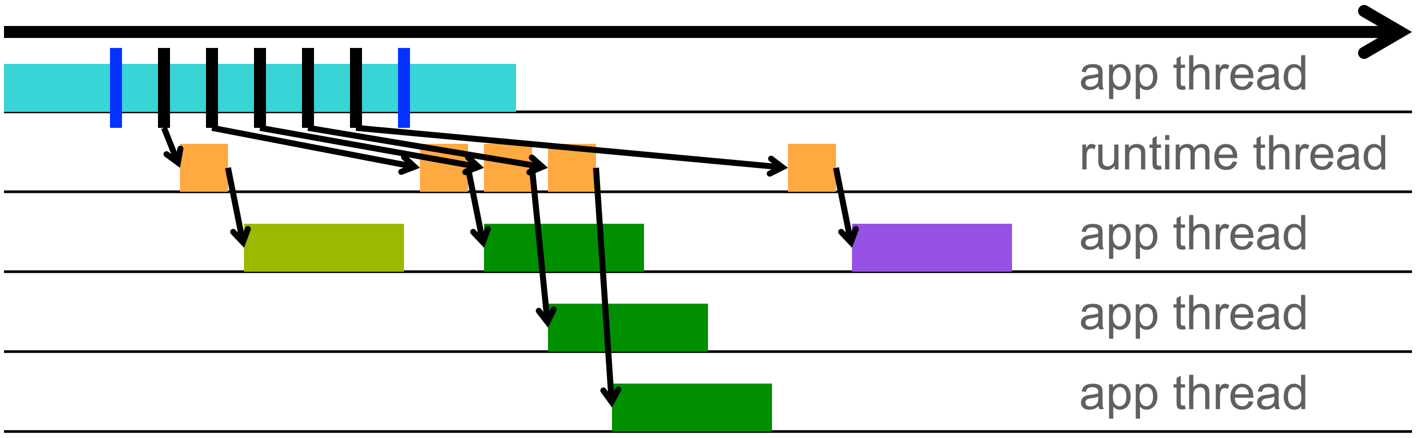

```
var r = region(…)
var p = partition(disjoint, r, …)
A(r)
for i = 0, 3 do
  B(p[i])
end
C(r)
```

Optimization: Mapping (Runtime)




```
unmap(r)
A(r)
map(r) -- blocks
for i = 0, 3 do
  unmap(r)
  B(p[i])
  map(r) -- blocks
end
```

Optimization: Mapping (Compiler)

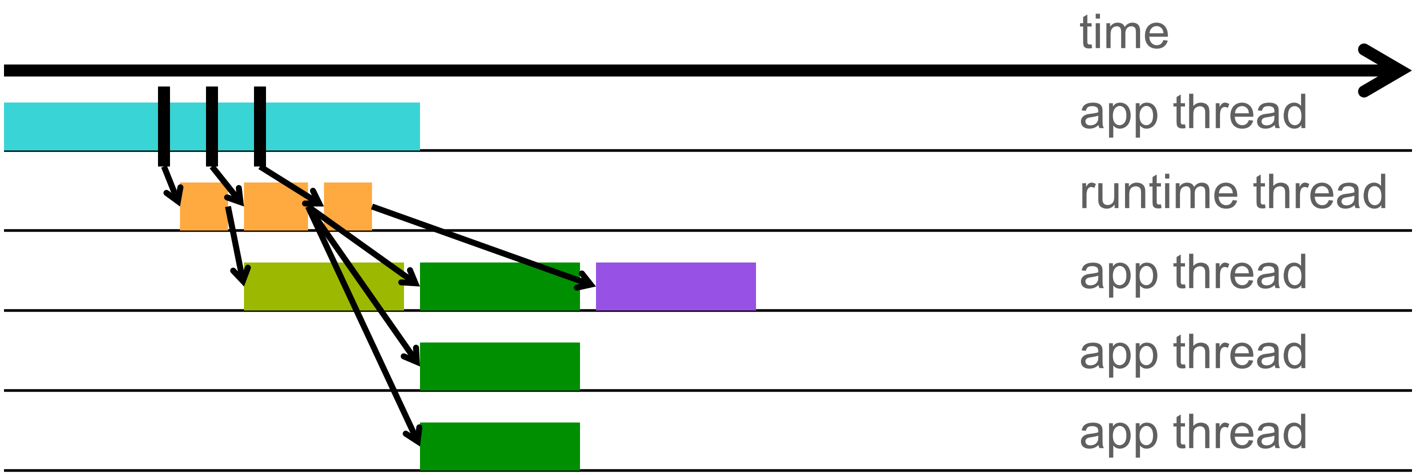


unmap(r)
A(r)
for i = 0, 3 do
  B(p[i])
end
C(r)
map(r) ***-- blocks***

Other Optimizations

- Futures
- Pointer Check Elision
- Dynamic Branch Elision
- Vectorization
- CUDA Kernel Generation

Work In Progress: Static Dependences



var r = region(…)

var p = partition(disjoint, r, …)

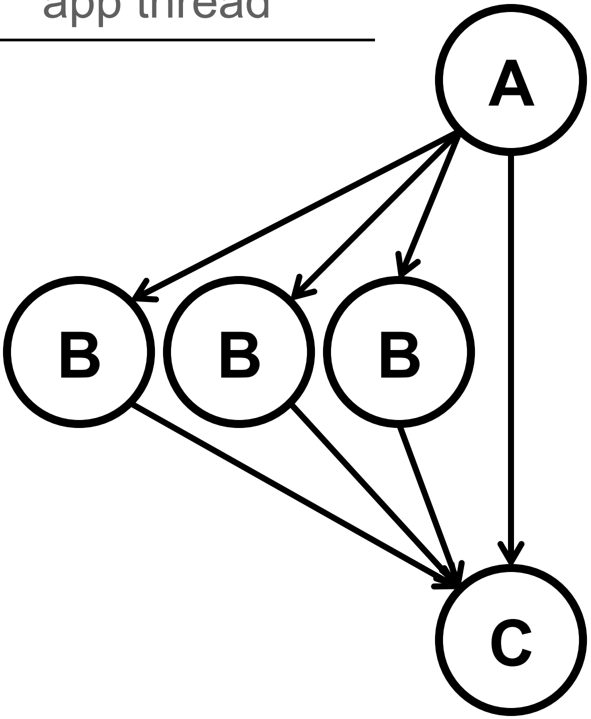
A(r)

for i = 0, 3 do

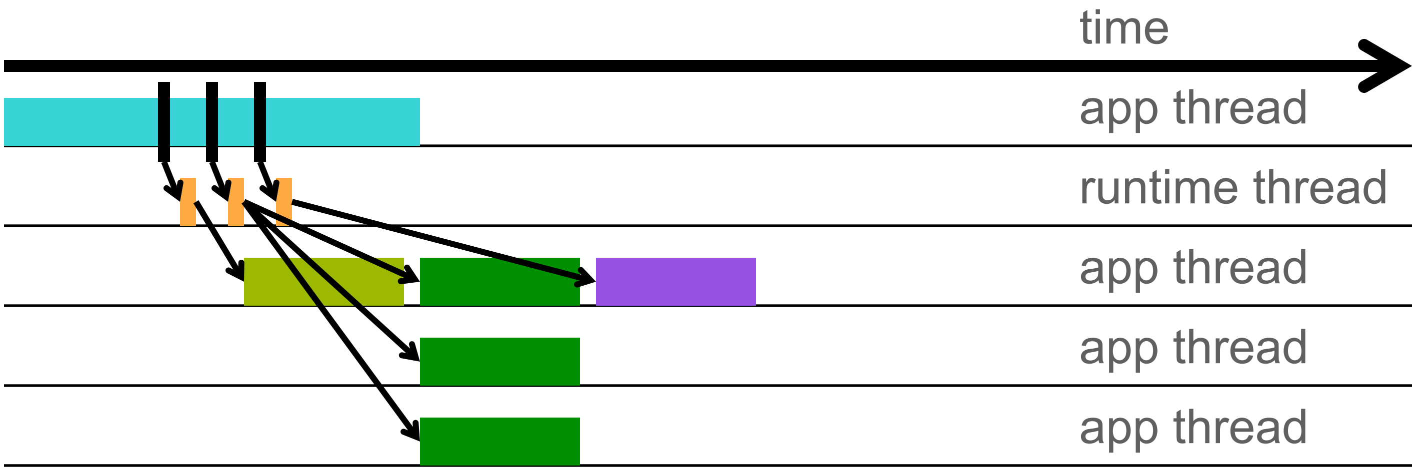
  B(p[i])

end

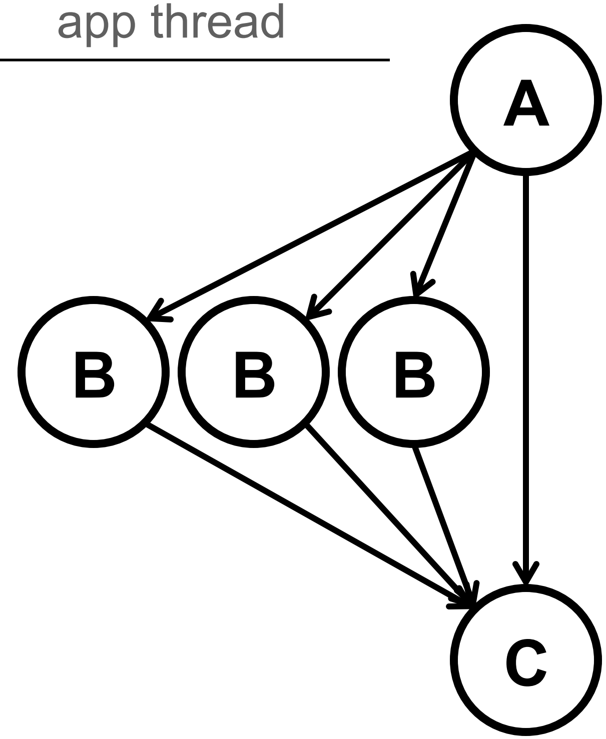
C(r)



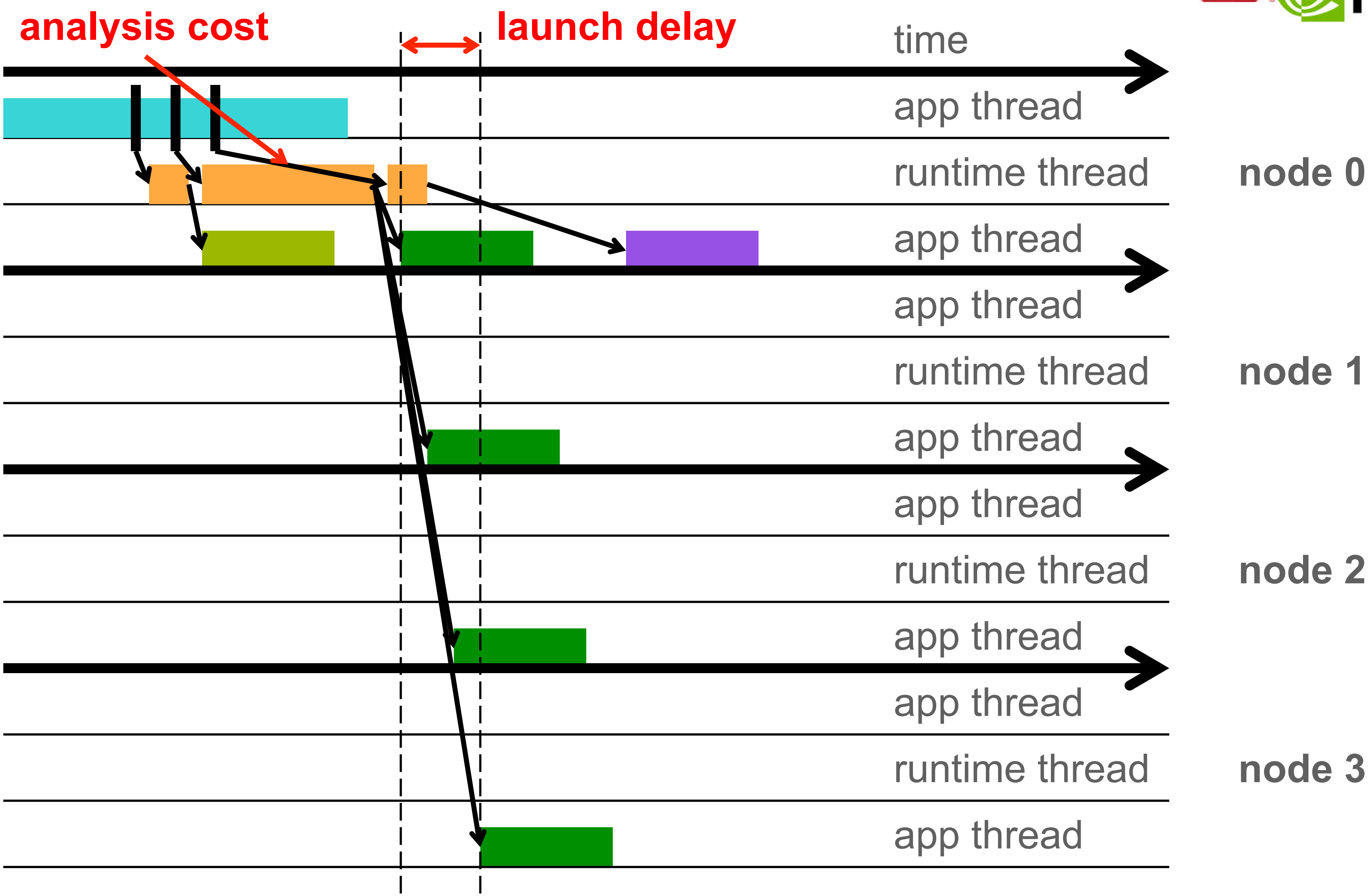
Work In Progress: Static Dependences



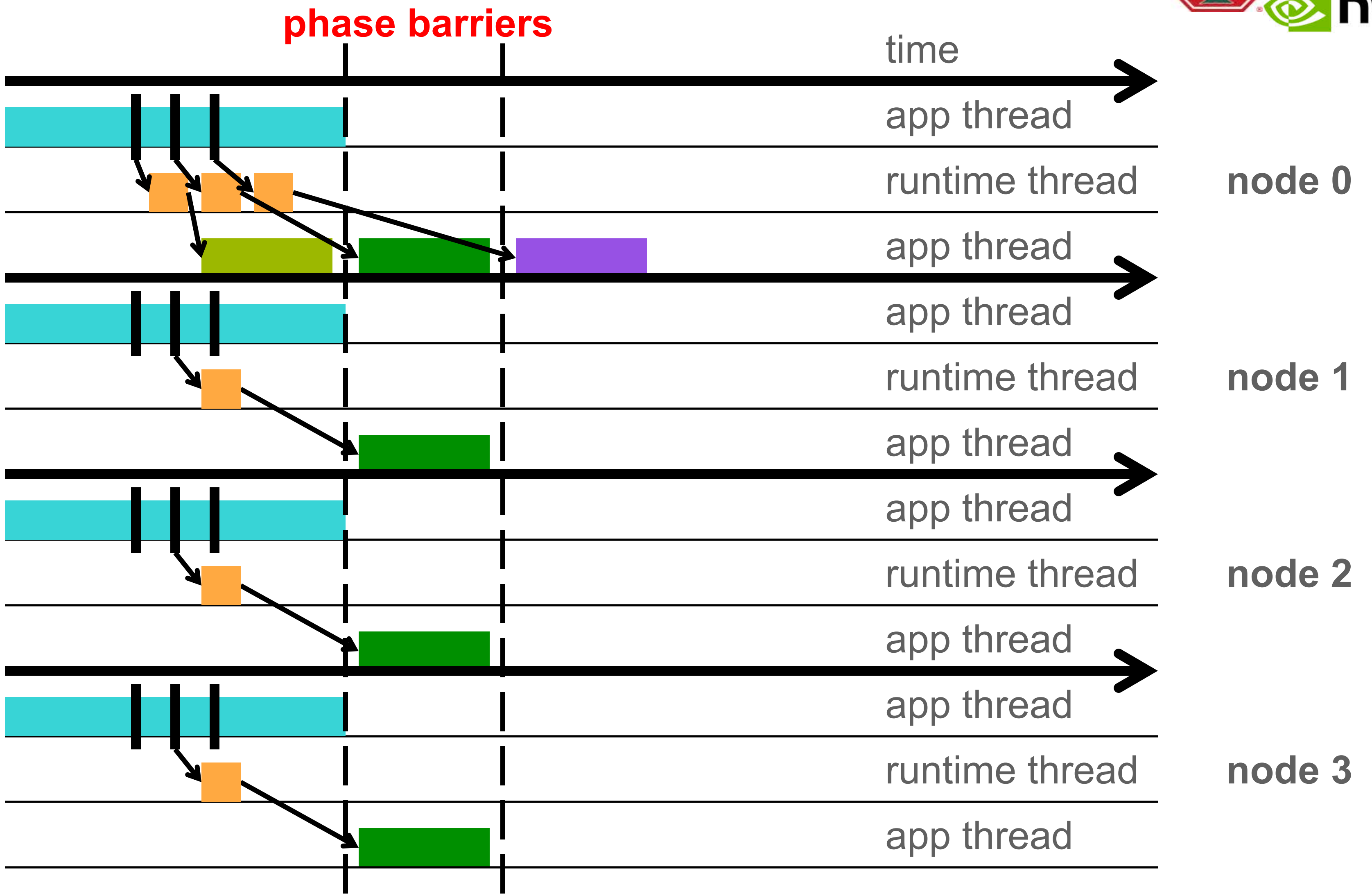

```
var r = region(…)
var p = partition(disjoint, r, …)
A(r)
for i = 0, 3 do
  B(p[i])
end
C(r)
```



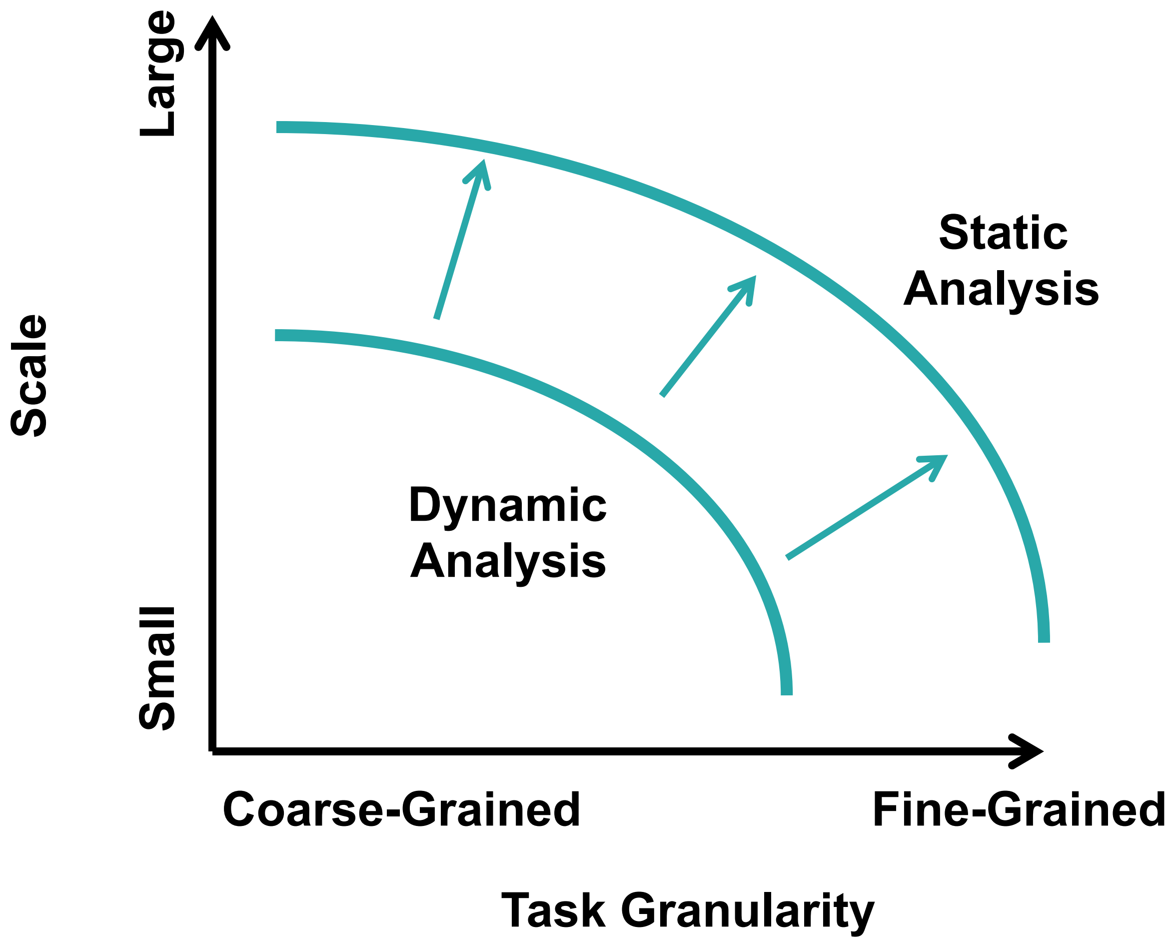
Work In Progress: SPMD



Work In Progress: SPMD

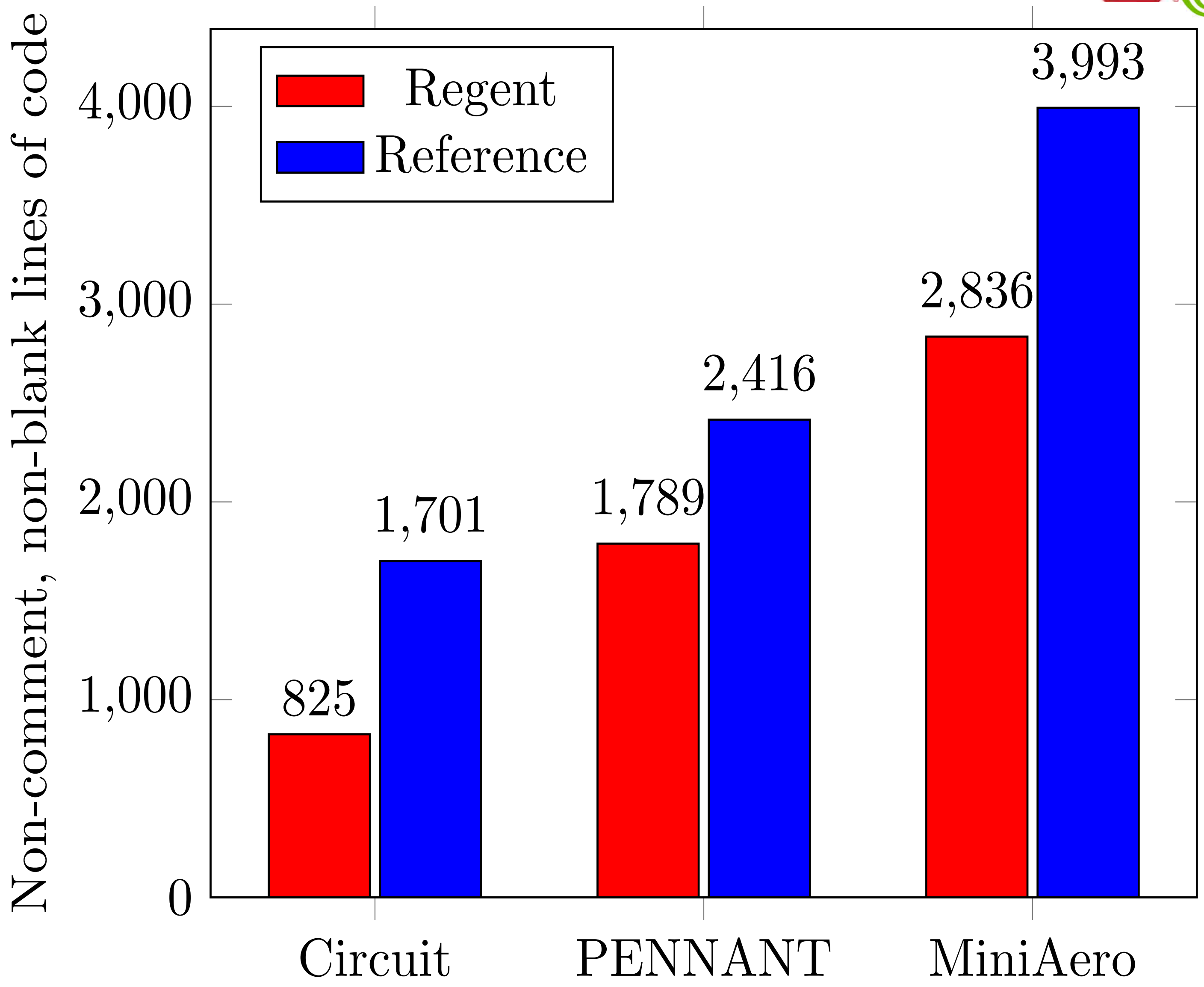


Pushing the Performance Envelope with Compilation

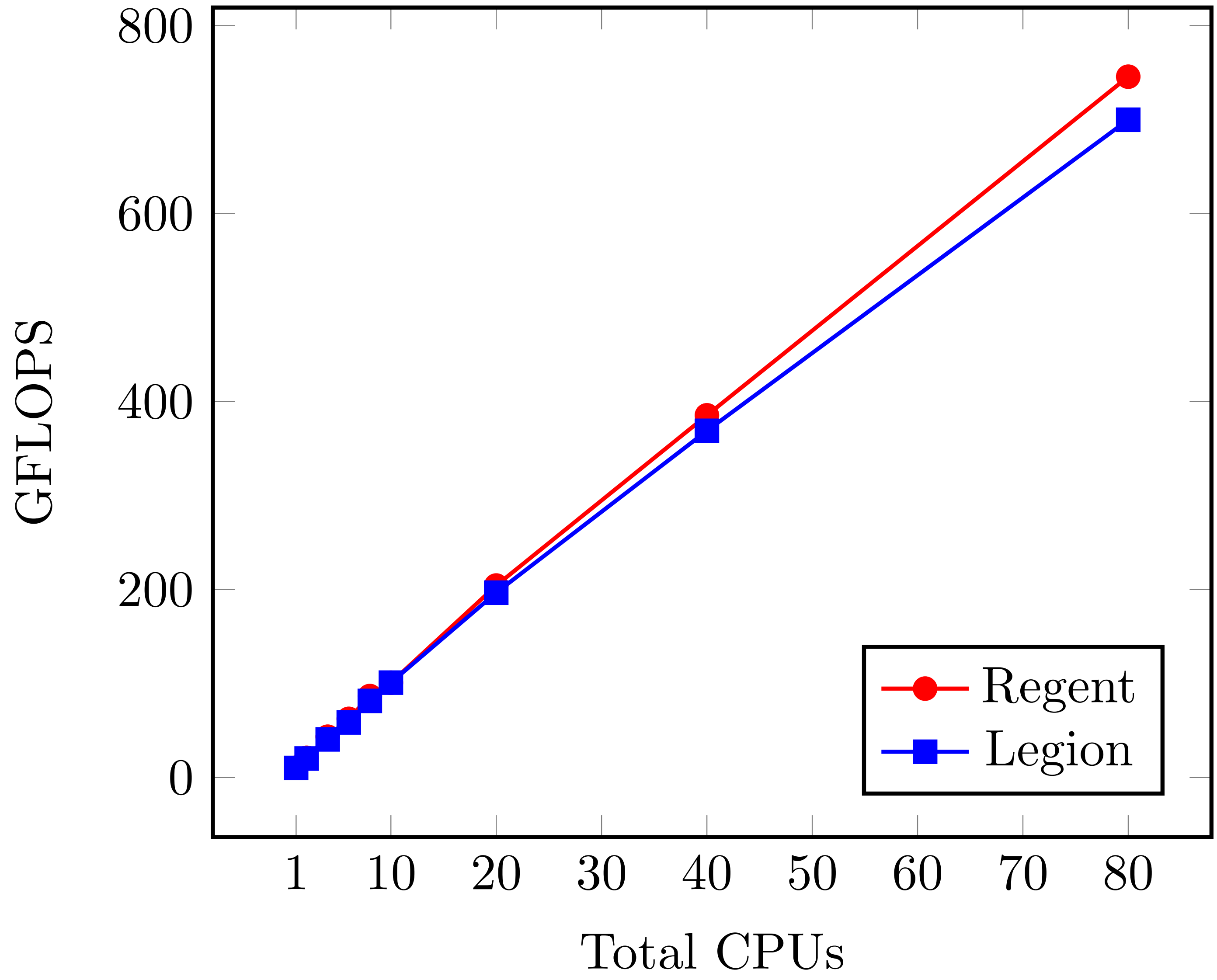


Questions?

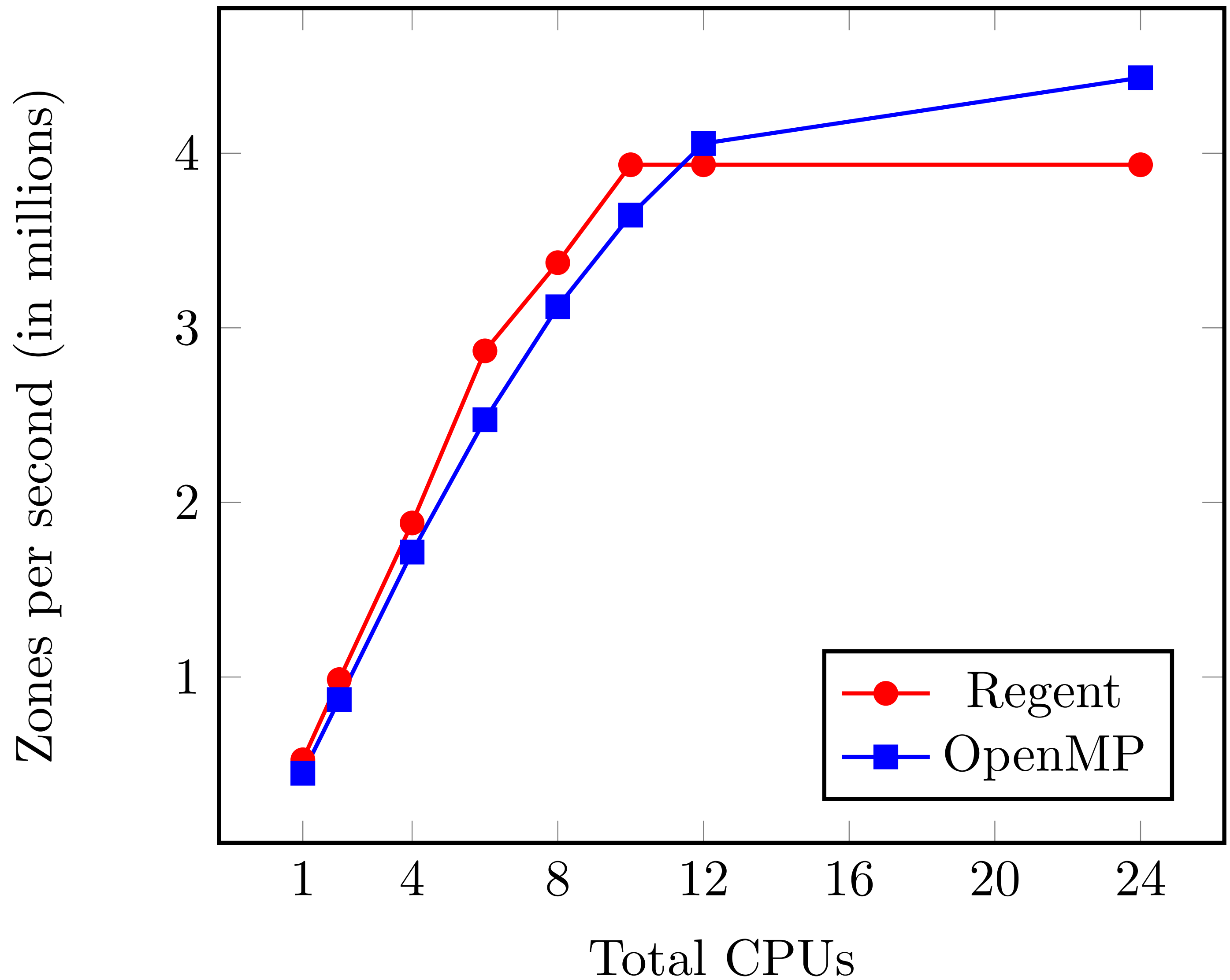
Lines of Code



Circuit: Absolute Performance

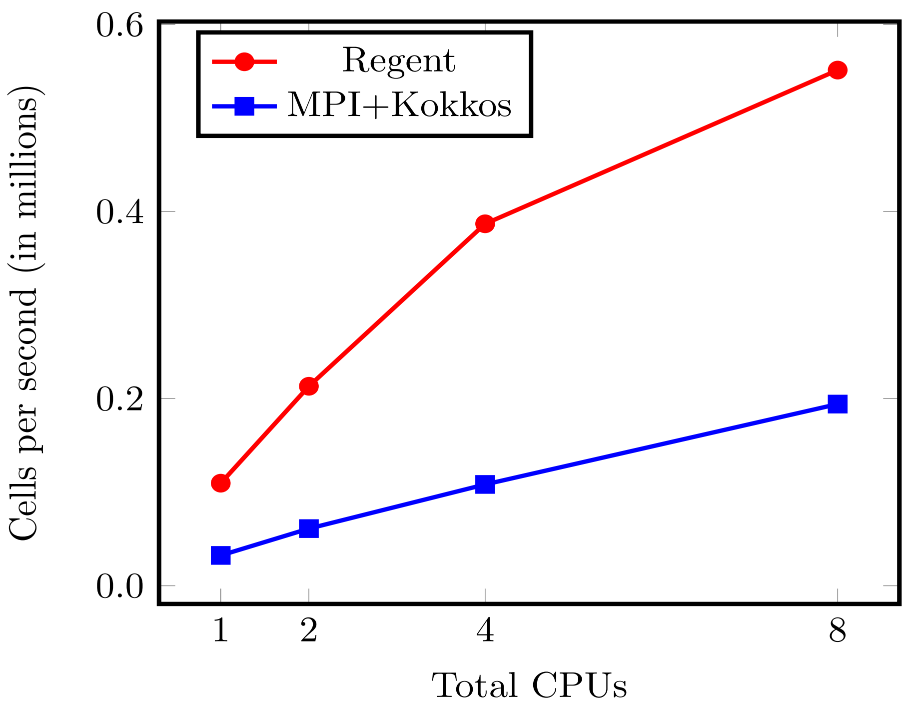


PENNANT: Absolute Performance

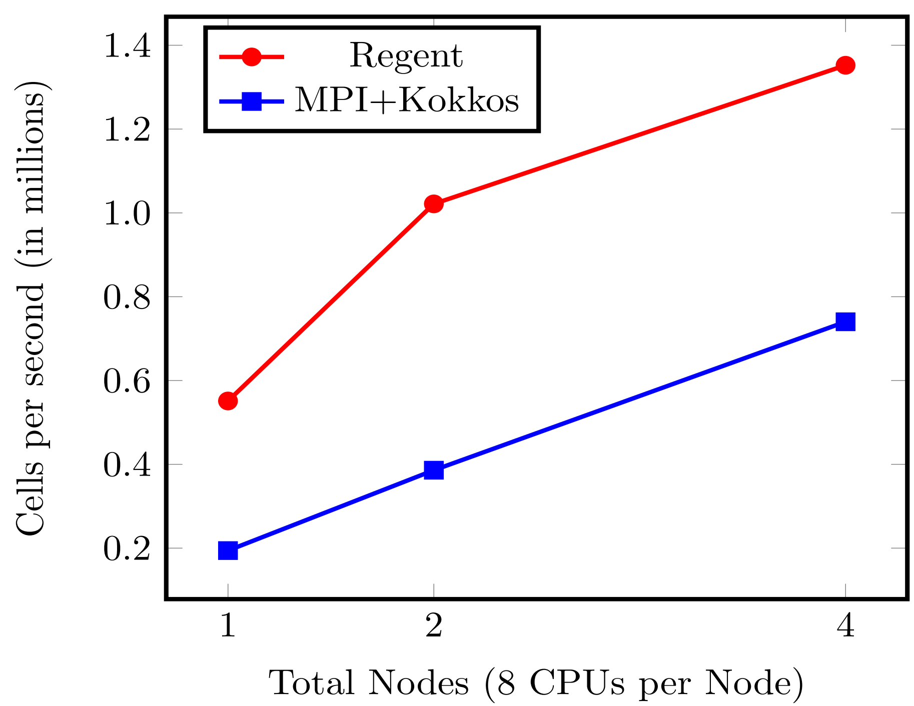


MiniAero: Absolute Performance

Single Node

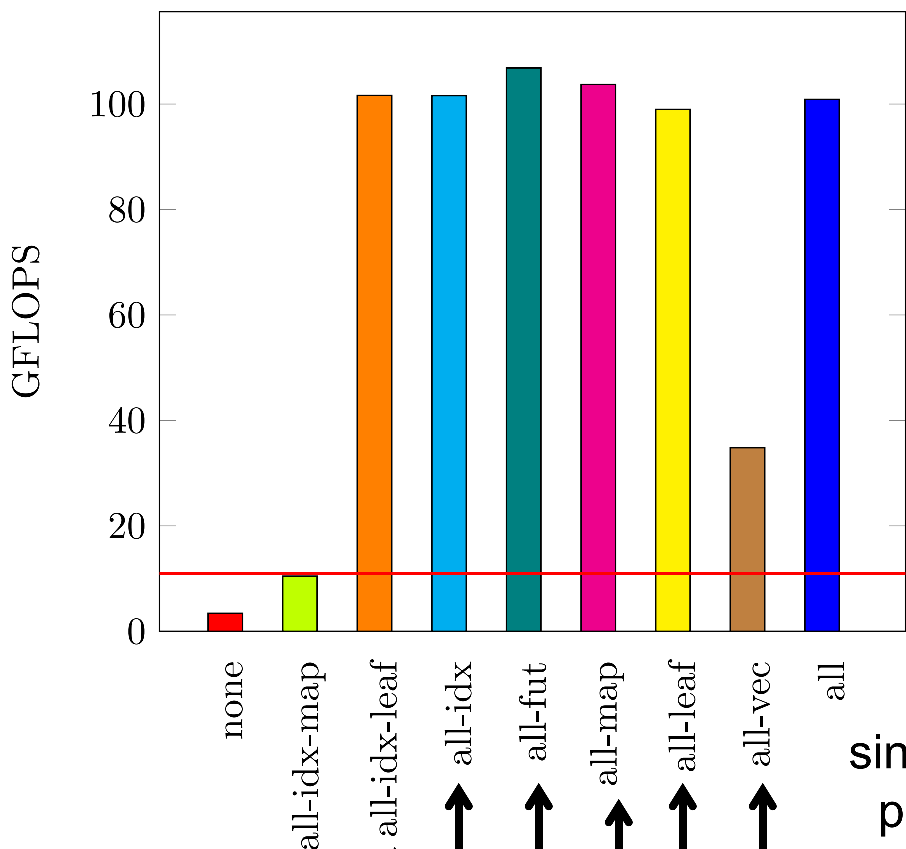


Multiple Nodes

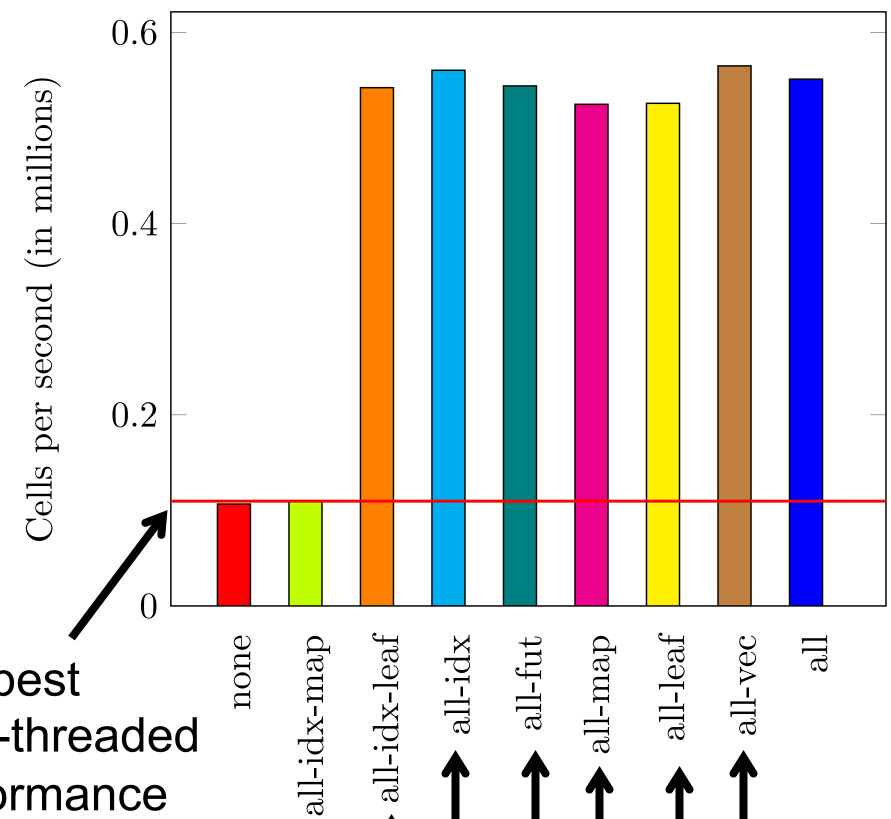


Impact of Optimizations

Circuit



MiniAero



best
single-threaded
performance

individual optimizations disabled

pairs of optimizations disabled

Impact of Optimizations

PENNANT