Rat: 一种函数式并行程序语言

答辩人： 苏醒
指导教师： 窦文华 教授

计算机所 641 教研室

2013-10-22

研究内容

本课题研究点主要包括：

- 并行程序语言的语法设计
  - 定位：Domain Specific Language
  - 目标：在保证效率的前提下提高易用性
  - 内容：类型系统，表达式种类，高阶特性

研究内容

本课题研究点主要包括：

- 并行程序语言的语法设计
  - 定位：Domain Specific Language
  - 目标：在保证效率的前提下提高易用性
  - 内容：类型系统，表达式种类，高阶特性
- 语言的实现技术
  - Runtime System：为内存管理、线程管理、任务划分等并行细节提供支持
  - 异构硬件的利用：CPU 与 GPU 的协同工作
  - 优化措施：存储器优化以及 GPU 片上共享存储器的优化

并行程序设计实例：向量内积

考虑一个经典数据并行问题：

向量内积

$$d = u \cdot v = \sum_{k=1}^{n} u_k v_k$$

并行程序设计实例：向量内积

考虑一个经典数据并行问题：

向量内积

$$d = u \cdot v = \sum_{k=1}^{n} u_k v_k$$

C 语言顺序实现

```
float dot(float* v1, float* v2, size_t len) {
    float retval = 0;
    for (int i = 0; i < len; ++i) {
        retval += v1[i] * v2[i];
    }
    return retval;
}
```

并行程序设计实例：向量内积

考虑一个经典数据并行问题：

向量内积

$$d = u \cdot v = \sum_{k=1}^{n} u_k v_k$$

Rat 实现

```
dot :: Numeric -> [a] -> [a] -> a
dot v1 v2 = fold + (zipWith * v1 v2)
```

Rat 的数据类型

Rat 采用静态数据类型系统，执行编译期类型检查（FIXME：这里的举例结合矩阵乘法的例子）

- 标量类型
  - 定长整数
    `(U)Int8, (U)Int16, ... (U)Int64, (U)Int`
  - 浮点数
    `Float, Double`
- 向量类型
  [e]包含元素类型为e（标量类型）的向量类型
- 结构类型
  (Int32, Double ...)
- 函数类型
  从`Int32`到`Int32`的函数类型 `Int32 -> Int32`

多态—type class

Rat 支持 type class，即某一类具有共同性质的数据类型。支持多态可以避免为不同类型的数据分别定义相同的操作。

- 数值类型 Numeric

```
class Numeric a where
  + :: a -> a ->
```

- 幺半群类型

```
class Monoid a where
  mempty :: a
  mappend :: a -> a -> a
```

多态—type class

Rat 支持 type class，即某一类具有共同性质的数据类型。支持多态可以避免为不同类型的数据分别定义相同的操作。

- 数值类型 Numeric

```
class Numeric a where
  + :: a -> a ->
```

- 幺半群类型

```
class Monoid a where
  mempty :: a
  mappend :: a -> a -> a
```

recall: C++ template

```
template <typename T>
T plus(T a, T b) { return a + b; }
```

type class 实例

以 0 为幺元，加法运算为结合律二元运算的 Float 上的幺半群

```
instance Monoid Float where
  mempty = 0
  mappend = +
```

type class 实例

以 0 为幺元，加法运算为结合律二元运算的 Float 上的幺半群

```
instance Monoid Float where
  mempty = 0
  mappend = +
```

type class 的另一作用：为可并行化操作提供约束。

```
parallelSum :: Monoid a => [a] -> a
parallelSum v = fold mappend v
```

函数式语言特性

- “一等公民”（first-class citizen）
- 纯函数特性（pure functional）
- 柯里化（curried）
- 恒值对象（immutable object）

函数式语言特性

“一等公民”

函数可以是

- 其他函数的参数

```
apply :: (a -> b) -> a -> b
apply f a = f a
```

函数式语言特性

“一等公民”

函数可以是

- 其他函数的参数

```
apply :: (a -> b) -> a -> b
apply f a = f a
```

- 函数的返回值

```
addF :: Float -> (Float -> Float)
addF f = + f
```

函数式语言特性

“一等公民”

函数可以是

- 其他函数的参数

```
apply :: (a -> b) -> a -> b
apply f a = f a
```

- 函数的返回值

```
addF :: Float -> (Float -> Float)
addF f = + f
```

- 匿名函数（lambda 表达式）

```
\ x y -> power x y
```

函数式语言特性

纯函数特性（pure functional）

也称“无副作用”（side-effect free），即只要给定同样的输入，就得到同样的输出。side-effect free 可以大大简化程序语义的分析。

- 纯函数举例：

  `sqrt, pow, exp ...`

- 副作用函数：

  `printf, readline ...`

函数式语言特性

柯里化（curried）

- 函数的形式参数支持部分实例化。

```
+ :: Float -> Float -> Float
+ 2 :: Float -> Float
```

函数式语言特性

柯里化（curried）

- 函数的形式参数支持部分实例化。

```
+ :: Float -> Float -> Float
+ 2 :: Float -> Float
```

- 另一种视角：将 n 元函数看作一个 1 元函数，该 1 元函数的返回类型是一个(n-1)元函数

```
+ :: Float -> Float -> Float
+ :: Float -> (Float -> Float)
```

函数式语言特性

恒值对象（immutable object）

无赋值操作，一个对象只在被定义的时候赋值，并在之后的生命周期中不再变更

```
-- definition
a = initialValue
-- assignment, compile error!
a = anOtherValue
```

函数式语言的优势

为什么采用函数式语言设计？

- 语义精确，容易分析
  数值计算领域关注计算，满足“side-effect free”的要求
- 描述问题本身的求解，隐藏计算机实现细节

函数式语言的优势

为什么采用函数式语言设计？

- 语义精确，容易分析
  数值计算领域关注计算，满足“side-effect free”的要求
- 描述问题本身的求解，隐藏计算机实现细节

其他编程范式

- 过程式（命令式）语言
  抽象级别低，细节隐藏差
- 对象式语言
  应用于数值计算领域无优势

编程界面

Rat 语言包括两类数据操作：

- 标量操作 Scalar Operation(SOP)
  - 标量原语 Scalar Primitive（SP）：基本的算数逻辑运算
    `+`, `-`, `*`, `/`, `mod`, `exp`, `sqrt`, ...
  - 用户自定义标量函数

编程界面

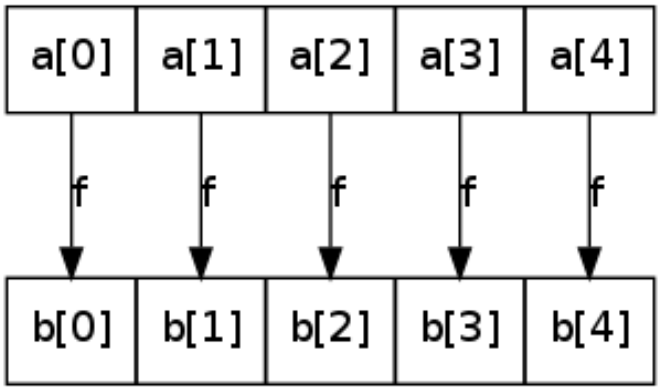
Rat 语言包括两类数据操作：

- 标量操作 Scalar Operation(SOP)
  - 标量原语 Scalar Primitive（SP）：基本的算数逻辑运算
    `+`, `-`, `*`, `/`, `mod`, `exp`, `sqrt`, ...
  - 用户自定义标量函数
- 向量操作 Vector Operation(VOP)
  - 向量原语 Vector Primitive(VP)
    `map`, `scan`, `permute`, `slice`, `compact`, `sparse`, `sort`, `zip`, `random`
  - 用户自定义向量函数

向量原语

向量原语是并行数据操作的核心

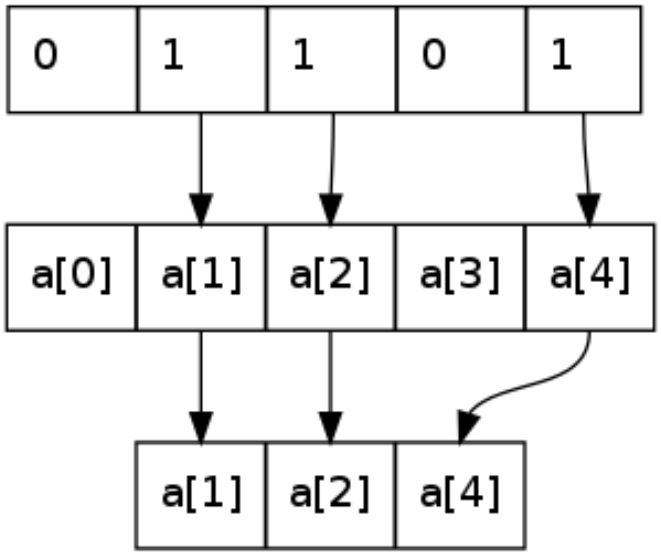
Figure: map



向量原语

向量原语是并行数据操作的核心

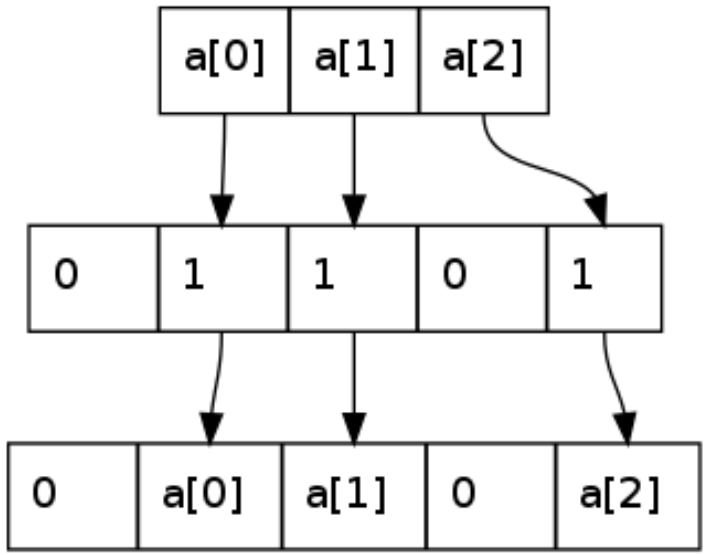
Figure: compact



向量原语

向量原语是并行数据操作的核心

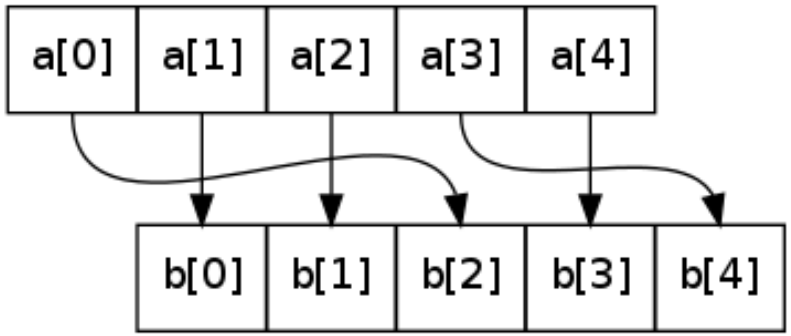
Figure: sparse



向量原语

向量原语是并行数据操作的核心

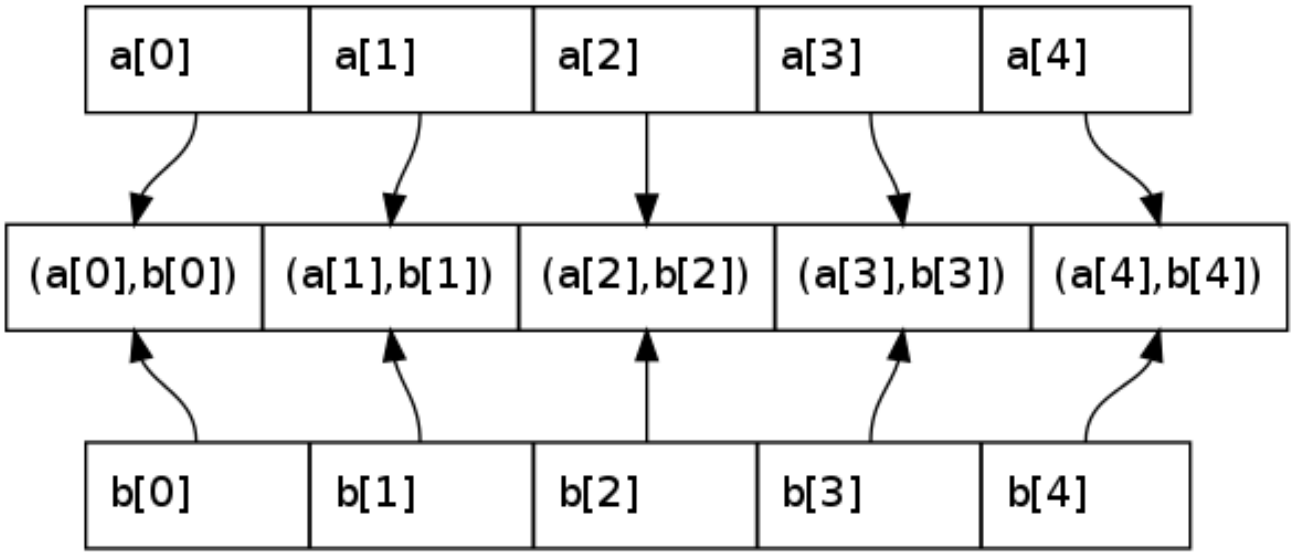
Figure: permute



向量原语

向量原语是并行数据操作的核心

Figure: zip



一个完整实例

n-body 问题

```
solveNBodyOneStep :: Float -> [Cell] -> [Cell]
solveNBodyOneStep t cells =
  zipWith3 makeCell masses newPositions newVelocities
  where
    positions = map position cells -- original positions
    velocities = map velocity cells -- original velocity
    calcSingleAcc c1 c2 =
      (g * mass c2) / (squre (position c1 - position c2) )
    calcAccelerate cell =
      fold + (map (calcSingleAcc cell) cells)
    accelerates = map calcAccelerate cells -- accelerate
    newVeciloties =                  -- new velocity
      zipWith (\v a -> v + a * t) velocities accelerates
    newPositions =                   -- new positions
      zipWith (\p v -> p + v * t) positions velocities
```

小结：Rat 语言的语法特点

- 高层的编程界面，数学化的程序表达，适用于数据并行编程问题
- 细节隐藏，消除内存管理、线程管理、消息传递等编程复杂性
- 强静态类型系统，极大降低运行期错误发生的可能性

源程序分析

level-1 语法树

level-1 语法树即为源程序树，包括以下结点类型：

Table: level-1 语法树的结点类型

type-def	variable-decl	variable-def
literal	variable-ref	function-app
lambda-exp	conditional	vector-comprehension
vector-element-ref	vector-slice-ref	local-binding

源程序分析

level-2 语法树

由 level-1 语法树变换得到的 level-2 语法树结构更为简化。

Table: level-2 语法树的结点类型

literal	variable-ref	function-app
lambda-exp	conditional	

Core 语言

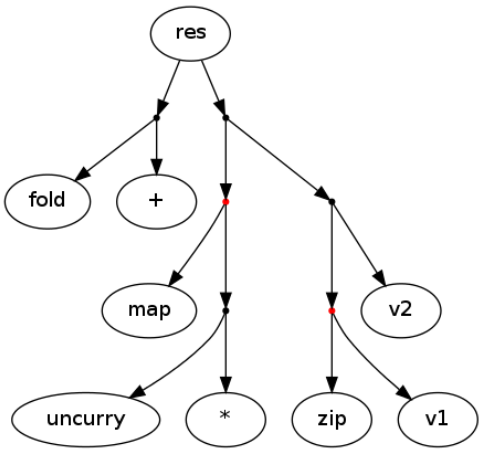
Rat 核心语言 Core，将 level-2 语法树中的柯里化（curried）函数变换为反柯里化（uncurried）形式。程序的优化与翻译在 Core 语言上执行。

Core 语言

实例：求解向量内积

```
dot :: [Float] -> [Float] -> Float
res v1 v2 = fold + (map (uncurry *) (zip v1 v2))
```

Figure: level-2 语法树到 Core 语言语法树的变换

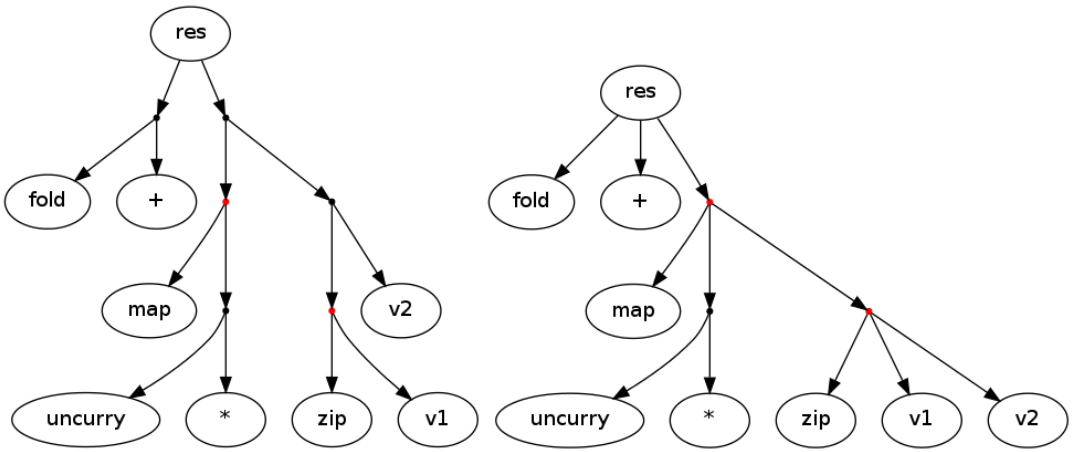


Core 语言

实例：求解向量内积

```
dot :: [Float] -> [Float] -> Float
res v1 v2 = fold + (map (uncurry *) (zip v1 v2))
```

Figure: level-2 语法树到 Core 语言语法树的变换



Core 语言的翻译

Core 语言运行于 Rat 虚拟机之上。Rat 虚拟机抽象层次较高，它的基本指令由 Rat 的内建标量函数与向量原语 VP 组成。

- 标量原语 SP

  `+, -, *, /, mod, exp, sqrt, ...`

- 向量原语 VP

  `map, scan, permute, slice, compact, sparse, sort, zip, random`

Core 语言的翻译

实例：求单精度浮点复数的模

$C = x + yi$ $\|C\| = \sqrt{x^2 + y^2}$

Core 语言的翻译

实例：求单精度浮点复数的模

$C = x + yi$ $\|C\| = \sqrt{x^2 + y^2}$

```
squareSumRoot :: Float -> Float -> Float
squareSumRoot x y = sqrt (x * x + y * y)
```

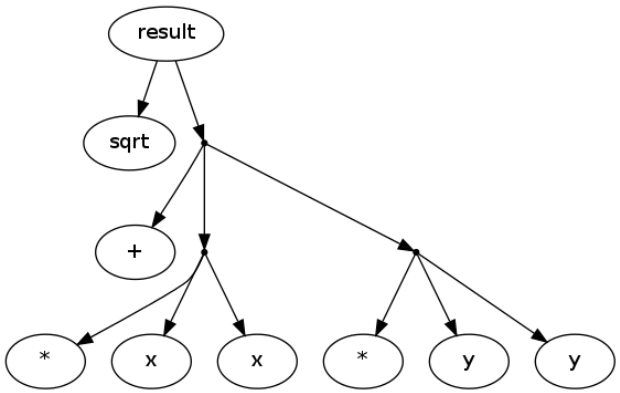
Core 语言的翻译

实例：求单精度浮点复数的模

$C = x + yi$ $\quad\quad$ $\|C\| = \sqrt{x^2 + y^2}$

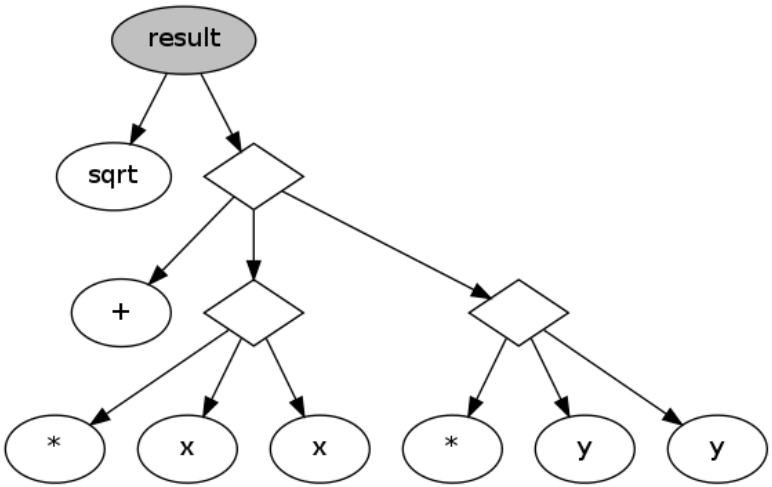
```
squareSumRoot :: Float -> Float -> Float
squareSumRoot x y = sqrt (x * x + y * y)
```

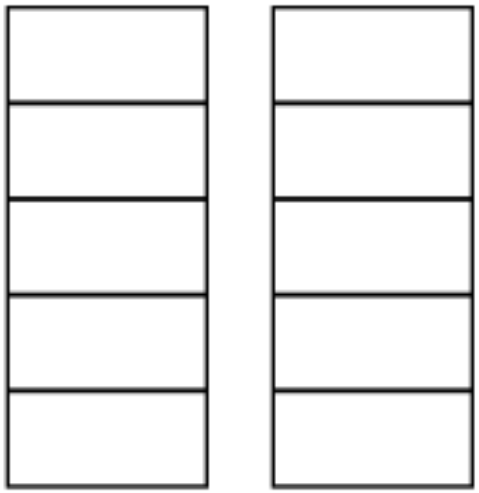
Figure: 复数求模函数的 Core 树



Core 语言虚拟机

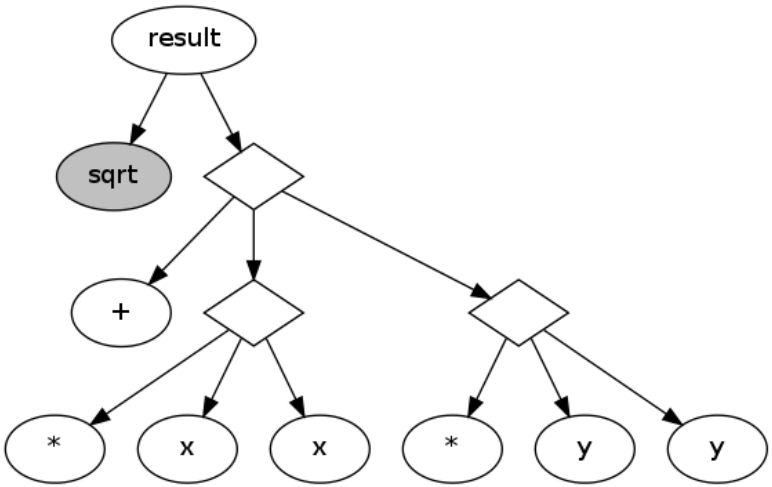
以原语操作为基本指令的虚拟机。

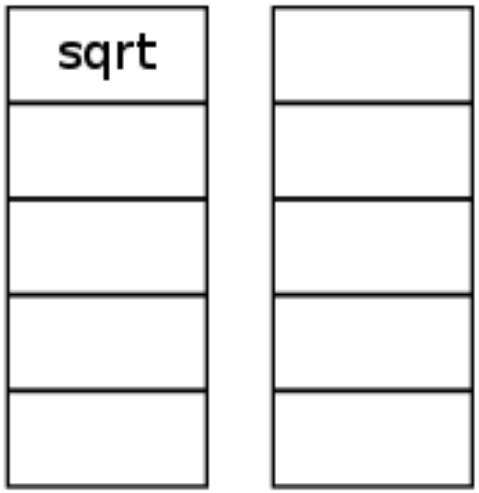




Core 语言虚拟机

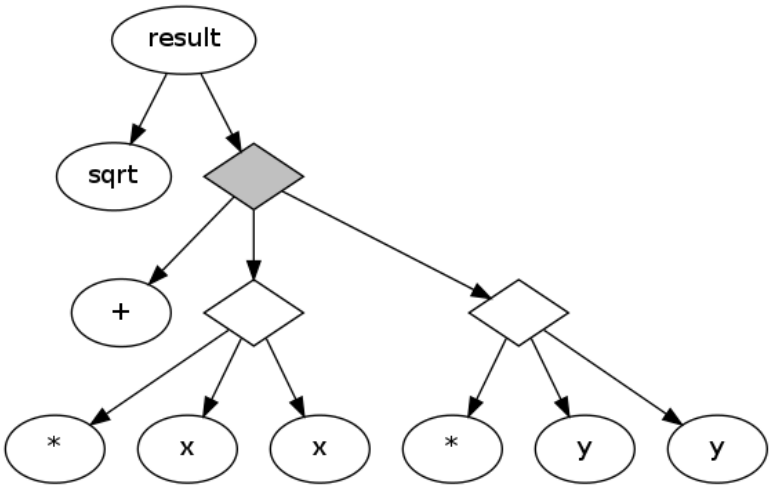
以原语操作为基本指令的虚拟机。

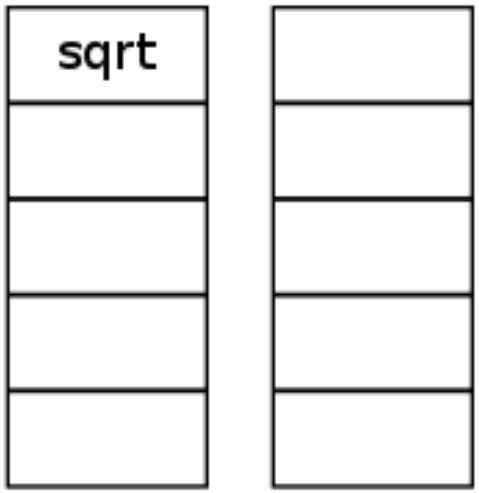




Core 语言虚拟机

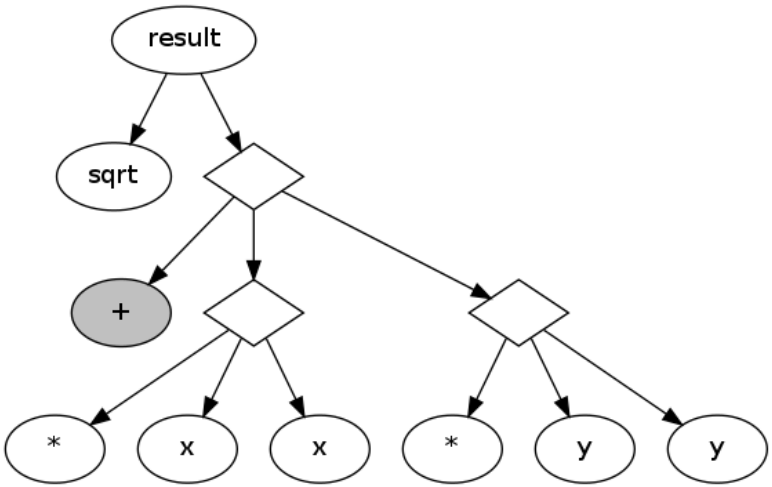
以原语操作为基本指令的虚拟机。

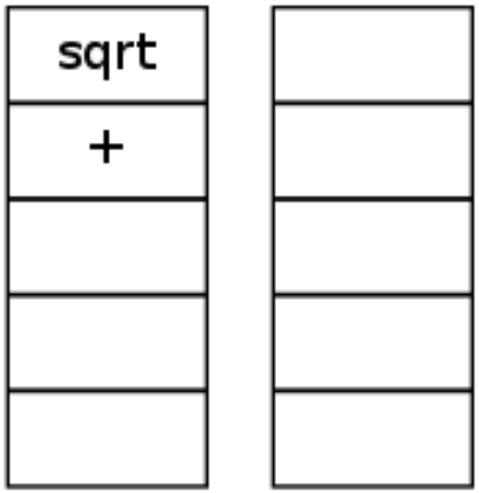




Core 语言虚拟机

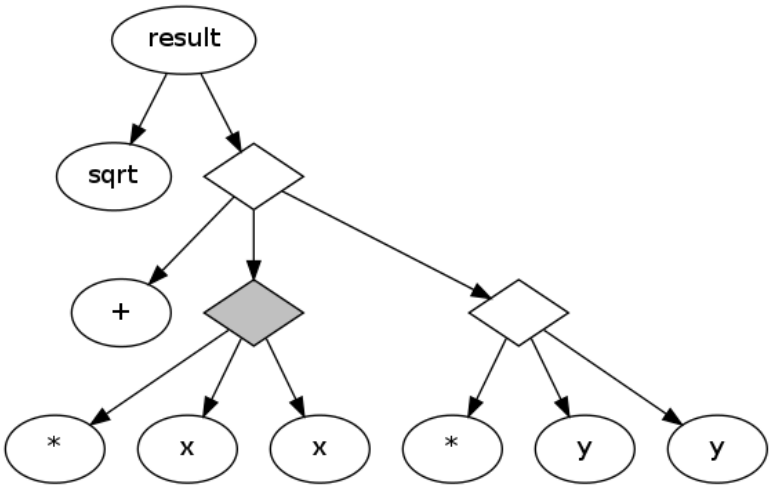
以原语操作为基本指令的虚拟机。

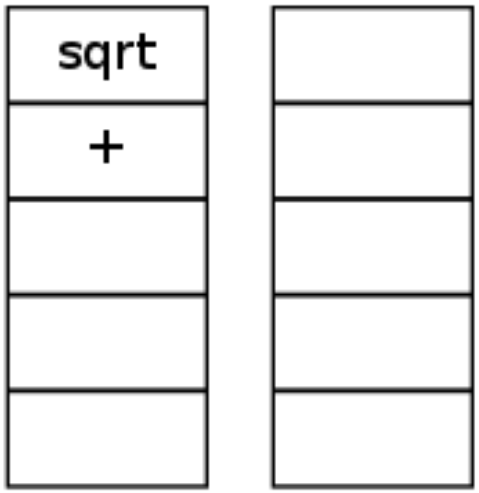




Core 语言虚拟机

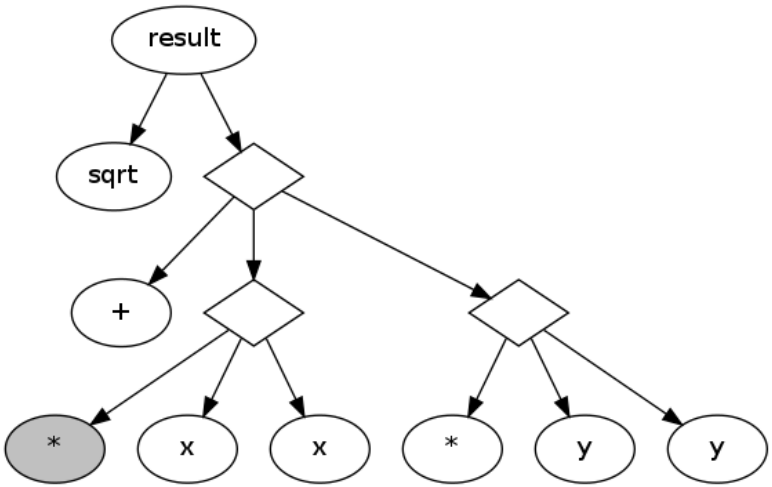
以原语操作为基本指令的虚拟机。

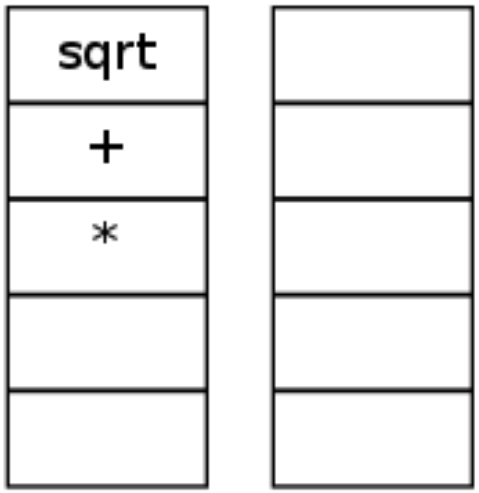




Core 语言虚拟机

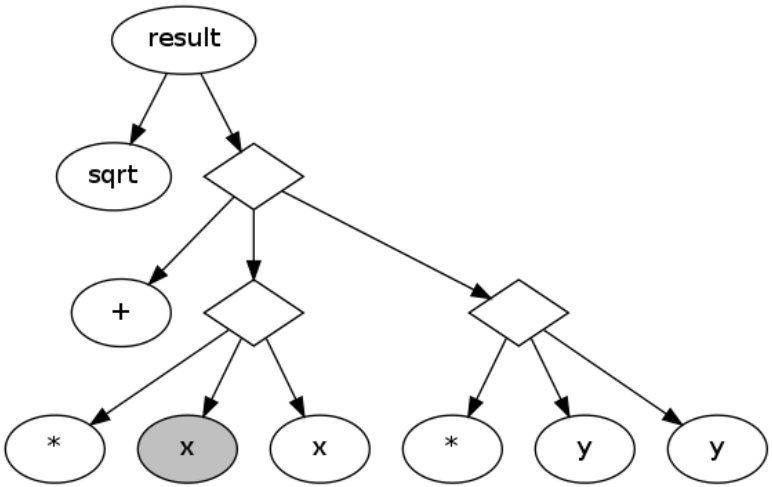
以原语操作为基本指令的虚拟机。

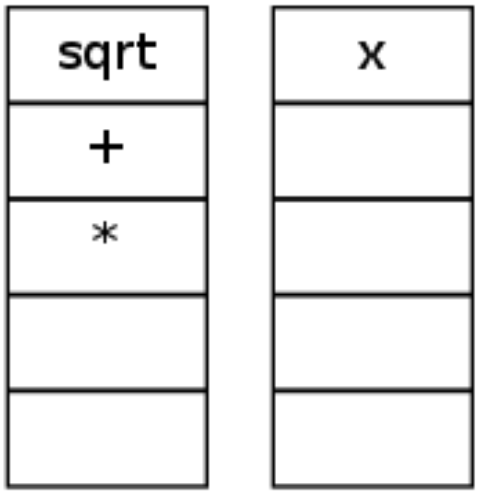




Core 语言虚拟机

以原语操作为基本指令的虚拟机。

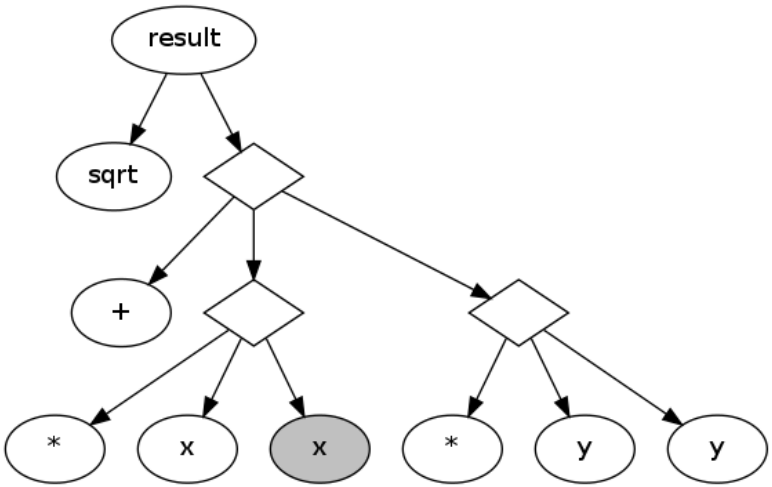


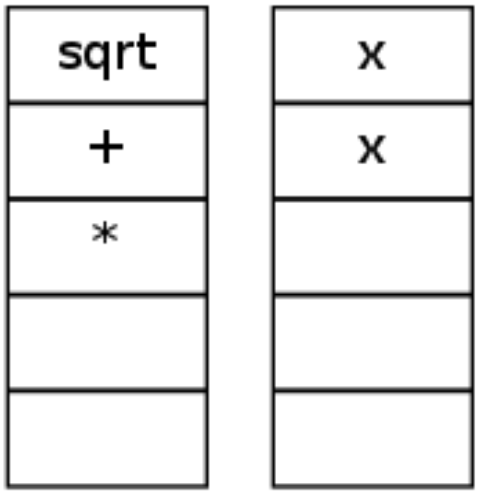



```
push(x)
```

Core 语言虚拟机

以原语操作为基本指令的虚拟机。

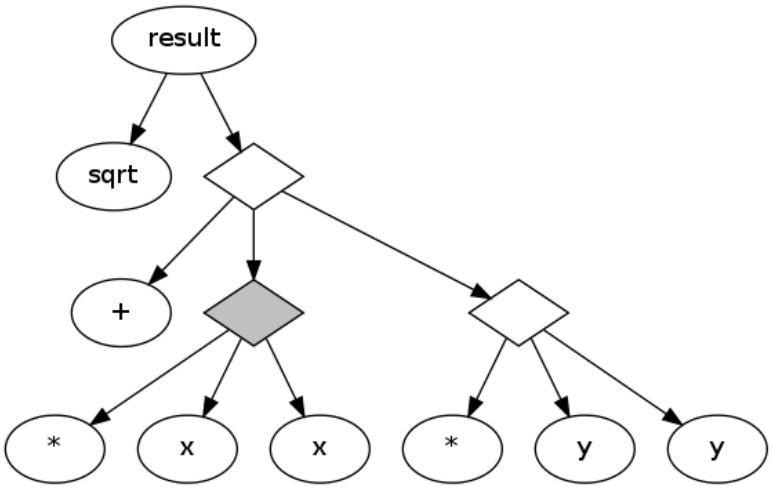


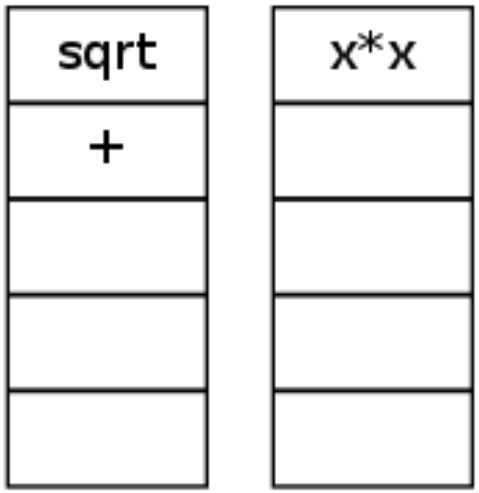



```
push(x); push(x)
```

Core 语言虚拟机

以原语操作为基本指令的虚拟机。

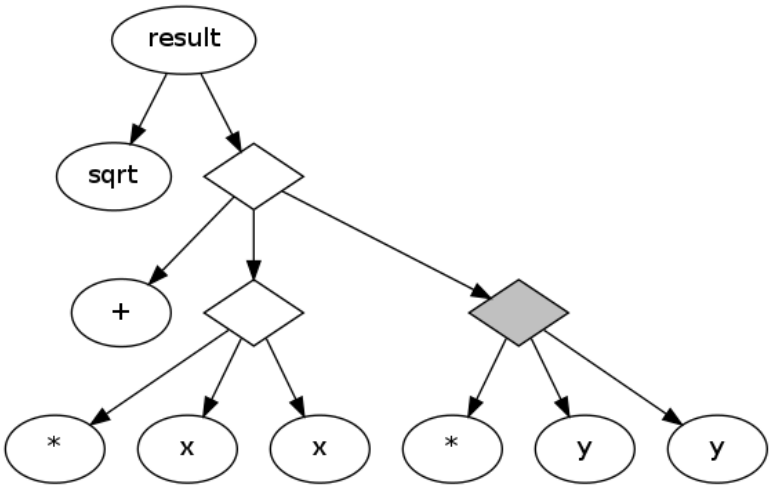


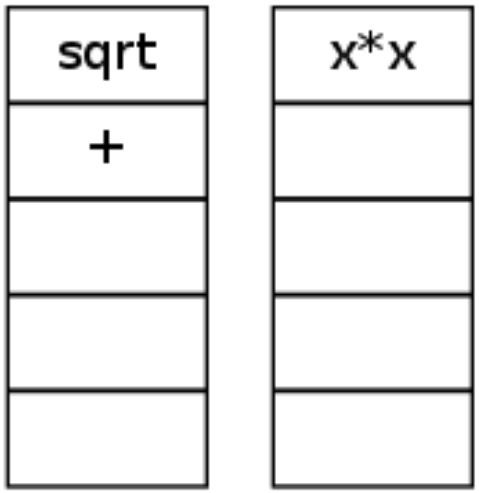



```
push(x); push(x); multiply
```

Core 语言虚拟机

以原语操作为基本指令的虚拟机。

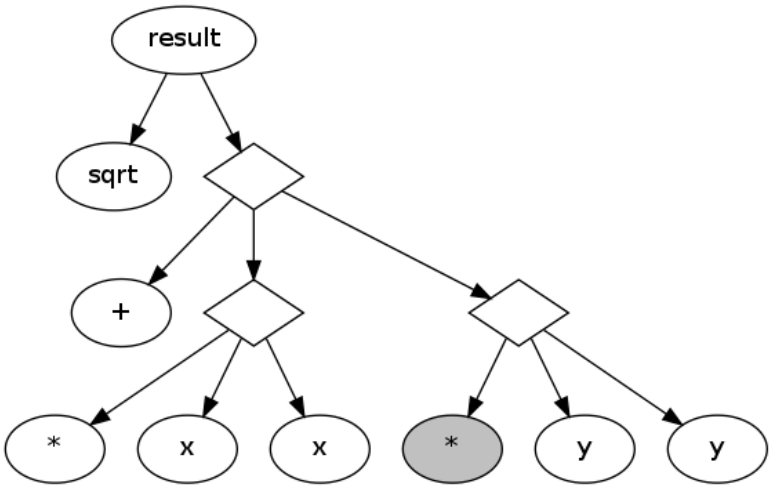


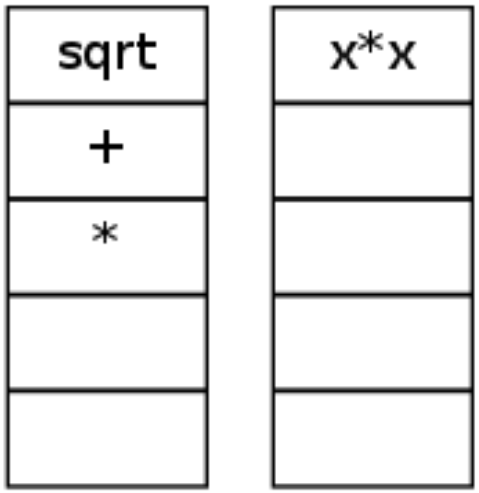



```
push(x); push(x); multiply
```

Core 语言虚拟机

以原语操作为基本指令的虚拟机。

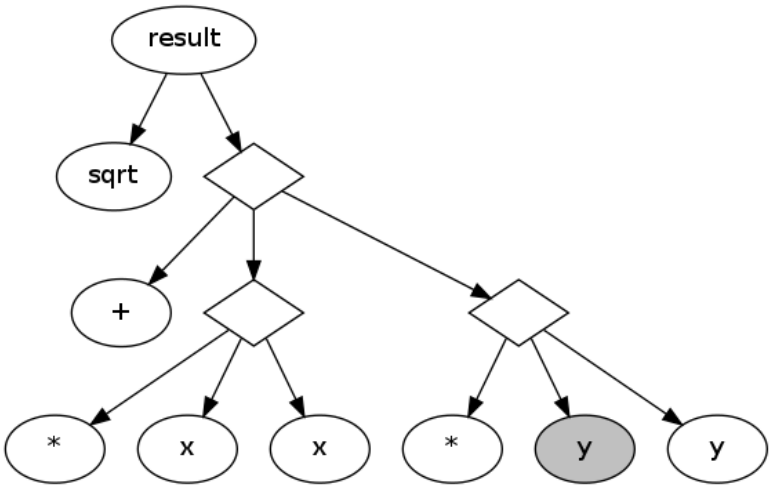


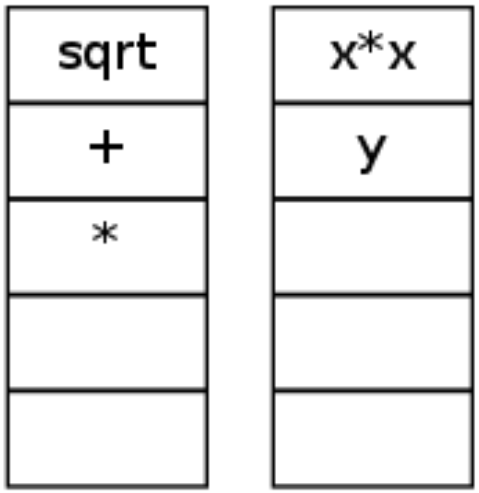



```
push(x); push(x); multiply
```

Core 语言虚拟机

以原语操作为基本指令的虚拟机。

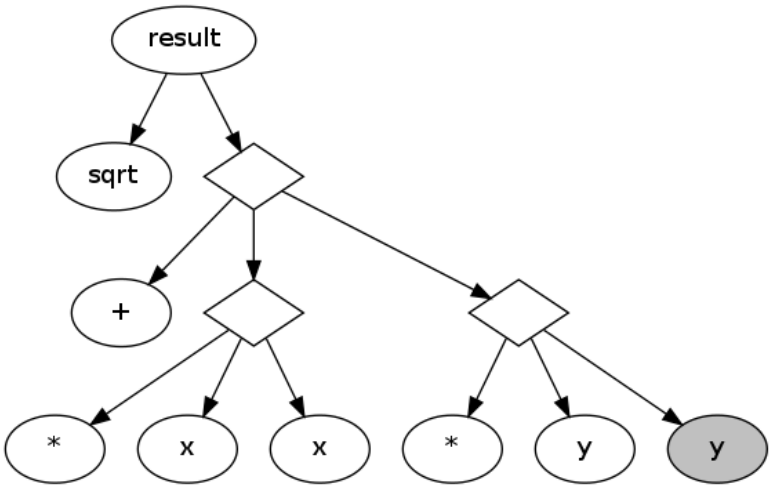


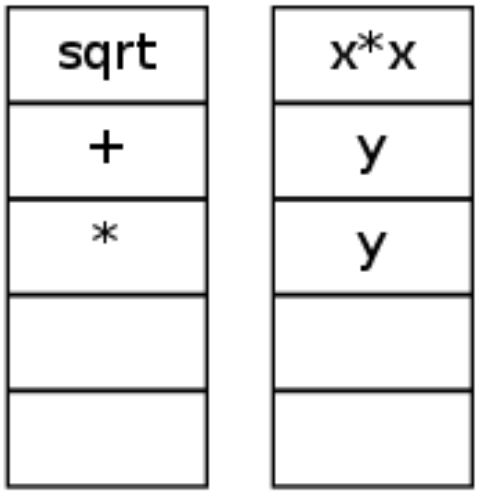



```
push(x); push(x); multiply;
push(y);
```

Core 语言虚拟机

以原语操作为基本指令的虚拟机。

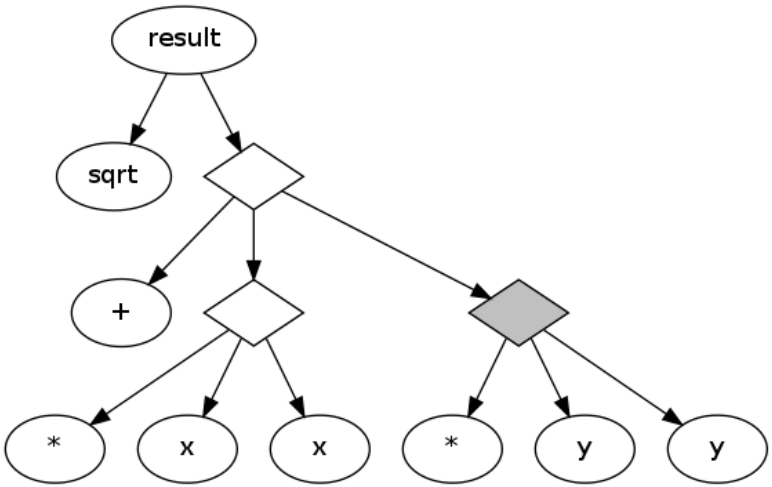


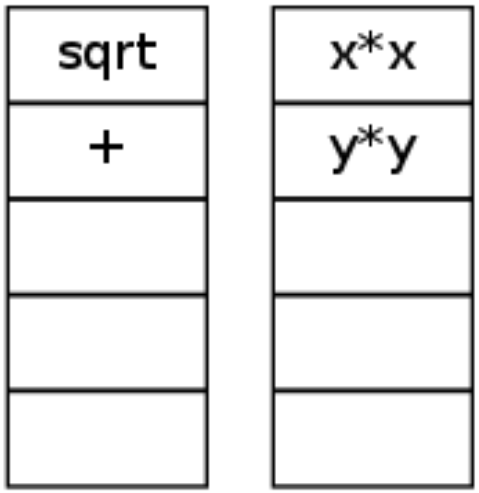



```
push(x); push(x); multiply;
push(y); push(y);
```

Core 语言虚拟机

以原语操作为基本指令的虚拟机。

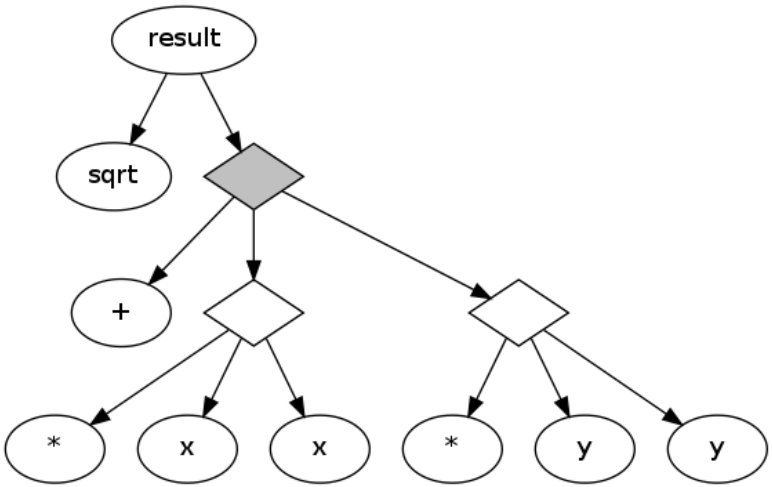


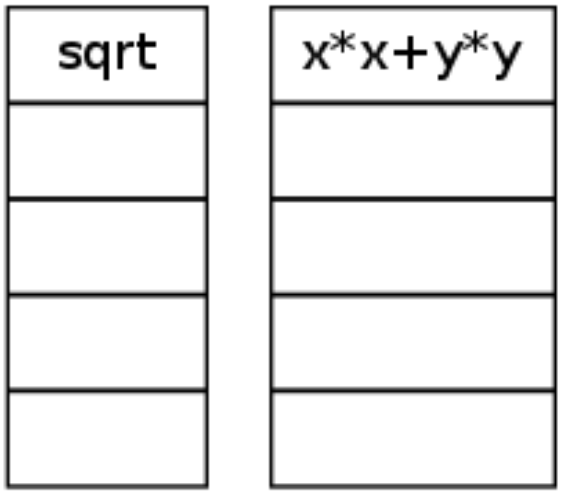



```
push(x); push(x); multiply;
push(y); push(y); multiply;
```

Core 语言虚拟机

以原语操作为基本指令的虚拟机。

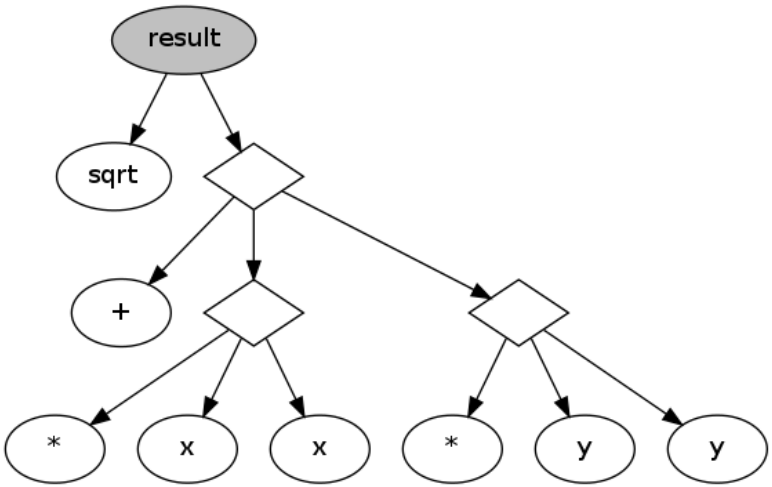


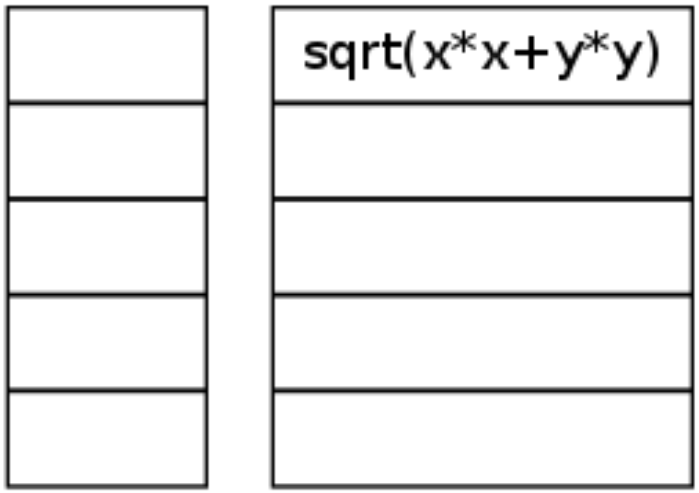



```
push(x); push(x); multiply;
push(y); push(y); multiply;
plus;
```

Core 语言虚拟机

以原语操作为基本指令的虚拟机。






```
push(x); push(x); multiply;
push(y); push(y); multiply;
plus; sqrt;
```

Core 虚拟机的实现

标量原语 SP 的翻译

内建标量函数被直接翻译成 C 表达式或函数调用

`+ a b` → `(a + b)`

`exp n` → `exp(n)`

Core 虚拟机的实现

向量原语 VP 的翻译

向量原语 VP 的翻译过程采用原语框架实例化实现。

- 原语框架
- 框架实例化

向量原语 VP 的翻译

原语框架

map 原语框架

```
typedef out_type (*scalar_op) (in_type);
int map(in_type* in, size_t inlen,
        out_type* out, size_t outlen,
        scalar_op sop) {
    size_t inl , outl, size = outlen;
    while (inl = partition(in, PARTITION_SIZE)) {
        outl = solve_part(in, inl, out, outlen);
        in += inl;
        out += outl;
        outlen -= outl;
    }
    return (size-outlen);
}
```

向量原语 VP 的翻译

框架实例化

map 原语实例化

```
typedef float (*scalar_op13) (float);
int map27(float* in, size_t inlen,
          float* out, size_t outlen,
          scalar_op13 sop) { ... }

int main() {
    ...
    sop = sqrt;
    map27(in, ilen, out, olen, sop);
}
```

优化技术

Rat 程序的优化技术简历在 Core 语言之上。主要的优化技术包括：

- 存储空间优化
- 向量原语聚合（VP fusion）
- 向量原语重排（VP reorder）

存储空间优化

逻辑上，每一个 VP 作用在一个向量上都会得到一个新的向量。向量要占用大量的存储器资源，对向量存储空间的优化利用意义重大。存储空间可以优化措施的情况包括：

存储空间优化

逻辑上，每一个 VP 作用在一个向量上都会得到一个新的向量。向量要占用大量的存储器资源，对向量存储空间的优化利用意义重大。存储空间可以优化措施的情况包括：

- 如果原有的向量不再被使用，那么它使用的内存区域就可以回收重用
- 某些 VP 的并行实现可以原地完成，或者某些条件下可以完成

存储空间优化

向量内存重用

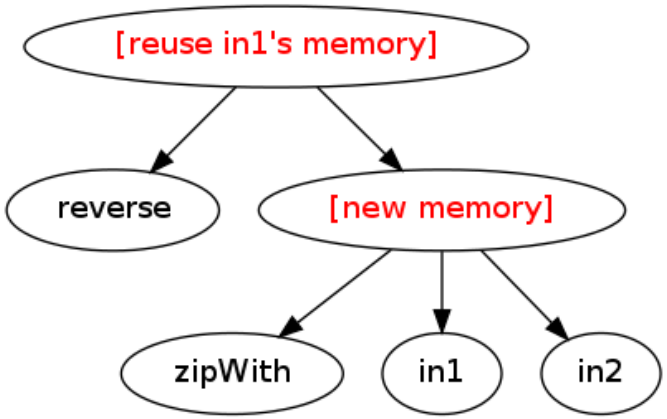
对于 Rat 程序中的任意向量，在编译期都可以确定它生命周期终点。在这一点之后的时刻，该向量的数据不在会被使用，那么它所占用的空间可以分配给其他向量使用。

存储空间优化

向量内存重用

对于 Rat 程序中的任意向量，在编译期都可以确定它生命周期终点。在这一点之后的时刻，该向量的数据不在会被使用，那么它所占用的空间可以分配给其他向量使用。

```
foo :: [Float] -> [Float] -> [Float]
foo v1 v2 = reverse (zipWith + v1 v2)
```



存储空间优化

原地并行 VP

向量原语的原地并行可执行能力如下表

Table: 各向量原语的原地并行执行能力

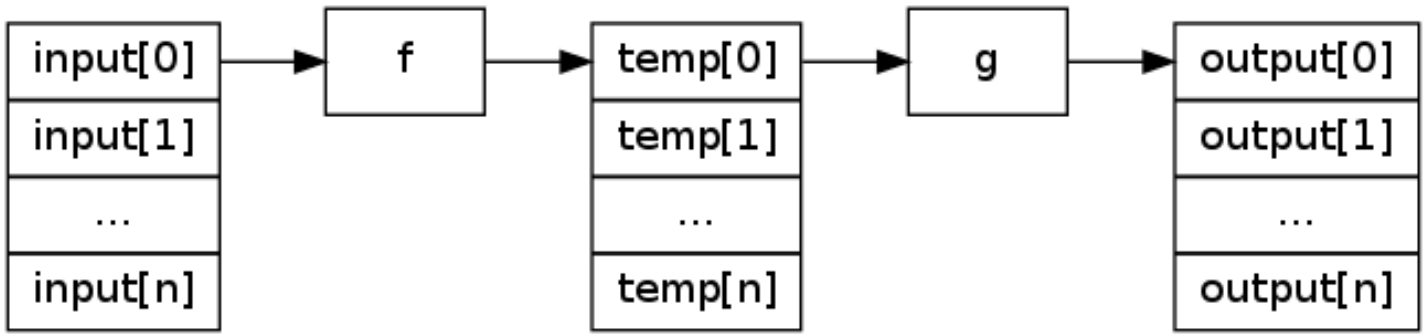
map	yes	scale	仅当内存缩小时
slice	yes	scan	no
compact	需要同步	sort	no
sparse	no	zip	yes
permute	需要同步	random	yes

向量原语聚合 VP fusion

向量原语聚合旨在合并相邻的 VP，以提高执行效率。

```
outputV = map f (map g inputV)
outputV = map (f . g) inputV
```

Figure: VP fusion 实例

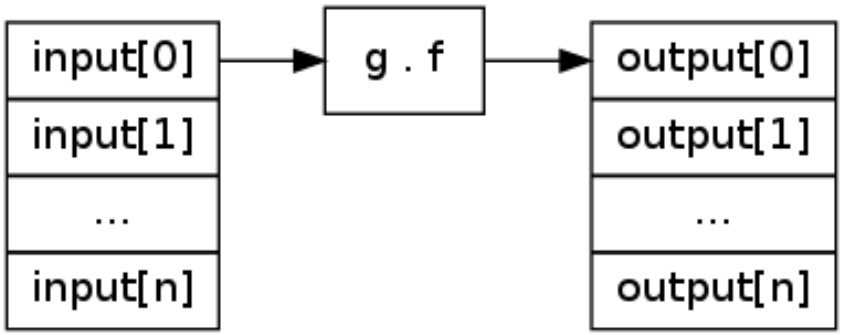


向量原语聚合 VP fusion

向量原语聚合旨在合并相邻的 VP，以提高执行效率。

```
outputV = map f (map g inputV)
outputV = map (f . g) inputV
```

Figure: VP fusion 实例

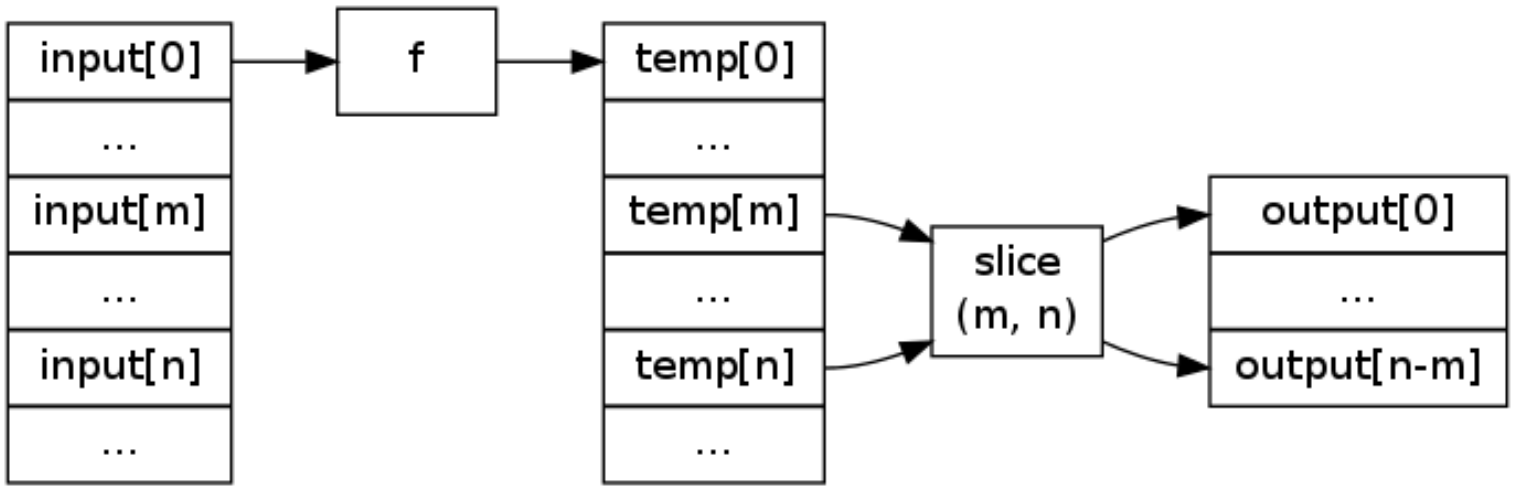


向量原语重排 VP reorder

向量原语重排旨在交换可以换序的 VP，以提高执行效率

```
outputV = map f (map g inputV)
outputV = map (f . g) inputV
```

Figure: VP reorder 实例

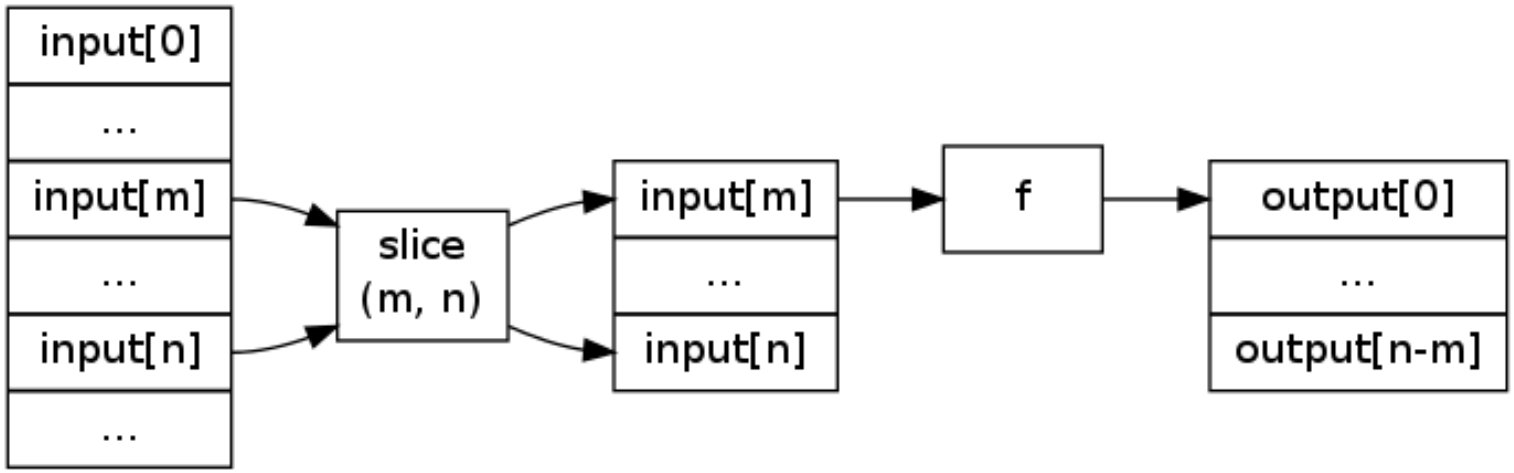


向量原语重排 VP reorder

向量原语重排旨在交换可以换序的 VP，以提高执行效率

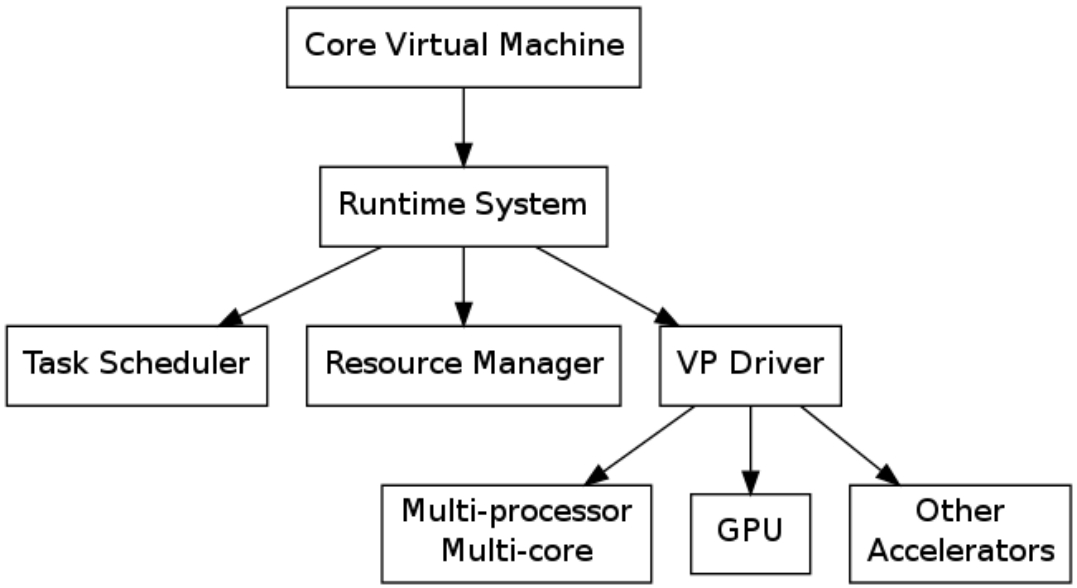
```
outputV = map f (map g inputV)
outputV = map (f . g) inputV
```

Figure: VP reorder 实例



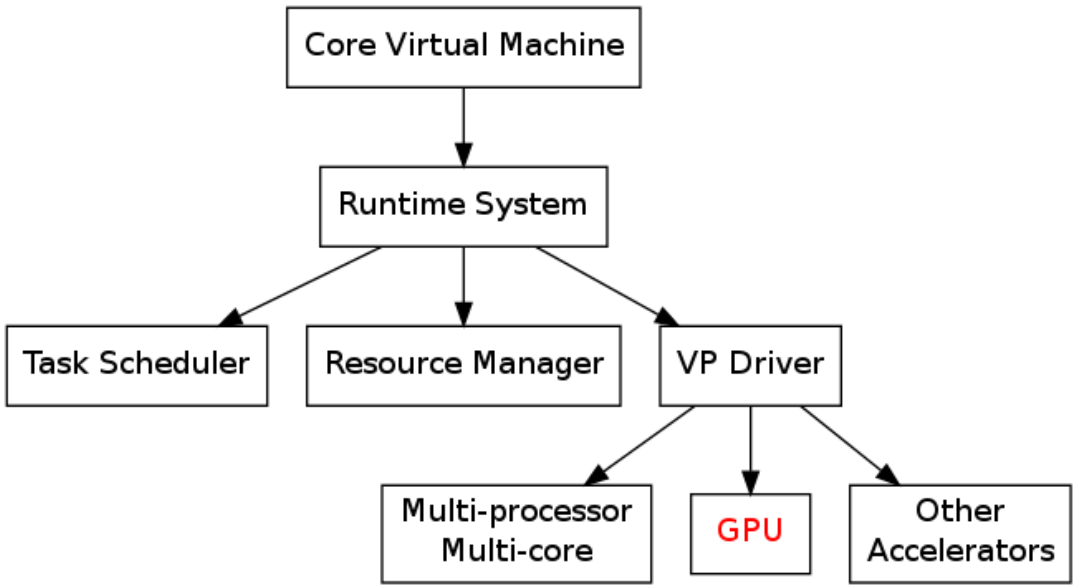
Rat 后端实现方案

Figure: Rat 后端结框图



Rat 后端实现方案

Figure: Rat 后端结框图



Runtime System

Task Scheduler

Task Scheduler 的功能：

- 任务的划分
- 子任务的调度与分派
- 结果的收集与合并
- 管理运行于不同设备任务之间的并行执行或流水化执行

Runtime System

Resource Manager

Resource Manager 的功能：

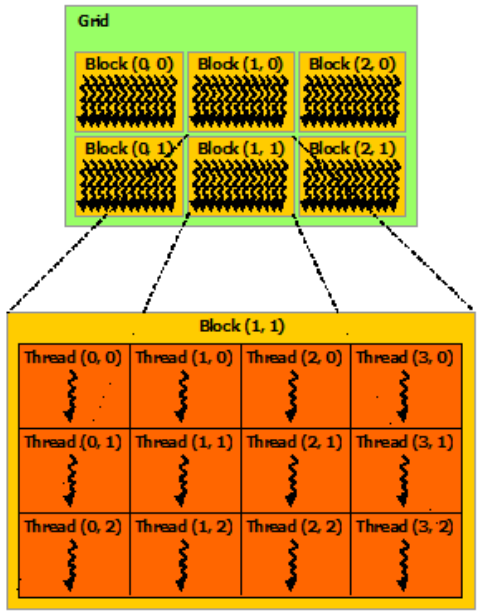
- 管理可用设备（CPU，GPU ...）
- 管理向量内存的分配与回收
- 对于标量内存执行采用垃圾回收机制

Runtime System

VP Driver

VP Driver 是在不同硬件上定义的抽象功能接口，提供向量原语的具体实现。

Figure: Nvidia GPU 线程模型



VP Driver 的 GPU 实现

对于 Rat 提供的向量原语，有的已经在 CUDPP[1] 中有高效实现。剩余的原语的实现过程也比较直观。

Table: VP 的 GPU 实现

map	ratMap	scale	ratCopy
slice	ratCopy	scan	cudappScan
compact	cudppCompact	sort	cudppSort
sparse	ratCopy	zip	
permute	ratCopy	random	cudppRand

[1] http://code.google.com/p/cudpp/

后端优化技术

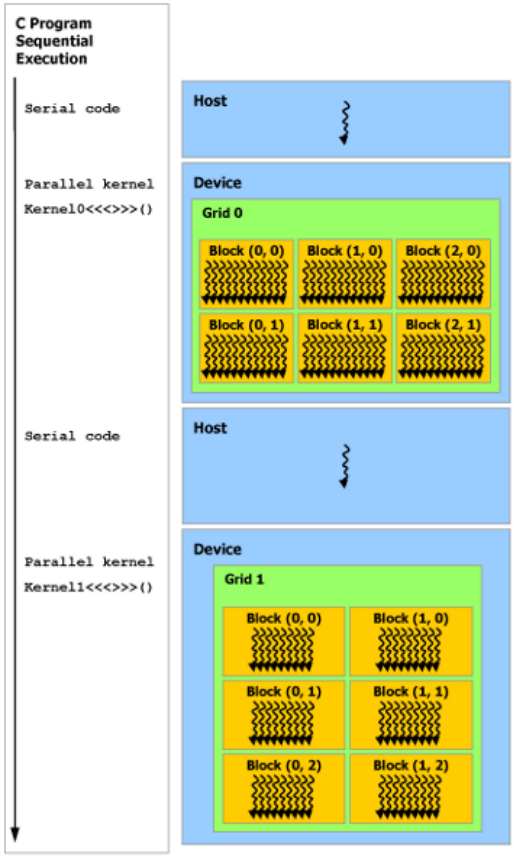
Rat 的后端实现平台为 Nvidia GPU，主要优化措施包括：

- 子程序并行执行
- GPU 访存操作优化

子程序并行执行

为了提高整个程序的并行度，在硬件允许的情况下，尽量使 CPU 与 GPU 同时工作。

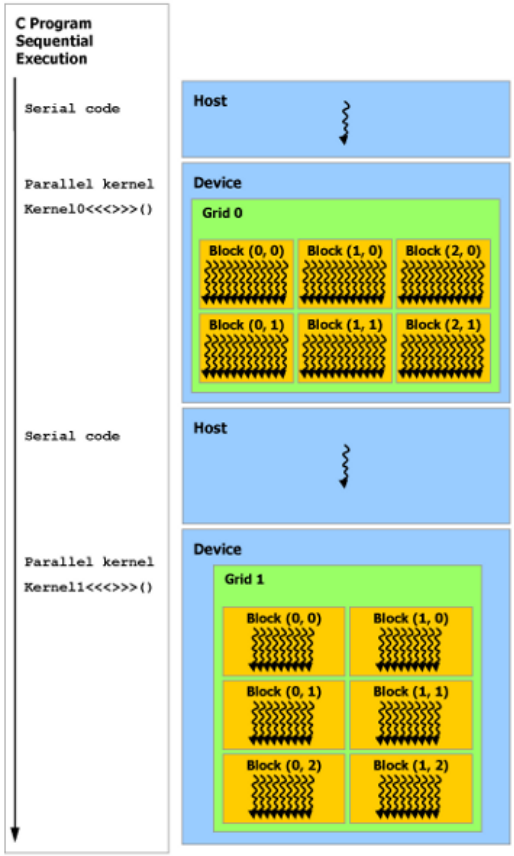
- GPU 端的 kernel 调用采用异步模式
- 重叠 Host-Device 之间的数据传输与 kernel 执行



子程序并行执行

为了提高整个程序的并行度，在硬件允许的情况下，尽量使 CPU 与 GPU 同时工作。

- GPU 端的 kernel 调用采用异步模式
- 重叠 Host-Device 之间的数据传输与 kernel 执行



“无副作用”：可以任意选取子程序并行执行，不影响程序正确性。

GPU 访存操作优化

要采用的访存优化措施主要包括：

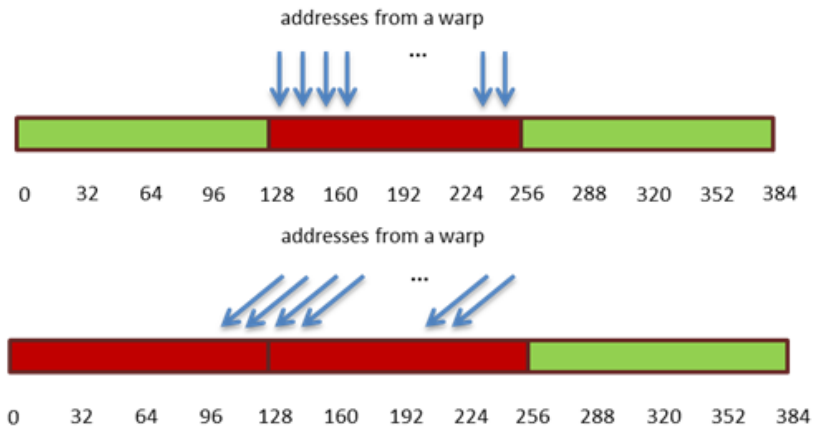
- Coalesced Access to Global Memory
- Utilizing Shared Memory

GPU 访存操作优化

Coalesced Access to Global Memory

GPU 上对于 global memory 的 Coalesced Access 可以在一次 L1 cache transection 内完成。

- 分配向量内存时对齐地址
- 分配 kernel 线程时匹配线程数目与数据块大小



GPU 访存操作优化

Utilizing Shared Memory

位于同一个 block 的线程共享一块片上存储器 shared memory，它的访问速度远远超过 global memory

- 对于对源操作数读取次数大于 1 的操作，先将数据传送到 shared memory 再执行计算
- 对于 fold 操作，先在 shared memory 中进行局部 reduce 的性能远远超过全局 reduce

Rat 实现情况

Rat 语言的实现正处于工作当中……

Table: Rat 语言各个组件实现情况

前端	Lexer	finished
	Parser	finished
	Type Checker	in progress
后端	Task Scheduler	in progress
	Resource Manager	finished
	VP Driver	finished

the end