Tutorial 中文版

Link of the original page:https://docs.flamegpu.com/tutorial/index.html

导论

本章提供了一份 FLAME GPU 2 的简介，并且以 Circles 模型为例说明如何在 FLAME GPU 2 中开发模型。

为了跟随这份导论，读者需要事先准备好一台带有满足 FLAME GPU C++ 或 Python 运行要求且适配 CUDA 的 GPU 的设备。

如果读者没有运行条件，导论在浏览器上提供了基于 Google Colab 的算例，只要电脑安装了现代浏览器并能接入网络就可以运行。不过，iPython notebooks 脚本性能也限制了部分特性的展示（比如可视化）。

FLAME GPU 的设计哲学

FLAME GPU 的目标是以GPU并行计算的方式加速多主体仿真的运行效率、扩大模型规模。框架的中心思想是，将 GPU 与建模者抽离，以便建模者可以在不考虑编写并行代码的情况下构建模型。FLAME GPU 还将模型描述与模型部署相分离。这简化了模型的验证过程，因为仿真器的代码已经隔离于模型完成了测试。

FLAME GPU 起始于应用 GPU 通用计算的早期阶段。GPU 硬件及软件在 FLAME GPU 开始后迎来了翻天覆地的变化，所以2.0版本是一个完全重写的库。它从一代的模板驱动 ABM 的结构转向了现代的 C++ API 和更清晰的智能体行为规范接口。它还添加了一系列新特性以确保仿真性能。比如：

- 对大 GPU 的支持——支持智能体函数的并发运行，确保了异构模型不会导致低设备利用率。
- 模型集成——运行集成模型的能力，同一模型可以以不同参数或随机种子运行。这在随机仿真中是必要的，并且 FLAME GPU 允许在单一计算节点上占用多个设备的集成规范。
- 子模型——FLAME GPU中的某些行为需要迭代过程，以确保与串行对应的可再现性(例如资源冲突解决)。FLAME GPU 2允许为这些行为描述可重用的子模型，这样它就可以从模型函数的其余部分中抽象出来。

创建项目

我们推荐使用提供的案例模板仓库来创建你自己的 FLAME GPU 2 模型。这些仓库为你提供了创建独立 FLAME GPU 2 模型的全部脚本。它们以 Circles 案例的部署作为开始，在余下的导论中我们将会清空它并从零开始。

- 如果你希望使用 CUDA/C++ 接口，请使用 FLAME GPU 2 example template。
- 如果你希望使用 Python 3.6+ 接口，请使用 FLAME GPU 2 python exmaple template。

一个 FLAME GPU 2 程序的结构

FLAME GPU 2 程序由四部分组成：

- 智能体/Host函数定义
- 模型声明
- 初始化
- 运行

智能体/Host 函数定义

这些函数定义了模型中的实际行为，通常定义在`main`函数之前，但是较大的模型可以使用更高级的技术来跨多个文件分割模型定义。关于智能体/Host函数定义的更多内容请参阅对应的完整指南。

模型声明

如果你使用 Python ，通常模型已经在`main`函数或主文件中声明了。这包括了一切对模型需要的智能体及消息类型的声明。我们推荐采用如下结构进行模型声明：

- 模型描述
- 消息描述
- 智能体描述
- 环境描述
- 函数执行顺序

初始化

模型的运行需要一个初始状态，这一般意味着需要设置一些初始的智能体和环境属性。这里有几种初始化的方式：

- 初始化函数：在仿真开始时运行一次的 host 函数。
- 输入文件：仿真可以在开始时从一个输入文件中加载智能体种群和环境参数。
- `AgentVector`，智能体种群和环境属性可以被外部地定义并用`CUDASimulation`在执行前设定，不过这个方法并不推荐使用。

执行

最后，为了执行你的模型，你必须将`ModelDescription`提供给`CUDASimulation`。在这一阶段你可以配置`simulation`及`CUDA`设定，也可以提供命令行参数。如果需要，你也可以为模型设置可视化功能。

万事俱备，调用`simulate()`来执行你的模型吧！

导论：建立 Circles 模型

在此之前，希望你已经下载并安装了其中一种案例模板。

Circles 模型简介

Circle 模型是一种简单的多主体模型，只包含了存在于二维或三维连续空间中的单种点状智能体。

智能体通过观察近邻的位置来决定自己如何移动。

模型旨在求解一种智能体形成圆形或球形聚簇的稳态。

下方的视频提供了 Circles 模型的展示。

配置 CMake

这一步仅仅在你使用 *C++* 或从源码构建 *pyflamegpu* 时是必要的。

FLAME GPU 2 使用 CMake 去管理构建进程，所以我们使用 CMake 生成一个由构建脚本组成的构建目录。它还可以通过下载某些缺失的依赖项来提供帮助。

需要用到的基本指令在 Linux 和 Windows 中有轻微不同，但是它们都必须在模板拷贝到的目录中执行。

关于从源代码构建FLAME GPU 2的更详细指南可以在这里找到。

```
# Create the build directory and change into it
mkdir -p build && cd build

# Configure CMake from the command line passing configure-time options.
cmake .. -DCMAKE_BUILD_TYPE=Release -DCMAKE_CUDA_ARCHITECTURES=61
```

> *-DCUDA_ARCH=61*表示构建适用于 *Pascal* 架构的 *SM_61 GPU*，你可能希望更改这一选项以适配你的设备。彻底忽略它将生成一个适用于所有当前架构的更大的二进制文件，这事实上将编译时间翻了架构数量的倍数。总之，采用较新而非特定架构的 *GPU* 也可以运行，但是会在某些程序为了较早架构编译的特性上受到限制。

项目的构建文件现在应当置于`build`目录下。

开始项目

Linux C++ 用户现在需要在他们常用的文本编辑器或 IDE 中打开一个新文件 `src/main.cu`。

Windows C++ 用户则使用 Visual Studio 打开新文件 `build/example.vcxproj`，接着在解决方案管理器中打开 `main.cu`。

Python 用户需要在常用编辑器或 IDE 中打开新文件 `model.py`。

在任何情况下，我们都会只保留 FLAME GPU include/import 语句。该语句允许文件访问完整的 FLAME GPU 2 库。

```
import pyflamegpu
```

模型描述

创建 FLAME GPU 模型的定义是去定义它，以创建`ModelDescription`作为开始。通过添加对消息、智能体和仿真环境的描述，它将用以描述完整的模型。

构造器 `ModelDescription` 使用的唯一参数是一个表示模型名称的字符串。名称只被用做创建可视化时的窗口标题。

一般 `ModelDescription` 被定义在程序流的开始。在 C++ 中这意味着写在`main()` 方法中，但是在 Python 中只是简单地被包含在主文件中（Python 确实允许指定一个输入函数）。

在模型描述之前，我们也将定义两个（常量）变量，以便定义环境的维数和智能体的数目。这些值将被用于一些地方，所以命名它们是有用的。

```
...
# Define some useful constants
AGENT_COUNT = 16384
ENV_WIDTH = int(AGENT_COUNT**(1/3))

# Define the FLAME GPU model
model = pyflamegpu.ModelDescription("Circles Tutorial")
...
```

消息描述

接着我们必须决定智能体如何交流传播。这通常在智能体函数返回值给消息之前完成，它们必须先被描述。

因为 Circles 模型中的智能体存在于一个连续空间并且希望找到它们在地的邻居，这里有三个潜在的消息类型适合模型：

- `MessageBruteForce`：每个智能体都能感知到每条消息，这在消息或智能体数目较大时会代价高昂。
- `MessageSpatial2D`：每个智能体输出信息到二维空间上的一个特定位置，智能体们只能感知到位置邻近于特定搜索起点的消息。
- `MessageSpatial3D`：每个智能体输出信息到三维空间上的一个特定位置，智能体们只能感知到位置邻近于特定搜索起点的消息。

在导论中，我们将实现二维的 Circles 模型，因此 `MessageSpatial2D` 是最恰当的消息类型。稍后将模型拓展至三维时需要一些小改动。

为了创建一个`MessageSpatial2D::Description`，`newMessage()`必须在先前创建的`ModelDescription`中调用。这是个模板函数，所以调用时必须使用带有需要的消息类型名称的模板，在我们的案例中是`MessageSpatial2D`。另外，唯一的参数是一个表示消息名称的字符串，这将在稍后将消息作为一个`AgentFunctionDescription`的输入或输出时使用。

> *Python* 界面不支持 *C++* 模板和嵌套类，所以命名风格上有所不同。在几乎所有情况，模板参数只是简单地缀于名称后面。

空间消息有一些在使用前必须指定的设置。

环境边框必须被`setMin()`和`setMax()`指定。空间消息可以被发送至任意一个位置，但为了最佳的性能，指定的边框应当把所有消息封入内部。出于这点考虑，我们将设置环境边框为`0`到前述步骤声明的`ENV_WIDTH`。

搜索半径也必须用`setRadius()`指定，这是从搜索起点到能被返回的消息的距离。这个半径被用于将覆盖的环境区域再分为离散网格，消息们会根据其网格位置被存储。导论中半径为`2`，稍后你可以在实验中修改这个值。

由于消息们被用于传播，你通常也会希望为它们添加变量。Circle 模型十分简单，位置由消息隐式地提供就已足够。然而，我们也会添加变量用于存储发送消息的智能体的 ID。这能用于确保智能体们不会处理它们自己的消息。添加变量使用`newVariable()`，这也是一个模板函数，其中模板参数是要用于变量的消息类型，唯一的常规参数是变量的名称。

> *FLAME GPU 2* 消息（和智能体）也许会拥有数组类型的变量。
>
> 在 *C++* 中，第二个模板参数被传递到*`newVariable()`*，例如 *`message.newVariable<int,3>("vector3");`*。
>
> 在 *Python* 中，第二个参数被传递到*`newVariableArray()`*中，如 *`message.newVariableArrayInt("vector3",3)`*。

FLAME GPU 提供了一种用于智能体 ID 的特殊类型，在 C++ 和 Python 中分别是`flamegpu::id_t`和`ID`。

```
...
# Define a message of type MessageSpatial2D named location
message = model.newMessageSpatial2D("location")
# Configure the message list
message.setMin(0, 0)
message.setMax(ENV_WIDTH, ENV_WIDTH)
message.setRadius(1)
# Add extra variables to the message
# X Y (Z) are implicit for spatial messages
message.newVariableID("id")
...
```

智能体描述

现在，是时候定义智能体了。在 FLAME GPU 中智能体是变量、智能体函数和可选状态的集合。由于 Circles 模型并非是状态性的，所以它们的用途在此不会涉及，但是你可以通过此处链接阅读智能体状态的相关内容。

为了定义新的`AgentDescription`类型，与新消息类型相似，`newAgent()`必须在先前创建的`ModelDescription`之前调用。唯一的参数是表示智能体名称的字符串，在稍后引用智能体类型时会用到（如，在 host 函数中）。对于 Circles 模型，我们仅将唯一的智能体类型命为`"point"`。

向智能体添加新变量与向消息添加变量十分相似，`newVariable()`被调用以提供变量类型、名称和可选的默认值。如果提供了默认值，它将被指派给新创建/诞生的智能体们。添加数组型变量和前一节遵循相同的规则，但是它们也需要指定初始值。

Circles 模型需要一个位置，这样我们可以添加三个`float`变量去表示它。另外，我们也会加入第四个`float`名为`"drift"`，这不是必须的，但可用于在无可视化时向我们提供一些可测量的东西。

```
...
message.newVariableID("id")

# Define an agent named point
agent = model.newAgent("point")
# Assign the agent some variables (ID is implicit to agents, so we don't
define it ourselves)
agent.newVariableFloat("x")
agent.newVariableFloat("y")
agent.newVariableFloat("z")
agent.newVariableFloat("drift", 0)
...
```

在设置智能体函数时，我们还将返回到这个代码块来。

环境描述

在 FLAME GPU 中，环境表示了智能体外部的状态。智能体对环境的属性具有只读访问权限，它们只能由主机函数更新。另外， FLAME GPU 2 添加了环境宏观属性以表示更大的环境数据，智能体对其的更新权限有限，这个高级特性不在导论中涉及，但可在此处探索。

在我们向环境添加属性前，我们还需要用`Environment()`从`ModelDescription`中取出`EnvironmentDescription`。

就像消息和智能体，`newProperty()`被用于向模型环境中添加属性。但是必须指定一个初始值作为第二参数。

Circles 模型只需要称为排斥的单一环境属性，这个`float`属性仅仅是调整模型中力（间接求解速度）的常数。最初设置为`0.05`。

另外，我们还会添加两个早前定义的常数，以让它们在模型中可用。

> *FLAME GPU 2* 允许环境属性作为`const`被标记，这防止了它们被意外更新。这是用于诸如数学常数的值。通过向`newProperty()`传递`true`（*C++*）或`True`（*Python*）作为第三参数可以启用这一功能。

```
...
message.newVariableID("id")

# Define an agent named point
agent = model.newAgent("point")
# Assign the agent some variables (ID is implicit to agents, so we don't
define it ourselves)
agent.newVariableFloat("x")
agent.newVariableFloat("y")
agent.newVariableFloat("z")
agent.newVariableFloat("drift", 0)
...
```

智能体函数描述实现

我们已为 Circles 模型定义了消息、智能体及环境，接下来就是实现智能体的行为并使用它们。

在 FLAME GPU 2 中，智能体函数使用 C++ `FLAMEGPU_AGENT_FUNCTION(name,input_message,out_message)`宏函数实现。它由编译器拓展，去生成一个智能体函数的全部定义（在 API 文档中查看拓展的案例）。不过，在我们的使用中只需要提供三个参数：函数名、函数的消息输入格式以及消息输出格式。然后函数就会以此来实现，宏调用被视作函数原型。

通过将函数指定为 C++ 字符串，可以在运行时编译智能体函数描述的 C++ 格式。这使得 Python 指定的模型可以动态编译。由于编译的特性，运行时编译为智能体函数的初始执行增加了少量额外成本。所幸，FLAME GPU 会缓存编译过的智能体函数为重复运行免除这部分成本（如果智能体函数/模型没有改变）。

使用 Python 指定智能体函数，既可以使用 C++ 格式，也可以通过导论所示的一种纯 Python 的描述实现（Python的一个子集，被称为Agent Python）。Python 中的智能体函数必须被定义为含有 `@pyflamegpu.agent_function` 装饰器及遵循含有函数名、消息输入输出类型指定的如下格式 `def outputdata(message_in: pyflamegpu.MessageNone, message_out: pyflamegpu.MessageNone):`。在编译之前， Python 实现会在运行时通过一个称为转译的过程将 Python 转换为 C++。

为了描述我们的行为，我们将从实现智能体函数开始，每个智能体输出一个消息，分享他们的位置。

我们将把这个函数命名为output_message（这个名字不应该用引号包裹），它没有消息输入，所以flamegpu::MessageNone(pyflamegpu.MessageNone in Agent Python)被用于输入消息参数，我们要输出我们上面定义的二维空间消息，所以flamegpu::MessageSpatial2D(pyflamegpu.MessageSpatial2D in Agent Python)被用于输出消息参数。

在这之后，我们可以实现智能体函数体。智能体函数提供了一个单一的输入参数，FLAMEGPU，这是一个指向DeviceAPI的指针，这个对象在智能体函数中提供了对所有可用的FLAME GPU特性（智能体变量、消息输入/输出、环境属性、智能体输出、随机）的访问。

为了实现输出消息的智能体函数，我们需要读取智能体的位置（"x"，"y"）变量和ID，然后设置消息的位置和"id "变量。

为了读取智能体的变量，在C++中使用了FLAMEGPU->getVariable（）函数。正如你现在所期望的，变量的类型必须作为一个模板参数传递，而它的名字是唯一的参数。要读取一个智能体的ID，需要调用FLAMEGPU->getID()，这个特殊的函数不需要其他参数。Python的实现使用相同的格式，将类型附加到函数名称上。这些函数可以通过pyflamegpu模块访问。例如，pyflamegpu.getVariableInt()表示一个int类型。

消息输出的功能通过FLAMEGPU->message_out（或者在Agent Python中命名为message_out的变量）来访问，这个对象根据最初在FLAMEGPU_AGENT_FUNCTION宏中（或者通过Python类型注解）指定的输出消息类型进行指定。二维空间中，flamegpu::MessageSpatial2D::Out，有两个可用的函数；setVariable()是所有消息输出类型所共有的，而setLocation()需要两个浮点参数，指定消息在二维空间的位置。Python的对应函数与其他地方的格式相同（例如，setVariableInt用于int类型）。

最后，所有的智能体函数必须返回flamegpu::ALIVE或flamegpu::DEAD（在Agent Python中分别为pyflamegpu.ALIVE或pyflamegpu.DEAD）。除非智能体函数在AgentFunctionDescription中通过setAllowAgentDeath()指定支持智能体死亡，否则应该返回flamegpu::ALIVE。如果flamegpu::DEAD被返回，而没有启用智能体死亡，如果SEATBELTS错误检查被启用，将产生一个异常。

下面你可以看到消息输出函数可能被组装起来。通常情况下，智能体函数会在源文件的顶部附近实现，直接放在任何包的导入操作之后。

```
...
# Agent Function to output the agents ID and position in to a 2D spatial
message list
@pyflamegpu.agent_function
def output_message(message_in: pyflamegpu.MessageNone, message_out:
pyflamegpu.MessageSpatial2D):
    message_out.setVariableUInt("id", pyflamegpu.getID())
    message_out.setLocation(
        pyflamegpu.getVariableFloat("x"),
        pyflamegpu.getVariableFloat("y"))
    return pyflamegpu.ALIVE
...
```

接下来实现消息输入智能体函数，这里引入了两个新概念：消息输入迭代器和访问环境属性。

每个FLAME GPU的消息类型都提供了访问消息的独特方法，在导论中，我们使用的是`MessageSpatial2D`类型。关于其他消息格式的使用细节，请参考智能体通信指南。

访问空间消息类型的唯一方法是通过一个迭代器，它返回关于所提供的搜索位置的摩尔邻域（由消息半径离散构造）的所有消息。这意味着，最初指定的搜索半径内的所有消息都将被返回，然而，用户有必要过滤掉那些包含在摩尔邻域内但不在此半径内的消息。此外，智能体也会收到他们自己的消息，所以不妨通过检查信源智能体的ID来过滤消息。

空间消息迭代器是通过`FLAMEGPU->message_in()`访问的（或者通过Agent Python中的`message_in`智能体函数参数），这需要两个浮动参数，指定搜索原点。通常情况下，这将被直接传递给一个基于C++范围的for循环，允许返回的消息被迭代。

在`MessageSpatial2D`的情况下，返回的`Message`对象只提供`getVariable()`方法来返回存储在消息中的变量和数组变量。与之对应的Python要求将类型和数组长度附加到函数名中（例如`getVariableIntArray3(...)`）。

访问环境属性与访问智能体和消息变量非常相似，`getProperty()`被调用到`FLAMEGPU->environment`。相当于Python要求将类型和数组长度附加到函数名称上（例如`getVariableIntArray3(...)`）。

Circles模型的消息输入智能体函数的其余部分包含一些模型特定的数学，所以你应该简单地使用下面提供的代码。不过，请仔细阅读以检查你是否理解了消息是如何被读取的。

```
...
# Agent Function to read the location messages and decide how the agent
should move
@pyflamegpu.agent_function
def input_message(message_in: pyflamegpu.MessageSpatial2D, message_out:
pyflamegpu.MessageNone):
    ID = pyflamegpu.getID()
    REPULSE_FACTOR = pyflamegpu.environment.getPropertyFloat("repulse")
    RADIUS = message_in.radius()
```

```
    fx = 0.0
    fy = 0.0
    x1 = pyflamegpu.getVariableFloat("x")
    y1 = pyflamegpu.getVariableFloat("y")
    count = 0
    for message in message_in(x1, y1) :
        if message.getVariableUInt("id") != ID :
            x2 = message.getVariableFloat("x")
            y2 = message.getVariableFloat("y")
            x21 = x2 - x1
            y21 = y2 - y1
            separation = math.sqrtf(x21*x21 + y21*y21)
            if separation < RADIUS and separation > 0 :
                k = math.sinf((separation /
RADIUS)*3.141*-2)*REPULSE_FACTOR
                # Normalise without recalculating separation
                x21 /= separation
                y21 /= separation
                fx += k * x21
                fy += k * y21
                count += 1
    fx /= count if count > 0 else 1
    fy /= count if count > 0 else 1
    pyflamegpu.setVariableFloat("x", x1 + fx)
    pyflamegpu.setVariableFloat("y", y1 + fy)
    pyflamegpu.setVariableFloat("drift", math.sqrtf(fx*fx + fy*fy))
    return pyflamegpu.ALIVE
...
```

现在，这两个智能体函数已经实现，它们必须被附加到模型上。

回到先前定义的智能体，首先我们用它来为我们使用`newFunction()`（C++ API）或`newRTCFunction()`（Python或C++ Agent API）定义的两个函数中的每一个创建`AgentFunctionDescription`。这两个函数都需要两个参数，首先是一个指代函数的名称，其次是上面定义的函数实现。

如果智能体函数是用Python语言指定的，那么它将需要使用`pyflamegpu.codegen.translate()`函数进行翻译。然后，产生的C++智能体代码可以被传递给`newRTCFunction()`。

返回的`AgentFunctionDescription`可以用来配置智能体功能，使其支持智能体的出生和死亡以及任何使用的消息输入或输出。由于我们使用的是消息，我们必须调用`setMessageOutput()`和`setMessageInput()`，传递给我们的消息类型的名称`("location")`。

```
#ensure to import the codegen module (usually at the top of your Python
file)
```

```
import pyflamegpu.codegen
...
agent.newVariableFloat("drift", 0)
# translate the agent functions from Python to C++
output_func_translated = pyflamegpu.codegen.translate(output_message)
input_func_translated = pyflamegpu.codegen.translate(input_message)
# Setup the two agent functions
out_fn = agent.newRTCFunction("output_message", output_func_translated)
out_fn.setMessageOutput("location")
in_fn = agent.newRTCFunction("input_message", input_func_translated)
in_fn.setMessageInput("location")

...
```

执行顺序

最后，模型的执行流程必须被设置。这可以通过使用旧的FLAME GPU 1风格的层来实现（见`ModelDescription::newLayer()`），或者使用新的依赖图API。在本导论中，我们将使用依赖API。

为了定义函数在模型中的执行顺序，必须指定它们的依赖关系。`AgentFunctionDescription`、`HostFunctionDescription`和`SubModelDescription`对象都实现了`dependsOn()`。这被用来指定模型的函数之间的依赖关系。

用`ModelDescription::addRoot()`指定图的根，最后通过`ModelDescription::generateLayers()`将依赖图转换为层。

这可以放在文件的末尾，跟随之前定义的环境属性。

```
...
# Message input depends on output
in_fn.dependsOn(out_fn)
# Dependency specification
# Output is the root of our graph
model.addExecutionRoot(out_fn)
model.generateLayers()
...
```

初始化函数

现在，模型的组件和行为已经设置完毕，是时候决定如何初始化模型了。FLAME GPU允许模型通过输入文件和/或用户定义的初始化函数来初始化，这可能取决于环境属性或从输入文件加载的智能体。

对于Circles模型，我们只需要在环境范围内随机散布一定数量的智能体。因此，我们可以简单地根据我们前面定义的一些环境属性来生成智能体。

与代理函数类似，C++的API使用FLAMEGPU_INIT_FUNCTION来定义初始化函数，它需要一个函数名称的单一参数。相比之下，Python有本地函数，所以它们的定义是不同的，必须创建一个pyflamegpu.HostFunction的子类，它实现的方法是def run(self, FLAMEGPU):。

初始化函数可以访问HostAPI，它是智能体函数中的DeviceAPI的主机（CPU）对应部分。它有类似的功能，还有一些额外的功能：智能体变量的归约、设置环境属性。

首先，我们需要生成一些随机数以决定位置。HostAPI包含提供对随机功能访问的HostRandom。这提供了uniform()。它只需要一个float模板参数，并将返回一个包含或排除范围[0, 1]的随机数。

我们唯一需要使用HostAPI特有的功能是智能体的诞生，在主机上可以创建任何数量的智能体而不受智能体函数的限制。首先我们获取"point"智能体的HostAgentAPI，这使我们能够访问影响该智能体的功能。然后，我们可以简单地调用newAgent()来创建新的智能体，返回的智能体具有正常的setVariable()功能，在初始化函数全部完成后将被添加到仿真中。

同样，初始化函数，在文件的顶部附近，与智能体函数并列。

把所有这些放在一起，我们可以使用下面的代码来生成初始智能体群体。

```
...
class create_agents(pyflamegpu.HostFunction):
    def run(self, FLAMEGPU):
        # Fetch the desired agent count and environment width
        AGENT_COUNT =
FLAMEGPU.environment.getPropertyUInt("AGENT_COUNT")
        ENV_WIDTH = FLAMEGPU.environment.getPropertyFloat("ENV_WIDTH")
        # Create agents
        t_pop = FLAMEGPU.agent("point")
        for i in range(AGENT_COUNT):
            t = t_pop.newAgent()
            t.setVariableFloat("x", FLAMEGPU.random.uniformFloat() *
ENV_WIDTH)
            t.setVariableFloat("y", FLAMEGPU.random.uniformFloat() *
ENV_WIDTH)
...
```

> 在初始化函数中使用*FLAME GPU*随机*API*，确保随机（以及模型）是根据执行时为仿真指定的随机种子。

与智能体函数类似，初始化函数必须被附加到模型上。初始化函数总是在模型开始时运行一次，所以没有必要使用层或依赖图，它们只是使用addInitFunction()（C++ API）或addInitFunction()（Python API）添加到ModelDescription中。

```
...
dependencyGraph.generateLayers(model)
model.addInitFunction(create_agents())
...
```

配置仿真

现在ModelDescription已经完成，所以是时候构建一个CUDASimulation来执行这个模型了。

在大多数情况下，这只是构建CUDASimulation，用命令行参数初始化它并调用simulate()。也可以在代码中设置这种配置，详情见用户指南。

```
...
# Import sys for access to run args (this can be moved to the top of
your Python file)
import sys

# Create and run the simulation
cuda_model = pyflamegpu.CUDASimulation(model)
cuda_model.initialise(sys.argv)
cuda_model.simulate()
```

你可以选择通过CUDASimulation配置日志或可视化，这些将在下面两节解释。

配置日志（可选）

在无可视化运行FLAME GPU模型时，你很可能想从运行中收集数据。这可以通过定义一个日志配置来实现。

在本教程中，我们将记录每一步"point"智能体的"drift"变量的平均值，如果模型工作正常，当智能体达到稳定状态时，这个值应趋于零。

为了实现这一点，我们必须首先创建一个StepLoggingConfig，将我们完成的ModelDescription传递给它的构造函数。

这个对象为记录智能体数据和环境属性提供了广泛的选项。我们只需要使用agent()请求AgentLoggingConfig。之后，我们只需调用logMean()，提供智能体变量的类型作为模板参数，它的名字作为唯一参数。

在StepLoggingConfig被完全定义后，可以使用setStepLog()将其附加到CUDASimulation中。

```
... # following on from model.addInitFunction(create_agents())
```

```
# Specify the desired StepLoggingConfig
step_log_cfg = pyflamegpu.StepLoggingConfig(model)
# Log every step
step_log_cfg.setFrequency(1)
# Include the mean of the "point" agent population's variable 'drift'
step_log_cfg.agent("point").logMeanFloat("drift")

# Create the simulation
cuda_model = pyflamegpu.CUDASimulation(model)

# Attach the logging config
cuda_model.setStepLog(step_log_cfg)

# Init and run the simulation
cuda_model.initialise(sys.argv)
cuda_model.simulate()
```

仿真完成后，可以用`getRunLog()`收集日志，如果在执行前配置了适当的输出文件，则可以将日志写入文件。

要了解更多关于使用日志配置的信息，请看用户指南。

配置可视化（可选）

通过使用可视化，许多模型在早期更容易快速验证，FLAME GPU提供了一个可视化器，能够根据其变量显示出智能体的位置、方向、比例和颜色。

可视化配置（`ModelVis`）是由`CUDASimulation`使用`getVisualisation()`创建的。这提供了许多配置可视化的高级选项，请参阅用户指南的完整概述，我们将在这里介绍Circles模型可视化的最低要求。

下面的代码定位初始摄像机，设置摄像机的移动速度（当用户使用键盘移动时），将`"point"`智能体渲染成冰球（这些是低多边形数量的球体，非常适合智能体数量庞大的可视化），并用白色的方块标记出环境的边界。

此外，仿真速度被限制为每秒25步。这使得仿真的演变可以更清楚地被可视化。这个小模型通常会以每秒数百步的速度执行，但是达到稳定状态的速度太快，无法观察。

重要的是在可视化配置完成后调用`activate()`，以最终确定并启动可视化器。

在大多数情况下，你会希望可视化在仿真完成后持续存在，这样可以探索终止状态。为了达到这个目的，必须在`simulate()`之后调用`join()`，以便在主程序线程终止之前抓住它。

> *FLAME GPU*被设计用于个人机器和通过*ssh*访问的服务器（如超算）。后者不可能有对可视化的支持，所以*FLAME GPU*也可以在不支持可视化的情况下构建。因此，用*`VISUALISATION`*宏的检查来包装可视化的具体代码是很有用的，允许模型在不考虑可视化支持的情况下编译/运行，而不是保持两个版本。

```
... # following on from cuda_model = pyflamegpu.CUDASimulation(model)

# Only run this block if pyflamegpu was built with visualisation support
if pyflamegpu.VISUALISATION:
    # Create visualisation
    m_vis = cuda_model.getVisualisation()
    # Set the initial camera location and speed
    INIT_CAM = ENV_WIDTH / 2
    m_vis.setInitialCameraTarget(INIT_CAM, INIT_CAM, 0)
    m_vis.setInitialCameraLocation(INIT_CAM, INIT_CAM, ENV_WIDTH)
    m_vis.setCameraSpeed(0.01)
    m_vis.setSimulationSpeed(25)
    # Add "point" agents to the visualisation
    point_agt = m_vis.addAgent("point")
    # Location variables have names "x" and "y" so will be used by
default
    point_agt.setModel(pyflamegpu.ICOSPHERE);
    point_agt.setModelScale(1/10.0);
    # Mark the environment bounds
    pen = m_vis.newPolylineSketch(1, 1, 1, 0.2)
    pen.addVertex(0, 0, 0)
    pen.addVertex(0, ENV_WIDTH, 0)
    pen.addVertex(ENV_WIDTH, ENV_WIDTH, 0)
    pen.addVertex(ENV_WIDTH, 0, 0)
    pen.addVertex(0, 0, 0)
    # Open the visualiser window
    m_vis.activate()

# Init and run the simulation
cuda_model.initialise(sys.argv)
cuda_model.simulate()

if pyflamegpu.VISUALISATION:
    # Keep the visualisation window active after the simulation has
completed
    m_vis.join()
```

运行仿真

此时，你应该有一个完整的模型，可以被（编译和）运行。

如果要在随机种子为**12**的情况下运行**500**步的模型，你需要传递运行时参数`-s 500 -r 12`。

如果你选择添加一个日志配置，你将需要额外指定一个日志文件，例如 `--out-step step.json`。

如果你已经包括了可视化，但是希望阻止它的运行，你应加入`--console`或`-c`。

如果你想继续学习Circles模型，请尝试这些扩展。

- 扩展模型，使其在三维环境中运行。
- 将模型扩展到在环绕的二维环境（环形）中运行。
- 扩展可视化，根据智能体的"drift"变量或读取的消息数量给其着色。
- 扩展模型，给智能体一个重量，影响他们对其他智能体施加/接收的力。

完整代码

如果已经跟随了全部的导论，那么你应该获得如下的代码

```
from pyflamegpu import *
import pyflamegpu.codegen
import sys

# Define some useful constants
AGENT_COUNT = 16384
ENV_WIDTH = int(AGENT_COUNT**(1/3))

# Define the FLAME GPU model
model = pyflamegpu.ModelDescription("Circles Tutorial")

# Define a message of type MessageSpatial2D named location
message = model.newMessageSpatial2D("location")
# Configure the message list
message.setMin(0, 0)
message.setMax(ENV_WIDTH, ENV_WIDTH)
message.setRadius(1)
# Add extra variables to the message
# X Y (Z) are implicit for spatial messages
message.newVariableID("id")

# Define an agent named point
agent = model.newAgent("point")
# Assign the agent some variables (ID is implicit to agents, so we don't
define it ourselves)
agent.newVariableFloat("x")
agent.newVariableFloat("y")
agent.newVariableFloat("z")
agent.newVariableFloat("drift", 0)

# Define environment properties
env = model.Environment()
env.newPropertyUInt("AGENT_COUNT", AGENT_COUNT)
env.newPropertyFloat("ENV_WIDTH", ENV_WIDTH)
env.newPropertyFloat("repulse", 0.05)

@pyflamegpu.agent_function
def output_message(message_in: pyflamegpu.MessageNone, message_out:
pyflamegpu.MessageSpatial2D):
```

```
    message_out.setVariableUInt("id", pyflamegpu.getID())
    message_out.setLocation(
        pyflamegpu.getVariableFloat("x"),
        pyflamegpu.getVariableFloat("y"))
    return pyflamegpu.ALIVE

@pyflamegpu.agent_function
def input_message(message_in: pyflamegpu.MessageSpatial2D, message_out:
pyflamegpu.MessageNone):
    ID = pyflamegpu.getID()
    REPULSE_FACTOR = pyflamegpu.environment.getPropertyFloat("repulse")
    RADIUS = message_in.radius()
    fx = 0.0
    fy = 0.0
    x1 = pyflamegpu.getVariableFloat("x")
    y1 = pyflamegpu.getVariableFloat("y")
    count = 0
    for message in message_in(x1, y1):
        if message.getVariableUInt("id") != ID :
            x2 = message.getVariableFloat("x")
            y2 = message.getVariableFloat("y")
            x21 = x2 - x1
            y21 = y2 - y1
            separation = math.sqrtf(x21*x21 + y21*y21)
            if separation < RADIUS and separation > 0 :
                k = math.sinf((separation /
RADIUS)*3.141*-2)*REPULSE_FACTOR
                # Normalise without recalculating separation
                x21 /= separation
                y21 /= separation
                fx += k * x21
                fy += k * y21
                count += 1
    fx /= count if count > 0 else 1
    fy /= count if count > 0 else 1
    pyflamegpu.setVariableFloat("x", x1 + fx)
    pyflamegpu.setVariableFloat("y", y1 + fy)
    pyflamegpu.setVariableFloat("drift", math.sqrtf(fx*fx + fy*fy))
    return pyflamegpu.ALIVE

# translate the agent functions from Python to C++
output_func_translated = pyflamegpu.codegen.translate(output_message)
input_func_translated = pyflamegpu.codegen.translate(input_message)
# Setup the two agent functions
out_fn = agent.newRTCFunction("output_message", output_func_translated)
out_fn.setMessageOutput("location")
in_fn = agent.newRTCFunction("input_message", input_func_translated)
in_fn.setMessageInput("location")
```

```
# Message input depends on output
in_fn.dependsOn(out_fn)
# Dependency specification
# Output is the root of our graph
model.addRoot(out_fn)
model.generateLayers()

class create_agents(pyflamegpu.HostFunction):
    def run(self, FLAMEGPU):
        # Fetch the desired agent count and environment width
        AGENT_COUNT =
FLAMEGPU.environment.getPropertyUInt("AGENT_COUNT")
        ENV_WIDTH = FLAMEGPU.environment.getPropertyFloat("ENV_WIDTH")
        # Create agents
        t_pop = FLAMEGPU.agent("point")
        for i in range(AGENT_COUNT):
            t = t_pop.newAgent()
            t.setVariableFloat("x", FLAMEGPU.random.uniformFloat() *
ENV_WIDTH)
            t.setVariableFloat("y", FLAMEGPU.random.uniformFloat() *
ENV_WIDTH)

model.addInitFunction(create_agents())

# Specify the desired StepLoggingConfig
step_log_cfg = pyflamegpu.StepLoggingConfig(model)
# Log every step
step_log_cfg.setFrequency(1)
# Include the mean of the "point" agent population's variable 'drift'
step_log_cfg.agent("point").logMeanFloat("drift")

# Create the simulation
cuda_model = pyflamegpu.CUDASimulation(model)

# Attach the logging config
cuda_model.setStepLog(step_log_cfg)

# Init and run the simulation
cuda_model.initialise(sys.argv)
cuda_model.simulate()

# Create and run the simulation
cuda_model = pyflamegpu.CUDASimulation(model)

# Only run this block if pyflamegpu was built with visualisation support
if pyflamegpu.VISUALISATION:
    # Create visualisation
    m_vis = cuda_model.getVisualisation()
    # Set the initial camera location and speed
```

```
    INIT_CAM = ENV_WIDTH / 2
    m_vis.setInitialCameraTarget(INIT_CAM, INIT_CAM, 0)
    m_vis.setInitialCameraLocation(INIT_CAM, INIT_CAM, ENV_WIDTH)
    m_vis.setCameraSpeed(0.01)
    m_vis.setSimulationSpeed(25)
    # Add "point" agents to the visualisation
    point_agt = m_vis.addAgent("point")
    # Location variables have names "x" and "y" so will be used by
default
    point_agt.setModel(pyflamegpu.ICOSPHERE);
    point_agt.setModelScale(1/10.0);
    # Mark the environment bounds
    pen = m_vis.newPolylineSketch(1, 1, 1, 0.2)
    pen.addVertex(0, 0, 0)
    pen.addVertex(0, ENV_WIDTH, 0)
    pen.addVertex(ENV_WIDTH, ENV_WIDTH, 0)
    pen.addVertex(ENV_WIDTH, 0, 0)
    pen.addVertex(0, 0, 0)
    # Open the visualiser window
    m_vis.activate()

# Init and run the simulation
cuda_model.initialise(sys.argv)
cuda_model.simulate()

if pyflamegpu.VISUALISATION:
    # Keep the visualisation window active after the simulation has
completed
    m_vis.join()
```

相关链接

- User Guide Page: What is SEATBELTS?