



Denglin HammingTM V2

dINNE Developer Guide

DL-DG/SW-031B-05

2024-12-27

Copyright©苏州登临科技有限公司，2019 - 2025，版权所有。

未经苏州登临科技有限公司事先书面同意，不得以任何形式或方式复制或传播本文件的任何部分。

商标和许可



和其它苏州登临科技有限公司的其它登临科技的图标为苏州登临科技有限公司的商标。本手册中提及的所有其他商标均为其各自所有者的财产。

通知

所购买的产品、服务和特性由苏州登临科技有限公司与客户签订的合同规定。本文件中描述的所有或部分产品、服务和特性可能不在采购范围或使用范围内。除非合同中另有规定，本文件中的所有声明、信息和建议均按“原样”提供，无任何明示或暗示的保证或陈述。本手册中的信息如有更改，恕不另行通知。本文件在编制过程中已尽一切努力确保内容的准确性，本文件中的所有声明、信息和建议不构成任何明示或暗示的保证。

苏州登临科技有限公司

苏州工业园区扬富路11号南岸新地一期商务楼栋5号1101室，江苏，中国

<http://www.denglin.ai>

email : support@denglin.ai

Change History

Version	Change description
05	Revise chapter 7 Network Configuration .
04	Revise TC device profile.
03	Revise chapter 5.1 Weight Share Mode and Cluster Option . Add chapter 7 Network Configuration . Add API <code>SetConstMemoryEXT</code> , <code>UploadConstEXT</code> , <code>GetConstInfoEXT</code> and <code>SerializeWithoutConstEXT</code> . Delete API <code>SetConstMemory</code> and <code>UploadConst</code> .
02	Add note in chapter 3.3 Serializing a Model .
01	Initial version.

Table of Contents

Table of Contents

1 Abstract

2 What is dINNE?

2.1 How does dINNE Work?

2.2 What does dINNE Provide ?

3 Using C++ API

3.1 Creating a Network

3.2 Building an Engine

3.3 Serializing a Model

3.4 Performing Inference

4 Extending dINNE with Custom Layers

5 Configuration

5.1 Weight Share Mode and Cluster Option

5.2 Builder Flag Option

5.3 Dump Dot Option

5.4 Dump Relay IR Option

5.5 Modulator callback Option

5.6 Print Profile Option

5.7 User Const Option

6 dINNE Build Modulator

7 Network Configuration

1 Abstract

This document demonstrates how to use the C++ API for performing a trained neural network on a DengLin Goldwasser II™ AI accelerator card. The guide provides step-by-step instructions for common user tasks such as, creating a dINNE network, invoking the dINNE builder, serializing and deserializing, and how to feed the engine with data and do inference.

登临科技保密材料

2 What is dINNE?

dINNE is a C++ library that can optimize high-performance neural network inference and a runtime engine for production deployment.

2.1 How does dINNE Work?

To optimize your model for inference, dINNE takes your network parse, performs optimizations including platform-specific optimizations, and generates the inference engine. This process is referred to as the build phase. The build phase can take considerable time, especially when running on other platforms. Therefore, a typical application will build an engine once, and then serialize it as a plan file for later use.

The build phase performs the following optimizations on the graph:

- Elimination of layers whose outputs are not used
- Elimination of operations which are equivalent to no-op
- The fusion of TU pattern (convolution, bias, activation, pooling...)
- The fusion of CU operations except TU
- Merging of concatenation layers by directing layer outputs to the correct eventual destination
- Folding constant layers

2.2 What does dINNE Provide ?

dINNE enables developers to import, generate and deploy optimized networks. Network can be imported directly from TensorFlow or other frameworks via ONNX or Caffe formats. User can also run custom layers through dINNE using the Plugin interface. dINNE provides a C++ implementation and a python implementation. The key interfaces in the dINNE core library are:

Network

- The network for the parser to parse the network model.

Builder

- The Builder interface allows the creation of an optimized engine from a network and a builder configuration.

Engine

- The Engine interface allows the application to execute inference. It supports synchronous and asynchronous execution, and enumeration and querying of the bindings for the engine inputs and outputs. A single engine can have multiple execution contexts, allowing a single set of trained parameters to be used for the simultaneous execution of multiple batches.

Parser

- This parser can be used to parse a TensorFlow network serialized pb file. It also provides the ability to register a plugin creator for custom op.

3 Using C++ API

The following sections highlight the dINNE user goals and tasks that you can perform using the C++ API. Further details are provided in samples in Hamming™ SDK and document **dINNE-API.pdf**.

3.1 Creating a Network

The first step in performing inference with dINNE is to create a dINNE network from your model. And import the model using the dINNE Parser which supports TensorFlow, ONNX, and Caffe formats.

1. Create the builder object and the network object:

```
Builder* builder = CreateInferBuilder();
Network* network = builder->CreateNetwork();
```

2. Create the TensorFlow parser object:

```
Parser* parser = CreateParser();
```

3. Declare the network input tensors' information including name, shape, format and output tensors' names to the parser:

```
parser->RegisterInput("Input_0", Dims(1, 224, 224, 3), kNHWC);
parser->RegisterOutput("Output_0");
```

4. Parse the TensorFlow pb model file to populate the network:

```
parser->Parse(tf_pb_file, *network);
```

5. You can also parse from the mem to populate the network:

```
enum ModelFormat {
    kOnnxModel,
    kTensorflowModel,
    kCaffeModel,
};

std::vector<char*> buffers;
std::vector<uint64_t> sizes;
parser->ParseBuffers(ModelFormat::kOnnxModel, (void**)buffers.data(), sizes.data(), 1,
    *network);
```

3.2 Building an Engine

The next step is to invoke the dINNE builder to create an optimized runtime executable.

1. Build the engine object by using the builder object:

```
Engine* engine = builder->BuildEngine();
```

2. Dispense with the network, builder, and parser if using one.

```
parser->Destroy();
network->Destroy();
builder->Destroy();
```

3.3 Serializing a Model

It is not absolutely necessary to serialize and deserialize a model before using it for inference - if desirable, the engine object can be used for inference directly.

To serialize, you are transforming the engine into a format to store and use at a later time for inference. To use for inference, you would simply deserialize the engine. Serializing and deserializing are optional. Since creating an engine from the Network can be time consuming, you could avoid rebuilding the engine every time the application reruns by serializing it once and deserializing it while running inference. Therefore, after the engine is built, users typically want to serialize it for later use.

Note:

You can only serialize engine before create any context.

1. Run the builder as a prior offline step and then serialize:

```
HostMemory *serializedModel = engine->Serialize();
// store model to disk
// ...
serializedModel->Destroy();
```

2. Call `Deserialize`:

```
Engine* engine = Deserialize(modelData, modelSize);
```

The final argument is a plugin creator array for applications using custom layers.

3.4 Performing Inference

The following steps illustrate how to perform inference in C++ since you have an engine.

1. Create some space to store intermediate activation values. Since the engine holds the network and trained parameters, additional space is necessary. These are held in an execution context:

```
ExecutionContext* context = engine->CreateExecutionContext();
```

An engine can have multiple execution contexts, allowing one set of weights to be used for multiple overlapping inference tasks. For example, you can process images in parallel CUDA streams using one engine and one context per stream.

2. Use the input and output blob names to get the corresponding input and output index:


```
int inputIndex = engine->GetBindingIndex(INPUT_BLOB_NAME);  
int outputIndex = engine->GetBindingIndex(OUTPUT_BLOB_NAME);
```

3. Using these indices, set up a buffer array pointing to the input and output buffers:

```
void* buffers[2];  
buffers[inputIndex] = inputBuffer;  
buffers[outputIndex] = outputBuffer;
```

4. dINNE execution is in asynchronous or synchronous mode.

```
// sync mode  
context->Execute(batchSize, buffers);  
// async mode  
context->Enqueue(batchSize, buffers, stream, nullptr);
```

It is common to enqueue asynchronous `memcpy()` before and after the kernels move data from GPU if it is not already there. The final argument to `enqueue()` is an optional CUDA event which will be signaled when the input buffers have been consumed and their memory may be safely reused.

4 Extending dINNE with Custom Layers

Beyond the TU and CU patterns, dINNE also supports customer defined layers, referred to as plugins, implemented and instantiated by an application, while their lifetimes must span their use within a dINNE engine.

The following steps illustrate how to use custom layers.

1. Construct an op in TensorFlow. For details, follow the steps in guide of custom_op from TensorFlow. The following example registers a customer op named TimeTwo, which has one input and one output:

```
REGISTER_OP("TimeTwo")
  .Attr("T: {int32, float}")
  .Input("in: T")
  .Output("out: T")
  .SetShapeFn([](::tensorflow::shape_inference::InferenceContext* c) {
    c->set_output(0, c->input(0));
    return Status::OK();
  });
```

2. Register the operator in TVM, including using the RELAY_REGISTER_OP macro in C++ to register the operator's arity and type information, defining a C++ function to produce a call node for the operator, and registering a Python API hook for the function. For example:

```
RELAY_REGISTER_OP("gpu_cpy")
  .describe(R"code(Custom Op.

)code" TVM_ADD_FILELINE)
  .set_num_inputs(1)
  .add_argument("program", "Tuple", "The program to execute before debugging.")
  .set_support_level(1)
  .set_attrs_type<CustomOpAttrs>()
  .add_type_rel("Debug", IdentityRel)
  .set_attr<TopPattern>("TopPattern", kOpaque)
  .set_attr<FTVMCompute>("FTVMCompute", DebugCompute);

Expr MakeGpuCpy(Expr expr) {
  auto dattrs = make_object<CustomOpAttrs>();
  static const Op& op = Op::Get("gpu_cpy");
  return Call(op, {expr}, Attrs(dattrs), {});
}

TVM_REGISTER_GLOBAL("relay.op._make.gpu_cpy").set_body_typed(MakeGpuCpy);

Array<te::Tensor> TimeTwoCompute(const Attrs& attrs, const Array<te::Tensor>& inputs,
                                const Type& out_type) {
  return Array<te::Tensor>{topi::identity(inputs[0])};
}

RELAY_REGISTER_OP("time_two")
  .describe(R"code(Time Two.

)code" TVM_ADD_FILELINE)
```

```

        .set_num_inputs(1)
        .add_argument("program", "Tuple", "The program to execute before debugging.")
        .set_support_level(1)
        .set_attrs_type<CustomOpAttrs>()
        .add_type_rel("Debug", IdentityRel)
        .set_attr<TopPattern>("TopPattern", kOpaque)
        .set_attr<FTVMCompute>("FTVMCompute", TimeTwoCompute);

Expr MakeTimeTwo(Expr expr) {
    static const Op& op = Op::Get("time_two");
    return Call(op, {expr}, Attrs(), {});
}

TVM_REGISTER_GLOBAL("relay.op._make.time_two").set_body_typed(MakeTimeTwo);

```

3. Create your own classes derived from the Plugin and PluginCreator classes, implement the related functions you need. The example below only illustrates a part of the functions:

```

class TimeTwoPlugin : public Plugin {
    ...
    int Initialize();
    int Enqueue(int batchSize, const void* const* inputs, const void* const*
input_stride, void** outputs, const void* const* output_stride, void* workspace,
cudaStream_t stream);
    const char* GetPluginType() const { return "time_two"; }
    ...
}

class TimeTwoPluginCreator : public PluginCreator {
    ...
    Plugin* CreatePlugin(const char* name, const PluginFieldCollection* fc);
    const char* GetPluginName() const { return "time_two"; }
    ...
}

```

4. If you want to get better performance when executing plugins, you can also implement the GetGraph function as follows. GetGraph has the same parameters with Enqueue, and generates a graph pointer. Note that you need to prepare the parameters for the GetGraph API before creating execution context.

```

class TimeTwoPlugin : public Plugin {
    ...
    int Initialize();
    void* GetGraph(int batchSize, const void* const* inputs, const void* const*
input_stride, void** outputs, const void* const* output_stride, void* workspace,
cudaStream_t stream);
    const char* GetPluginType() const { return "time_two"; }
    ...
}

```

5. Register the created PluginCreator with macro named `REGISTER_DLNNE_PLUGIN(name, name_space)` in the tests which used the plugin op, wherein `name` is the PluginCreator name you created, `name_space` is the library the plugin belongs to. For example:

```
REGISTER_DLNNE_PLUGIN(ZeroOutPluginCreator, &ZeroOutPluginCreator::mNamespace);
```

6. Register the plugin op in TVM via interface `RegisterUserOp(tvmLibrary, pyModule, firstOpName)` while `tvmLibrary` is the library created in step 2, wherein `pyModule` is the python script which registers the op into TVM python, and `firstOpName` is the first string name of the operations you defined. For example:

```
std::string lib_name = "libcustom_op.so";
std::string library = path + lib_name;
std::string module_name = "converter.py";
std::string module = path + module_name;
const std::string output_op_name = "TimeTwo";
...
result = parser->RegisterUserOp(library.c_str(), module.c_str(), "GpuCpy");
```

```
# We use tensorflow to import and pb, tensorflow needs import those .so files to
recognize those
# TensorFlow custom ops
from tensorflow_gpu_cpy import gpu_cpy
from tensorflow_time_two import time_two

# To use dl register API
import dl
from dl import op

# Register a converter for frontend
# GpuCpy is TensorFlow op name
# When register a GlobalFunc, we use the name relay.op._make.custom_op,
# in custom_op.cc, you can see
#
TVM_REGISTER_GLOBAL("relay.op._make.custom_op").set_body_typed(MakeCustomOp);
# So in converter, we call dl.op.custom_op, the rule is relay.op._make.{Constructor},
# then the constructor will be dl.op.{Constructor}
@op.register_frontend_converter('TensorFlow', 'GpuCpy')
def converter(inputs, attrs, params, mod):
    return dl.op.gpu_cpy(inputs[0])

# Register converter for TimeTwo
@op.register_frontend_converter("TensorFlow", "TimeTwo")
def converter(inputs, attrs, params, mod):
    return dl.op.time_two(inputs[0])
```

Note: To use the plugin you constructed, do the following:

1. Make sure the layer/node name in your graph have the same name as the return value of plugin function `const char * PluginCreator::GetPluginName()`. The definition of the op in the graph as follows:

```
node {  
  name: "TimeTwo"  
  op: "time_two"  
  input: "mul"  
}
```

2. Make sure the return value of `GetPluginType()` is the same as the return value of `GetPluginName()` as the example above.

5 Configuration

dINNE provides options to configure Builder and Engine, for example, max batch size, weight share mode, working cluster, and so on.

```
struct BuilderConfig {
    int max_batch_size{1};
    WeightShareMode ws_mode{kSingle};
    IBuildModulator* callback{nullptr};
    bool dump_dot{false};
    bool dump_ir{false};
    bool print_profiling {false};
    uint32_t flags{};
};

enum BuilderFlag : uint32_t {
    kSpmAlloc = 1,          // enable optimization for SPM allocation
    kUserConst = 1 << 4,  // enable the user to control the constant memory, so different
                           // engines can share the same memory
}
```

5.1 Weight Share Mode and Cluster Option

dINNE can do multi-cluster configuration as a Goldwasser AI accelerator card can do inference on multiple clusters in parallel.

1. Configure weight share in Builder. Weight share means sharing weight memory on different clusters, which can save memory cost effectively.

```
enum WeightShareMode {
    kSingle = 1,  // inference network on single cluster without weight share
    kShare2 = 2,  // inference network on two clusters with weight share
    kShare4 = 4,  // inference network on four clusters with weight share
}

WeightShareMode mode = kSingle;

auto builder = CreateInferBuilder();

BuilderConfig builder_cfg;
builder_cfg.ws_mode = mode;

builder->SetBuilderConfig(builder_cfg);
```

2. Configure clusters when Engine builds execution context.

To build a WeightShareMode with kSingle, user can specify the single cluster (kCluster0, kCluster1, kCluster2, kCluster3) which do inference for the network.

To build a WeightShareMode with kShare2, user can specify the two clusters (kCluster01, kCluster23, kCluster02, kCluster13, kCluster03, kCluster12) which do inference for the network.

To build a WeightShareMode with kShare4, user must specify the four clusters (kCluster0123) which do inference for the network.

```
enum ClusterConfig {
    kCluster0,
    kCluster1,
    kCluster2,
    kCluster3,
    kCluster01,
    kCluster23,
    kCluster02,
    kCluster13,
    kCluster03,
    kCluster12,
    kCluster0123,
}

ClusterConfig config = kCluster2; // specify cluster 2 doing inference
engine->CreateExecutionContext(config);
```

5.2 Builder Flag Option

Option `flags` is used to set the `BuilderFlag` parameters. **Note:** Though `flags` is a member of `BuilderConfig`, `SetBuilderConfig` does not set `flags`. To set `flags`, use `SetFlag` or `SetFlags` instead.

```
auto builder = CreateInferBuilder();

builder->SetFlag(kSpmAlloc); // append a new flag kSpmAlloc
builder->SetFlags(kSpmAlloc | kUserConst); // reset flags to this input
builder->GetFlag(kSpmAlloc); // query whether kSpmAlloc is enabled
builder->GetFlags(); // get all enabled flags as bitmask
```

5.3 Dump Dot Option

Option `dump_dot` provides a visual representation for the network. Default is `false`, when user sets it to `true`, it generates a dot file in the current working directory.

```
auto builder = CreateInferBuilder();

BuilderConfig builder_cfg;
builder_cfg.dump_dot = true;

builder->SetBuilderConfig(builder_cfg);
```

5.4 Dump Relay IR Option

When `dump_ir` is set to `true`, Relay IR prints on the standard output.

```
auto builder = CreateInferBuilder();

BuilderConfig builder_cfg;
builder_cfg.dump_ir = true;

builder->SetBuilderConfig(builder_cfg);
```

5.5 Modulator callback Option

Option `callback` is used to set `IBuildModulator`. See [dINNE Build Modulator](#) for details.

5.6 Print Profile Option

When `print_profiling` is set to `true`, how much time each queue on the device takes is output to stdout after a network model is run.

```
auto builder = CreateInferBuilder();

BuilderConfig builder_cfg;
builder_cfg.print_profiling = true;

builder->SetBuilderConfig(builder_cfg);
```

5.7 User Const Option

When flag `kuserConst` is enabled, you should get hash and size for per const memory by `GetConstInfoEXT`, allocate many pieces of device memory, copy the constant contents to per device memory by `UploadConstEXT`, and configure per device memory for the engine by `SetConstMemoryEXT`. In this way, several engines can share the same piece of const memory if there constants are the same.

Note:

Make sure that `UploadConstEXT` and `SetConstMemoryEXT` return `true` before creating execution context.

1. Directly create the engines, here the default `ClusterConfig kCluster0` is for example, other configs are also fine as long as their memories are correctly allocated. Use `GetConstInfoEXT` to get all const info, you should call this func twice, first to get num of const, then get hash and mem size.

```
#include "cu_ext.h"

// set the builder flag
auto builder = CreateInferBuilder();
builder->SetFlag(kUserConst);
auto engine0 = builder->BuildEngine(network0);
auto engine1 = builder->BuildEngine(network1); // network0 and network1 must have the
same constants

uint32_t const_size{}, const_size1{};
// num_of_const_ret is not null, this time will get size of const
engine->GetConstInfoEXT(0, nullptr, nullptr, &const_size);
engine1->GetConstInfoEXT(0, nullptr, nullptr, &const_size1);
```



```

std::vector<uint64_t> hash_vec(const_size, 0);
std::vector<uint64_t> size_vec(const_size, 0);
std::vector<uint64_t> hash_vec1(const_size1, 0);
std::vector<uint64_t> size_vec1(const_size1, 0);
//numOfConstRet is null, this time will get hash and mem size of per const
engine->GetConstInfoEXT(const_size, hash_vec.data(), size_vec.data(), nullptr);
engine1->GetConstInfoEXT(const_size1, hash_vec1.data(), size_vec1.data(), nullptr);

```

2. Alloc memory for per const , upload const and set const for engine.

```

std::unordered_map<uint64_t, void*> const_map{};
for (int i = 0; i < const_size; i++)
{
    void* const_mem;
    cudaMalloc(&const_mem, size_vec[i]);
    engine->UploadConstEXT(hash_vec[i], const_mem);
    engine->SetConstMemoryEXT(hash_vec[i], const_mem);

    const_map[hash_vec[i]] = const_mem;
}

```

3. set const memory for engine1.

```

for (int i = 0; i < const_size1; i++)
{
    //try find hash in const_map, it means we can share const with engine in step2
    if (const_map.find(hash_vec1[i]) != const_map.end())
    {
        engine1->SetConstMemoryEXT(const_map.find(hash_vec1[i]).first,
const_map.find(hash_vec1[i]).second);
    }
    else//not find, alloc mem, upload and set
    {
        void* const_mem;
        cudaMalloc(&const_mem, size_vec1[i]);
        engine1->UploadConstEXT(hash_vec1[i], const_mem);
        engine1->SetConstMemoryEXT(hash_vec1[i], const_mem);
    }
}

```

4. you can use `SerializeWithoutConstEXT` to serialize without some const memory.

```

//each hash in hash_list should be found in engine
auto mem = engine->SerializeWithoutConstEXT(hash_list.data(), hash_list.size())

```

6 dINNE Build Modulator

dINNE provides a callback mechanism to participate the graph compilation process, which is driven by interface `IBuildModulator`.

The following steps illustrate how to use the modulator.

1. Implement interface `IBuildModulator`:

```
class BuildModulatorImpl : public dl::nne::IBuildModulator {
    bool ScheduleDAG(IDAG *graph) override {
        // step 1. get all nodes in graph
        auto nodes = graph->GetNodes();

        // step 2. do your schedule
        auto scheduled_nodes = YourSchedule(nodes);

        // step 3. set the scheduled nodes back to graph
        graph->SetNodes(scheduled_nodes);

        return true;
    }

    bool GenerateSubgraph(const IDAG *graph,
                        ISubgraphContainer *container) override {
        // step 1. partition the graph to subgraphs as you want
        std::vector<IDAGNodeList*> subgraphs = YourPartition(graph);

        // step 2. add the subgraphs to the subgraph container
        for (auto &nodes : subgraphs) {
            container->AddSubgraph(&nodes);
        }

        return true;
    }
};
```

2. Set the modulator to the dINNE builder by `BuilderConfig`:

```
BuildModulatorImpl modulator;
Builder* builder = CreateInferBuilder();

BuilderConfig builder_cfg;
builder_cfg.callback = &modulator;
builder->SetBuilderConfig(builder_cfg);
```

Note: To work fine with the modulator, do the following:

1. For `ScheduleDAG()`:
 - After your scheduling, the number of nodes of the graph should be identical to that of the origin graph.
 - You should not add your own node to the graph.

2. For GenerateSubgraph():

- The total number of nodes in all subgraphs should be identical to that of the origin graph.
 - The subgraphs should be added in order.
3. If you do not want to implement some of the interfaces, just return false.

For more details, see the modulator samples in Hamming™ SDK and document **dINNE-Build-Modulator-API.pdf**.

登临科技保密材料

7 Network Configuration

dINNE provides options to configure Network, for example, `--device-profile`, `--continuous-tiling`.

`--device-profile`

`--device-profile` describes GPU device status, if device profile is already set by user, dINNE will build the `Engine` that can fit in the states specified, also provides a way to build an `Engine` offline, ie, without GPU environment.

By default, device profile is not set, and dINNE will check current GPU device states to build a suitable `Engine`.

Supported device profiles are as follows:

Number	Device Profile	Product Name
1	v2.4c.4t.512	Goldwasser II-GS40
2	v2.4c.2t.7s.256	Goldwasser II-GS30
3	v2.4c.2t.6s.4m.128	Goldwasser II-GS25
4	v2.4c.1t.4s.4m.50	Goldwasser II-KS21-T
5	v2.4c.1t.2s.2m.25	Goldwasser II-KS15-T

The example below shows how to build an `Engine` that can be executed on device "v2.4c.2t.7s.256" with weight share mode being `kShare2`:

```
auto network = builder->CreateNetwork();
network->SetConfig("--device-profile=v2.4c.2t.7s.256");
parser->Parse(model_file, *network);
BuilderConfig builder_cfg{};
builder_cfg.ws_mode=kShare2;
builder->SetBuilderConfig(builder_cfg);
auto engine = builder->BuildEngine(*network);
```

`--continuous-tiling`

Whether to enable continuous Tiling Ops. Tensor compiler will determine whether it is possible to tile based on the semantics of op in the subnet, the dimensions of tiles, whether continuous tiles are possible, and the performance benefits brought by tiles to determine whether continuous Tiling is necessary for these OPs

Supported options are as follows:

```
option = 0, 1
Default value = 1
```

The example below shows how to build an `Engine` that enable "`--continuous-tiling`":

```
auto network = builder->CreateNewwork();
network->SetConfig("--continous-tiling=1");
parser->Parse(model_file, *network);
auto engine = builder->BuildEngine(*network);
```

--tear-tu, --mat-vec-ping-pong-schedule

Whether to differentiate between the Matrix and Vector modules of TU Op. If enabled, TU Op will be divided into Matrix TU Op and Vector TU Op according to their functions during instruction scheduling and allocation, in order to achieve better parallel execution results.

Supported options are as follows:

```
Option = 0, 1
Default value = 1
```

The example below shows how to build an `Engine` that enable "--mat-vec-ping-pong-schedule":

```
auto network = builder->CreateNewwork();
network->SetConfig("--mat-vec-ping-pong-schedule=1");
parser->Parse(model_file, *network);
auto engine = builder->BuildEngine(*network);
```

--graph-grid-mode

Set the partitioning method for the Graph Grid. The optional modes are greedy merge and topo aggregate, with the default being greedy merge. Greedy merge will continuously merge nodes from the graph into a Graph Grid until Spm cannot fit or Cost deteriorates. The topo aggregate uses the aggregation points of the graph as the partitioning points of the Graph Grid.

Supported options are as follows:

```
Option = greedy-merge, topo-aggregate
Default value = greedy-merge
```

The example below shows how to build an `Engine` that enable "graph-grid-mode" with "--topo-aggregate":

```
auto network = builder->CreateNewwork();
network->SetConfig("--graph-grid-mode=topo-aggregate");
parser->Parse(model_file, *network);
auto engine = builder->BuildEngine(*network);
```