



# 登临 Hamming<sup>TM</sup> V2

## 软件栈简介

DL-DG/SW-040A-04

2024-12-17

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5号1101室，江苏，中国

<http://www.denglin.ai>

email : support@denglin.ai

## 更新历史

版本	更新描述
04	更新章节 <b>1.3.1 dICU</b> ；更新章节 <b>1.3.3 dINNE</b> 。
03	新增软件栈编译器工具链dITC和dITVM及说明。
02	内容修订；修改关于dIDNN的说明；更新dICU章节。
01	初始版本。

# 章节目录

- 1. HammingTM V2 软件栈介绍
  - 1.1 术语表
  - 1.2 Hamming V2 总体架构
  - 1.3 Hamming V2 开发者接口简介
    - 1.3.1 dICU
      - [dlCU-Programming-Guide.pdf](#)
      - [dlCU-Extended-API.pdf](#)
      - [dlCU-Extended-Math-API.pdf](#)
      - [DengLin-Compute-Compiler-User-Guide.pdf](#)
      - [DengLin-Hamming-V2-Texture-Feature-Guide.pdf](#)
    - 1.3.2 dlJPEG
      - [dlJPEG-Programming-Guide.pdf](#)
      - [dlJPEG-API.pdf](#)
    - 1.3.3 dINNE
      - [dINNE-Developer-Guide.pdf](#)
      - [dINNE-API.pdf](#)
      - [dINNE-Build-Modulator-API.pdf](#)
      - [dINNE-ONNX-Quantization-Operator.pdf](#)
      - [dINNE-Quant-Introduction-To-Quantization.pdf](#)
      - [dINNE-Quant-Quantization-Tutorial-With-ResNet-V1-50.pdf](#)
      - [dINNE-Quant-TU-Operator.pdf](#)
      - [dINNE-TVM-Quantization.pdf](#)
      - [dlTVM Relay IR 子图划分操作指导.pdf](#)
      - [dlTVM Frontend Operator Coverage](#)
      - [dlTVM Relay Core Tensor Operators](#)
    - 1.3.4 dIVID
      - [dIVID-Programming-Guide.pdf](#)
      - [dIVID-API.pdf](#)
    - 1.3.5 dIOCL
    - 1.3.6 dIML
      - [dIML-API.pdf](#)
    - 1.3.7 dISS
    - 1.3.8 dIBLAS
      - [dIBLAS-Introduction.pdf](#)
      - [dIBLASLt-Introduction.pdf](#)
    - 1.3.9 dISOLVER
      - [dISOLVER 使用说明.pdf](#)
- 2. 使用 dINNE
  - 2.1 如何整合 dINNE
    - 2.1.1 工作流
    - 2.1.2 集成编译器
    - 2.1.3 集成 Runtime
  - 2.2 如何量化模型
  - 2.3 使用 C++ API
  - 2.4 使用 Python API
  - 2.5 使用 Custom op
    - 2.5.1 实现 Custom Relay op
    - 2.5.2 为 custom op 注册 custom kernel
    - 2.5.3 将定义好的 custom op 动态导入 dINNE
- 3. 使用 dICU

#### 4. 使用命令行工具dlsmi

# 1. Hamming™ V2 软件栈介绍

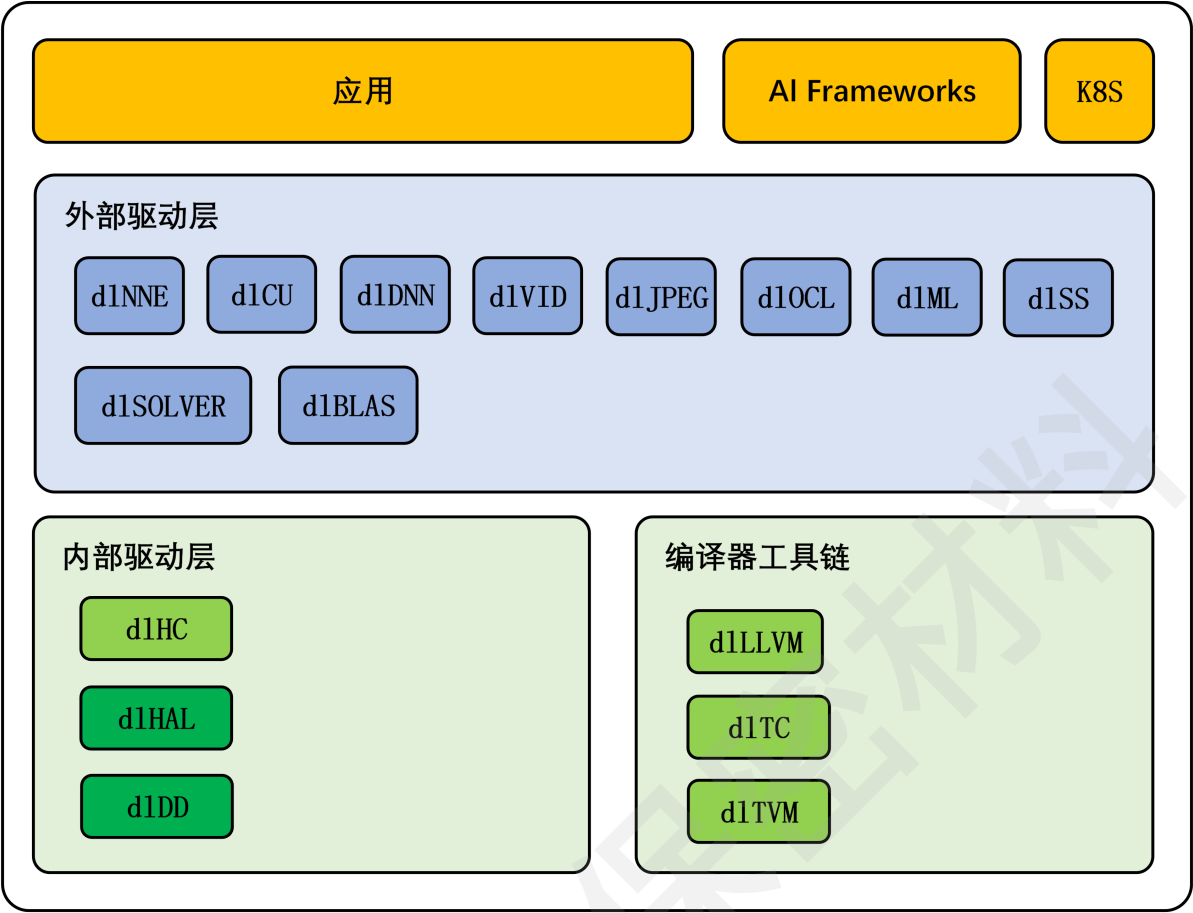
## 1.1 术语表

简称	描述
dINNE	登临 Neutral Network Engine
dICU	兼容 CUDA 应用的驱动运行时库
dIDNN	提供兼容 cuDNN API 的库
dIJPEG	JPEG Codec 库
dIVID	Video Codec 库
dIOCL	OpenCL 库
dIML	提供登临设备监控和管理功能的库
dISS	登临系统软件包（包括GPU容器应用、系统监测、推理服务器、GPU虚拟化等软件包）
dIHC	HC 是 Heterogeneous Compute 的缩写, 它提供了面向 CU 和 TU 编程的内部驱动 API
dILLVM	基于 LLVM 的 CUDA&OpenCL C/C++ 编译器
dIKL	登临算子 Kernel 库
dIHAL	Hardware Abstract Layer 硬件抽象层驱动
dIDD	设备驱动程序（包括 Linux Kernel Device Driver）
dIBLAS	登临平台BLAS库，兼容cuBLAS
dISOLVER	dISOLVER 支持CUDA 11.7 cuSOLVER的部分求解功能API
dITC	登临Tensor编译器
dITVM	基于tvm实现的dINNE前端

## 1.2 Hamming V2 总体架构

Hamming V2 软件栈是驱动登临 Goldwasser™ II AI 加速卡的软件包。

Hamming V2 软件栈示意如下：



Hamming V2 软件栈示意图

提示：

Hamming V2 软件栈的模块在不同系统中略有区别，详细情况请咨询support@denglin.ai。

1.3 Hamming V2 开发者接口简介

Hamming 提供以下接口：

- d1CU
- d1JPEG
- d1VID
- d1NNE
- d1DNN
- d1ML
- d1BLAS
- d1SOLVER

### 1.3.1 dICU

dICU 是一个兼容标准 CUDA 编程的库。它提供了一组与 CUDA Driver&Runtime 兼容的 API 和一个兼容 CUDA C/C++ 的编译器 dlcc。dICU 相关文档如下：

#### dICU-Programming-Guide.pdf

该文档介绍了 dICU 编程模型，dICU 编程接口和 C++ 扩展 API。

- dICU 编程模型介绍了 dICU 编程模型背后的主要概念。
  - Kernel 是 dICU C 允许程序员定义的 C 函数。dICU 通过 kernel 扩展了 C。kernel 被调用时会被 N 个 dICU 线程执行 N 次。kernel 使用 `__global__` 声明符定义，使用一种新的 `<<< ::>>>` 执行配置语法指定执行某一指定 kernel 的线程数。
  - 线程层次 (Thread Hierarchy)：线程可以使用一维，二维，三维索引标识，形成一维，二维，三维的线程块。一个块内的线程数目是有限的。在目前的 GPU 上，一个线程块可以包含多达1024个线程。一个 kernel 可被多个同样大小的线程块执行，所以总的线程数等于每个块内的线程数乘以线程块数。线程块内线程数和网格内线程块数由 `<<< ::>>>` 语法确定，参数可以是 `int` 或者 `dim3` 类型。网格内的每个块可以通过一维、二维或三维索引唯一确定，在 kernel 中此索引可通过内置的 `blockIdx` 变量访问。
  - 存储器层次 (Memory Hierarchy)：dICU 线程在执行时可以访问多个存储空间。每个线程块有对块内所有线程可见的共享存储器 (shared memory)，共享存储器的生命期和块相同。所有的线程都可以访问同一全局存储器 (global memory)。
  - 异构编程 (Heterogeneous Programming)：dICU 编程模型假定线程在物理上独立的设备 (device) 上执行，设备作为主机 (host) 的协处理器。dICU 编程模型同时假定主机和设备各自维护着独立的 DRAM 存储器空间，分别被称为主机存储器空间和设备存储器空间。因此，程序通过调用 dICU runtime 来管理全局存储器空间，包括设备存储器的分配和释放，主机和设备间的数据传输。
- dICU 编程接口是 dICU 向熟悉 C 语言的用户提供的一种简单的编写在设备上执行的代码的方式。dICU C 包括 C 的最小扩展集和一个 runtime library。dICU 编程模型章节介绍了语言的核心扩展，这些扩展允许程序员像定义 C 函数一样定义 kernel 和在每次 kernel 调用时使用新的语法指定网格和块的尺寸。任何包含这些扩展的源文件都必须使用 dlcc 编译。
  - dlcc 是基于 LLVM/clang 的 CUDA 和 OpenCL 编译工具。详见文档 **DengLin-Compute-Compiler(dlcc)-User-Guide.pdf**。
  - dICU runtime 由 curt 库实现，所有入口点的前缀都是 `cuda`。dICU runtime 提供的 C 函数运行在主机上，用于分配和释放设备存储器、在主机存储和设备存储间传输数据、管理多设备的系统等。

dICU runtime API 是基于更底层的 dICU driver API 构建的，应用也可以访问 driver API。driver API 通过展示底层概念提供了额外的控制，如 dICU context (类似设备上的主机进程)、和 dICU 模块 (类似设备上的动态链接库)。大多数应用不使用 driver API，因为在使用 runtime API 时它们不需要这些额外的控制，context 和模块管理都是隐式的，因此代码更简明。

- 初始化：dICU runtime 没有显式的初始化函数，当第一次调用一个 runtime 函数时初始化。在初始化时，runtime 为系统中的每个设备建立一个 dICU context。该 context 作为设备的 primary context，被应用中的主机线程共享。当主机线程调用 `cudaDeviceReset()` 时，会销毁被主机线程操作的设备的 primary context。任何以这个设备为当前设备的主机线程调用的 runtime 函数将为设备重新建立一个 primary context。
- 设备存储器 (Device Memory)：异构编程章节提到 CUDA 编程模型假定系统包含主机和设备，各自有独立的存储器。kernel 不能操作设备存储器，所以 dICU runtime 提供了函数用于分配，释放，拷贝设备存储器，和在设备与主机间传输数据。



设备存储器可被分配为线性存储器。线性存储器存在于40位地址空间内，所以独立分配的存储器实体能够通过指针引用。线性存储器通常使用 `cudaMalloc()` 分配，通过 `cudaFree()` 释放，使用 `cudaMemcpy()` 在设备和主机间传输。

- 共享存储器 (Shared Memory)：共享存储器通过 `__shared__` 存储器空间区分符来分配。共享存储器应当比全局存储器更快，故应尽量使用共享存储器访问而非全局存储器。该章节提供了一个矩阵相乘的代码示例。该示例中，通过将计算分块，利用共享存储器，节约了全局存储器带宽，因为在全局存储器中，A只被读了 (B.width/block size) 次同时B读了 (A.height/block size) 次。
- 异步并发执行 (Asynchronous concurrent execution)：dICU 将主机上的运算，设备上的运算，主机与设备间的存储转移，指定设备的存储器中的存储转移视为独立的可以同时执行的操作。

- 主机和设备间的异步执行：异步程序库函数在设备完成请求的任务前就将控制权返给主机线程。通过异步调用，许多设备操作会排队等待被 dICU 驱动器执行。这种方式减轻了主机线程管理设备的负担，让其能执行其他的操作。kernel 启动，单个设备的存储复制，和由带有 Async 后缀的函数执行的存储拷贝操作，这三个设备操作相对于主机是异步的。
- 数据传输和 kernel 执行重叠：设备内拷贝可与 kernel 执行或/和设备拷贝同时执行。设备内拷贝是通过标准存储器拷贝函数启动，并且目标地址和源地址位于同一设备上。
- 流 (Streams)：应用通过流管理并发操作。流是一系列顺序执行的命令 (可能是不同的主机线程发起的)。不同流之间会相对无序地或并发地执行它们的命令，因而无法保证正确性。dICU 提供了流的创建和销毁，默认流，显式同步，隐式同步，和回调函数等方式来控制流。
- 图 (Graphs)：图是一种新的工作提交模型。图是一系列操作的组合，例如 kernel 启动。图的定义与执行是分离的。一张图被定义一次，就可以重复地启动。与流相比，将图的定义和执行分离能带来很多优化空间。图中一个节点 (node) 代表一个操作，边界代表操作间的依赖。这些依赖约束着操作的执行顺序。图可以通过两种方式创建：Graph APIs 和 流捕获 (stream capture)。

该章节详细描述了图的结构，图的创建和使用方法，利用流捕获创建图时如何处理流间依赖和无效的操作等。

- 事件 (Events)：通过在应用的任意点上异步地记载事件和查询事件于何时完成，runtime 精密地监测设备运行进度并准确地计时。当一个事件之前的所有任务或指定的流中的所有命令都完成后，该事件才完成。当所有流中的所有之前的任务和命令都完成后，零号 stream 中的事件才完成。

该章节介绍了如何创建和销毁事件，如何用事件记录事件创建和销毁之间所消耗的时间。

- 同步调用：直到设备真正完成任务，同步函数调用的控制权才会返回给主机线程。
- 表面存储器 (Surface memory)：使用 `cudaArraySurfaceLoadStore` 标签建立的 CUDA 数组，可以通过表面对象 (surface object) 或表面参照 (surface reference)，使用表面函数 (详见文档 **dICU Programming Guide** 的附录 **A.1 Surface Functions**) 读写。最大表面高和宽都是65535。
  - 表面对象 API：一个表面对象是通过 `cudaCreateSurfaceObject()` 从类型为 `cudaResourceDesc` 的资源描述符创建的。
  - CUDA 数组是为图片像素获取而优化的不透明的存储器层次。CUDA 数组可以是一维的，二维的，或三维的 (暂不支持)，也可由多个元素组成，每个元素可有1, 2或4个组件，这些组件可能是有符号或无符号8, 16或32位整形，16位浮点，或32位浮点。CUDA 数组只能由 kernel 通过表面函数访问。
- 附录 C++ 扩展 API，介绍了如何使用 dICU 表面函数 (Surface function) 读写 CUDA 数组。表面函数中的 `boundaryMode` 指定边界模式 (boundary mode)，边界模式指定了越界 (out-of-range) 的表面坐标如何处理。

## dICU-Extended-API.pdf

该文档介绍了 dICU 扩展函数：

- `cuMemAllocChannel`：在 DDR channel 上分配存储器。
- `cudaSetClusterMask`：通过指定 cluster mask 来指定待用的 cluster。
- `cudaGetClusterMask`：获取当前 cluster mask。
- `cudaGetSpm`：获取 SPM 存储器。
- `cudaEnterSpmSection`：指定流独占 SPM。
- `cudaLeaveSpmSection`：指定流停止独占 SPM。
- `cuModuleGetFunctionAsync`：在制定流上异步获取模块中函数。
- `cudaGetDeviceProperties_ext`：返回设备信息。

## dICU-Extended-Math-API.pdf

该文档介绍了 dICU 的扩展 Build-in APIs：

- 量化和反量化：登临 Minsky™ 架构优化了量化和反量化，使其具有更高的性能。该章节介绍了量化和反量化的原理，使用方法和详细的 API。
- SGLDG/SGSTG：登临 Minsky 架构为优化连续读写全局存储的效率，设计了 SGLDG/SGSTG 指令。该章节介绍了 SGLDG 和 SGSTG 的原理，使用方法和详细的 API。

## DengLin-Compute-Compiler-User-Guide.pdf

该文档介绍了 dlcc 的使用方法。dlcc 是基于 LLVM/clang 的 CUDA 和 OpenCL 编译工具，提供了很多选项，如指定 GPU 最多可用的寄存器，允许积极的损耗更大的浮点数优化，生成动态链接库等，赋予了程序更灵活的控制权。

## DengLin-Hamming-V2-Texture-Feature-Guide.pdf

该文档介绍了 Denglin Tetxure 特性。Denglin Texture 特性主要应用于深度学习推理的预处理与后处理模块。Denglin Texture 当前版本只支持 CUDA Texture 部分特性，具体情况请阅读文档 **DengLin Hamming V2 Texture Feature Guide.pdf**。

### 1.3.2 dIJPEG

dIJPEG 提供了一组 JPEG 硬件编解码处理的 APIs。dIJPEG 相关文档如下：

#### dIJPEG-Programming-Guide.pdf

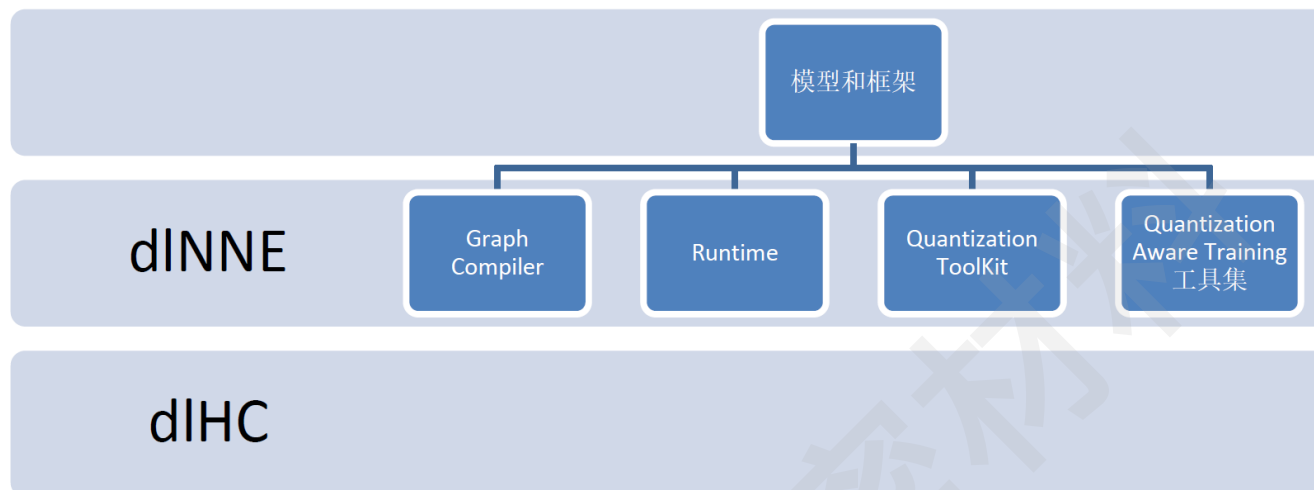
该文档介绍了 JPEG 编解码引擎，工作流程，及如何使用 dIJPEG APIs 编程 JPU。

- 概述：登临 Goldwasser AI 加速卡内置多个 JPEG 编解码引擎 (简称 JPU)。不同的 JPU 可以并行编解码。dIJPEG APIs 用于 JPU 编程。
- 编码是将 YUV 数据编码成 JPEG 图片。工作流程如下：
  1. 获取 JPU device 并在 device 上创建一个 JPEG 会话 (session)。

2. 在 device memory 上为 JPU 分配 input buffer 和 output buffer。
  3. 将 YUV 数据复制到 input buffer。
  4. 进行编码并输出压缩的 JPEG 数据。
  5. 将生成的 JPEG 头文件和压缩的 JPEG 数据组合以生成完整的 JPEG 图片。
  6. 清理。
- 解码是将 JPEG 图片解码成 YUV 数据。工作流程如下：
    1. 获取 JPU 并在 device 上创建一个 JPEG 会话。
    2. 解析 JPEG 头文件以获取 output buffer 的大小。
    3. 在 device memory 上为 JPU 分配 input buffer 和 output buffer。
    4. 将 JPEG 图片复制到 input buffer。
    5. 解码 JPEG 图片并输出 YUV 数据。
    6. 清理。
  - 使用 dlJPEG APIs:
    - `dljpegGetDevice()` 获取一个 JPEG device。
    - `dljpegCreateSession()` 创建一个 JPEG session。编解码的任务都是在 JPEG 会话中执行的。
    - (仅适用于解码) 调用 `dljpegParseHeader()` 解析 JPEG 头文件信息。
    - device memory：JPU 的输入数据和输出数据都存放在 device memory 中。如果输入数据存放在 host memory，必须将其复制到 device memory。
      - `dljpegMalloc()` 为 JPU 分配 device memory。
      - `cudaMemcpy()` 在 host memory 和 device memory 之间复制数据。
      - `cudaFree()` 释放由 `dljpegMalloc()` 分配的 device memory。
    - 解码：
      - `dljpegDecode()` 工作在同步模式下，带有的参数可以设置 scale，rotation，ROI 等。输出的 YUV 数据会被存放到1至3个 buffer 中，具体的数量由输出格式的平面 (output format plane) 的数量决定。比如 YUV420P 输出数据会存放到3个 buffer 中，而 NV12 输出数据会被存放到2个 buffer 中。
      - `dljpegDecodeAsync()` 工作在异步模式下。该函数的参数中带有一个 CUDA event。当解码完成时，该 CUDA event 会被触发。input buffer 和 output buffer 必须保留到解码完成。
    - 编码: `dljpegEncode()` 工作在同步模式下。带有的参数可以设置 scale，rotation，encode quality 等。输入的 YUV 数据存放在不同的 buffer 中，由输入格式的平面 (input format plane) 的数量决定。输出数据的大小由参数 `scanline_size` 获取。
    - (仅适用于编码) `dljpegFillFileHeader()` 创建 JPEG 头文件。
    - (仅适用于编码) 用户定制 Huffman 表和量化表 (quantization table)。高级用户可以定制 Huffman 表和量化表。如果没有定制表，编码时会使用默认表。
    - 清理：
      - `cudaFree()` 释放 `dljpegMalloc()` 分配的 buffer。
      - `dljpegFreeHeader()` 释放 `dljpegParseHeader()` 分配的头信息。
      - `dljpegDestroySession()` 销毁 JPEG session。
      - 不需要销毁 JPEG device。

**dIJPEG-API.pdf**

dIJPEG-API.pdf 是 dIJPEG APIs 的参考文档。

**1.3.3 dINNE**

dINNE 的主要部分是一个 C++ 库。dINNE 同时提供 C++ 和 Python APIs。dINNE 的主要接口有：

- Graph Compiler  
dINNE Compiler 的作用是将已经训练好的模型编译为 dINNE Runtime 可以使用的模型。
- Runtime Engine  
dINNE Runtime 用来在登临加速卡上执行编译好的模型。
- Training Aware Quantization 支持工具集  
为用户进行 Training Aware Quantization 进行支持。

dINNE 相关文档如下：

**dINNE-Developer-Guide.pdf**

该文档介绍了 dINNE 及其使用方法，主要包含以下内容：

- dINNE 是用于优化高性能神经网络推理的 C++ 库，或是可被部署到生产环境中的运行时引擎 (runtime engine)。
  - 为了优化推理模型，dINNE 会解析神经网络，执行优化，并生成推理引擎，该过程称为构建阶段。构建阶段可能需要相当长的时间，尤其是在其他平台上运行时。因此，典型的应用会构建一次引擎，然后将其序列化为计划文件以备后用。

构建阶段在图上执行以下优化操作：

- 消除不使用输出的层
- 消除相当于 no-op 的操作
- TU 模式的融合 (卷积、偏置、激活、池化 ...)
- 除 TU 之外的 CU 操作的融合
- 通过将层输出定向到正确的最终目的地来合并串联层

- 折叠常量层
- dINNE 使开发人员能够导入、生成和部署优化的网络。网络可以通过 ONNX, DarkNET, TensorFlow 或其他框架导入。用户还可以使用插件接口 (Plugin interface) 通过 dINNE 运行自定义层。dINNE 提供了 C++ 和 python 实现。

dINNE 核心库的关键接口包括：

- 网络 (Network)：用于解析网络模型。
- 构建器 (Builder)：Builder 界面允许从网络和 builder 配置创建优化引擎。
- 引擎 (Engine)：引擎接口允许应用程序执行推理。它支持同步和异步模式，以及引擎输入和输出绑定的枚举和查询。单个引擎可以有多个执行上下文，从而允许使用一组经过训练的参数同时执行多个批处理。
- 解析器 (Parser)：该解析器可用于解析 TensorFlow 网络序列化的 pb 文件。
- 使用 dINNE API, 详见文档 **dINNE-API.pdf**。

1. 从模型创建一个 dINNE 网络，并使用支持 TensorFlow、ONNX 和 Caffe 格式的 dINNE 解析器导入模型。

1. 创建 builder 对象和网络对象。
2. 创建网络模型解析器。
3. 向解析器声明网络输入张量的信息，包括名称，形状，格式，和输出张量的名称。
4. 解析网络模型文件以填充网络。

2. 创建一个引擎，调用 dINNE builder 来创建优化的运行时可执行文件。

1. 使用 builder 对象创建引擎对象。
2. 如果使用了网络、构建器和解析器，则释放它们。

3. 序列化一个模型。

在将模型用于推理之前，可以选择是否序列化和反序列化模型。引擎对象也可以直接用于推理。序列化将引擎转换为一种便于稍后存储和使用的格式，只需反序列化引擎即可用于推理。由于从网络创建引擎可能很耗时，因此将其序列化一次，并在运行推理时反序列化它来避免每次应用程序重新运行时重建引擎。引擎构建完成后，用户通常将其序列化以备后用。具体步骤如下：

1. 运行构建器，然后序列化。
2. 创建一个运行时对象来反序列化。

4. 使用引擎执行推理：

1. 创建一些空间来存储中间激活值。

由于引擎保存网络和训练参数，因此需要额外的空间。这些数据都被保存在执行上下文中。一个引擎可以有多个执行上下文，允许一组权重用于多个重叠的推理任务。例如，您可以使用一个引擎和每个流一个上下文处理并行 CUDA 流中的图像。

2. 使用输入输出 blob 名称获取对应的输入输出索引。
3. 使用这些索引，设置一个指向输入和输出缓存区的缓存区数组。
4. 以异步或同步模式执行 dINNE。

如果数据不存在，那么在 kernels 从 GPU 移动数据之前和之后将异步 `memcpy()` 入队是很常见的。`enqueue()` 的最后一个参数是一个可选的 CUDA 事件，当输入缓存区被消耗并且它们的内存可以安全地重用，它将发出信号。

- 使用自定义层扩展 dINNE。

除了 TU 和 CU 模式之外，dINNE 还支持自定义层 (称为插件)，由应用程序实现并实例化，而它们的生命周期必须跨越它们在 dINNE 引擎中的使用。

1. 在 TensorFlow 中构建一个 op。



2. 在 TVM 中注册算子，包括使用 C++ 中的 RELAY\_REGISTER\_OP 宏注册算子的元数和类型信息，定义一个 C++ 函数来为算子生成调用节点，以及为该函数注册一个 Python API hook。
  3. 从 Plugin 和 PluginCreator 类派生用户的类，实现用户需要的功能。
  4. 为了在执行插件时获得更好的性能，也可以使用 GetGraph 函数。GetGraph 的参数与 Enqueue 相同，可以生成图形指针。
  5. 使用名为 REGISTER\_DLNN\_PLUGIN(name, name\_space) 的宏注册创建的 PluginCreator，其中 name 是用户创建的 PluginCreator 名称，name\_space 是插件所属的库。
  6. 通过接口 RegisterUserOp(tvmLibrary, pyModule, firstOpName) 在 TVM 中注册插件 op，而 tvmLibrary 是在步骤 2 中创建的库，其中 pyModule 是将 op 注册到 TVM python 中的 python 脚本，firstOpName 是用户定义的 operations 的第一个字符串名称。
- 配置 dINNE。
    - 权重共享模式 (WeightShareMode) 选项和 cluster 选项。dINNE 可以进行多 cluster 配置，因为 Goldwasser AI 加速卡可以在多 cluster 上执行并行推理。
      1. 在 Builder 中配置 WeightShareMode。权重共享意味着 cluster 可以共享不同 cluster 上的权重内存，从而有效地节省内存成本。
      2. 当引擎构建执行上下文时，配置 cluster。例如：要使用 kShare2 构建 WeightShareMode，用户可以指定对网络进行推理的两个 cluster (kCluster01, kCluster23)。
    - dump\_dot 选项提供网络的可视化表示。该选项默认被设为 false，当设置为 true 时，会在当前工作目录中生成一个点文件。
    - dump\_ir 选项设为 true 时，Relay IR 会被打印在标准输出上。
    - Modulator callback 选项，详见文档 **dINNE-Developer-Guide.pdf** 的章节 *dINNE Build Modulator*。
    - print\_profiling 选项。当该选项设为 true 时，设备上的每个队列花费的时间在网络模型运行后会被输出到标准输出。
  - dINNE Build Modulator，提供了参与图编译过程的回调机制，该机制由接口 IBuildModulator 驱动。详见文档 **dINNE-Build-Modulator-API.pdf**。
    1. 实现接口 IBuildModulator。
    2. 通过 BuilderConfig 将 modulator 设置为 dINNE builder。

详见 Hamming SDK 中的 modulator 样例。

## dINNE-API.pdf

dINNE API 优化高性能神经网络推理，可用于从模型创建 dINNE 网络，创建引擎，序列化和反序列化模型，执行推理等。该文档介绍了 dINNE API 的类层次结构 (Class Hierarchy) 和完整的 API (Full API)，包括命名空间 (Namespaces)，类和结构体 (Classes and Structs)，枚举类型 (Enums)，函数 (Functions)，宏定义 (Defines)。

## dINNE-Build-Modulator-API.pdf

dINNE Build Modulator API 提供了参与图编译过程的回调机制。该文档介绍了 dINNE Build Modulator API 的类层次结构 (Class Hierarchy) 和完整的 API (Full API)，包括命名空间 (Namespaces)，类和结构体 (Classes and Structs)，枚举类型 (Enums)，类型定义 (Typedefs)。

### dINNE-ONNX-Quantization-Operator.pdf

该文档描述了登临定义的 ONNX quantization operators，并给出了如何用 ONNX API 进行构造的示例。

### dINNE-Quant-Introduction-To-Quantization.pdf

该文档主要包含以下内容：

- dINNE 支持 4-bit (int4/uint4), 8-bit (int8/uint8) 的对称和非对称量化，混合精度 (int4/uint4, int8/uint8, float16, float32) 量化，Per-Channel 的方式量化，可指定权值/激活量化值域范围。
- 量化原理。
- 量化流程，包括加载模型，插入模拟量化节点，权值/激活量化值域范围统计，保存统计模型，模型转换，保存模型。
- 附录提供了量化后的模型中可能包含的量化相关的自定义的算子。

### dINNE-Quant-Quantization-Tutorial-With-ResNet-V1-50.pdf

该文档以图像分类模型 ResNet-V1-50 为例，说明了如何使用 TensorFlow 深度学习框架和量化工具，快速量化浮点网络模型。该示例包含步骤：导入依赖，加载/处理数据样本，加载模型，插入模拟量化节点，量化值域范围统计，模型转换，保存模型。

### dINNE-Quant-TU-Operator.pdf

该文档描述了张量运算硬件加速单元 (Tensor Unit, TU) 用于加速神经网络中常见的计算密集型任务，主要包含以下内容：

- Hamming 软件栈当前支持的算子类型 (如预处理，矩阵乘法，卷积，张量元素运算，激活，池化，张量变换)，各算子类型包含的算子 (如 Pad，MatMul，Conv2D，ElementWiseAdd，Relu6，AvgPool，Transpose 等)，各算子的输入/输出数据类型和输入/输出/权重数据格式，及各算子的特性。
- TU 算子模式。
- TU 算子编程约束。
- 附录描述了如何计算权重膨胀后的大小，和计算数据/权重对齐后的大小。

### dINNE-TVM-Quantization.pdf

- Quantization operators：为支持混合精度模型，dINNE 引入了一组支持精度为 uint8 量化计算的 quantization operators。而 `dl.dequantize` 可以将量化后的 operator 和正常的 operator 结合在一起。该章节详细介绍了如何使用 quantization ops API。
- 量化：dINNE TVM 提供了一个不依赖 Goldwasser AI 加速卡的自动量化工具包 (quantization toolkit)，可将一个浮点模型 (float model) 量化，并可用 dINNE 编译量化后的模型。该章节详细介绍了如何使用 quantization API。

## dITVM Relay IR 子图划分操作指导.pdf

主要介绍在Relay IR中加入子图划分信息的操作方法。

- **Relay IR**：是Apache TVM项目开发的中间表示，用于表示模型、优化模型和高效部署模型。
- **子图划分**：在Relay IR上插入表示子图边界的表达式，用来确定子图的区域信息，划分出子图。

有两种方式可以在Relay IR中加入子图划分信息：

1. 使用Relay API构造计算图时，可以直接使用子图标注API加入子图划分信息。
2. 从原始模型（ONNX）导入计算图时，可以指定一个子图标注配置文件。

## dITVM Frontend Operator Coverage

描述了dITVM对各个框架的算子支持情况，包括支持的框架版本、数量和具体的算子列表。

## dITVM Relay Core Tensor Operators

描述了登临 dITVM Relay 核心功能支持的张量操作符。

### 1.3.4 dIVID

dIVID 提供了一组视频硬件编解码功能的 APIs。dIVID 相关文档如下：

#### dIVID-Programming-Guide.pdf

该文档介绍了视频编解码引擎，工作流程，及如何使用 dIVID APIs 编程 VPU。

- 概述：登临 Goldwasser AI 加速卡内置的多个视频编解码引擎 (简称 VPU)，提供了适用于多种视频标准的完全硬件加速的解码和编码能力。解码器引擎只能解码。编码器引擎只能编码。多个 VPU 可以并行编解码，一个 VPU 可以同时进行多个解码或编码工作。dIVID APIs 用于 VPU 编程。

输入和输出数据被放入视频 buffer。VPU 从输入 buffer 获取输入数据，并将处理后的数据放到输出 buffer。对于编码任务，输入数据为 YUV 或 RGB 帧，输出数据为压缩视频流。对于解码任务，输入数据为压缩视频流，输出数据为 YUV 帧。这些特定的视频 buffer 可以通过一组专用的 dIVID APIs 进行管理。

编码或解码时需要处理各种事件。应用程序必须定义回调函数来处理这些事件。某些回调函数可以保持未定义状态，未定义回调函数的事件就会使用默认方式处理相应的事件。

- 编码和解码的流程类似：
  1. 获取编码或解码设备。
  2. 在该编码或解码设备上创建并激活会话。
  3. (仅适用于编码) 如有需要，设置编码配置。
  4. 准备输入和输出 buffer。
  5. 运行会话。
  6. 在无限循环中运行流程事件处理函数。
  7. 在回调中处理事件。
  8. 处理完所有输入数据后退出流程事件循环。
  9. 清理。
- 使用 dIVID APIs：



1. `dlvidGetDevice()` 获取设备 `DL_VID_DEVICE`。
2. 一个 VPU 可以同时处理多个视频流。为了保护视频流的处理不受另一个视频流的影响，每个视频流需要创建一个视频会话。每个视频会话提供一个上下文来编码或解码单个视频流。它是一个逻辑沙箱 (logical sandbox)，可以对流进行编码或解码并保护其免受其他流的影响。
  1. `dlvidCreateSession()` 创建会话。一个 `DL_VID_DEVICE` 上可以创建并运行多个会话。会话中的 `codecFormat` 必须是 VPU 支持的格式。  
回调和会话绑定。当需要处理某些编解码器事件时会调用回调，例如 VPU 工作状态改变、输出 buffer 准备就绪、输入 buffer 耗尽。如果不需要处理该事件，回调函数可以为 null。
  2. `dlvidActivateSession()` 激活会话。运行会话前必须激活。激活会话时会将 VPU 置于准备运行的状态。
  3. `dlvidRunSession()` 运行会话。VPU 开始工作。
  4. `dlvidStopSession()` 停止会话。某些比特流帧大小可能会改变，此时需要停止会话，然后刷新旧的 buffer，将新尺寸的 buffer 发送到 VPU，然后运行会话以继续解码新尺寸的帧。
3. 视频 buffer。buffer 作为输入或输出在 dVID 和 VPU 之间传递。buffer 包含视频流或帧数据以及数据的描述。对于输出 buffer，VPU 输出格式由 `buffer -> format` 设置。

VPU 上有一个输入 buffer 队列和一个输出 buffer 队列用于会话。输入 buffer 和空的输出 buffer 被发送到输入队列和输出队列。VPU 将使用 buffer 作为输入和输出。当处理完输入 buffer 数据时，输入回调将通知用户用新输入的数据填充 buffer。当输出 buffer 数据准备好时，输出回调将通知用户获取输出数据。VPU 无需等待即可处理队列中的下一个 buffer。如果没有更多可用的输入或输出 buffer，VPU 将停止编解码器并等待 buffer。因此，缺少 buffer 可能会影响性能。

1. 创建或销毁 buffer。

`dlvidCreateBuffer()` 创建一个 buffer。buffer 类型、帧高度、帧宽度等 buffer 信息在 `DL_VID_BUFFER_CREATE_INFO` 中。大多数 buffer 信息在创建后无法更改。

`dlvidDestroyBuffer()` 销毁一个 buffer。

2. 分配或释放 buffer memory。因为 VPU 无法使用 host memory，所以 buffer 存储器在 device memory。

`dlvidAllocateBufferMemory()` 为 buffer 分配 device memory。

`dlvidFreeBufferMemory()` 释放 buffer 占用的 device memory。在销毁 buffer 之前必须释放 device memory。

3. buffer 上的数据交换。输入数据需要复制到在 device memory 上的 buffer data memory 中，并且输出数据也需要从 buffer 中复制。

`cudaMemcpy()` 从 buffer 复制或向 buffer 复制数据。

`dlvidGetBufferMemoryPointer()` 为 `cudaMemcpy()` 获取 CUDA 指针。

当从 buffer 复制数据时，`dlvidGetBufferFilledLength()` 获取数据的尺寸。

将数据复制到 buffer 以后，`dlvidSetBufferFilledLength()` 设置数据的尺寸。

4. 为 VPU 注册或注销 buffer。buffer 必须在发送到 VPU 之前注册到 VPU。注册让 VPU 知道如何使用 buffer 并进行内存映射。每个 buffer 只需要注册一次。通常我们注册足够数量的 buffer 并重用它们直到编解码任务完成。例如注册 2 个输入 buffer 并将它们发送到 VPU。输入数据处理完毕后，VPU 将逐一返回用过的输入 buffer。用新的输入数据填充返回的输入 buffer 并再次发送到 VPU。

`dlvidRegisterBuffer()` 注册 buffer 到 VPU。

`dlvidUnregisterBuffer()` 注销 buffer。

5. 发送 buffer 到 VPU。buffer 只能在注册后才能发送到 VPU，否则会导致错误。

`dlvidEmptyBuffer()` 发送输入 buffer 到 VPU。

`dlvidFillBuffer()` 发送输出 buffer 到 VPU。

4. 回调。回调用于处理一些编解码事件，如输出 buffer 准备就绪，输入 buffer 处理完毕等。回调提供的功能很多，例如，`RpcPrint`: VPU 请求打印信息；`Error`: VPU 请求报告一个硬件错误，`StateChanged`: 告知会话状态改变。更多的回调功能，请阅读文档 **dIVID-Programming-Guide.pdf**。

5. 事件循环。

`dlvidProcessSessionEvent()` 在无限循环中开始事件处理循环。回调会因为相应的事件被调用。

### **dIVID-API.pdf**

dIVID-API.pdf 是 dIVID APIs 的参考文档。

### **1.3.5 dIOCL**

dIOCL 提供标准的 OpenCL API，请参考OpenCL官方文档。

### **1.3.6 dIML**

dIML 提供一组监控和管理登临硬件设备的 API。

#### **dIML-API.pdf**

dIML-API.pdf 是 dIML APIs 的参考文档。

### **1.3.7 dISS**

dISS 是登临系统软件包，包括GPU容器应用、系统监测、推理服务器、GPU虚拟化等软件包。

### **1.3.8 dIBLAS**

dIBLAS 提供登临平台上的BLAS ( Basic Linear Algebra Subprograms ) 接口，请参考 dIBLAS-Introduction.pdf 文档。

#### **dIBLAS-Introduction.pdf**

dIBLAS-Introduction.pdf 是 dIBLAS 的使用参考文档。

### **dIBLASLt-Introduction.pdf**

介绍了dIBLASLt支持的算子列表。

dIBLASLt是一个新的轻量级库，专门用于GEgeneral Matrix-to-Matrix Multiply ( GEMM ) 操作，具有新的灵活API。这个新库在矩阵数据布局、输入类型、计算类型以及通过参数选择算法实现方面具有非常好的灵活性。

### **1.3.9 dISOLVER**

dISOLVER 支持CUDA 11.7 cuSOLVER的部分求解功能API，请参考 dISOLVER 使用说明.pdf 文档。

#### **dISOLVER 使用说明.pdf**

dISOLVER 使用说明.pdf 是 dISOLVER 的使用参考文档。

## 2. 使用 dINNE

### 2.1 如何整合 dINNE

dINNE 可以在深度学习模型的训练，方案开发和方案部署三个阶段进行整合：

- 模型训练阶段

在模型训练阶段，dINNE 通过 Plugin 的方式与用户的框架进行整合。

- 方案开发阶段

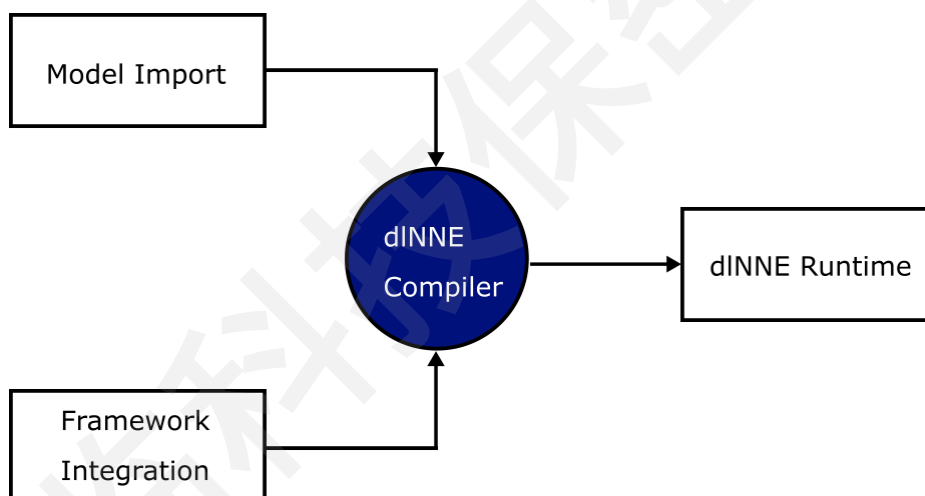
在方案开发阶段，用户可以将 dINNE 中的 Graph Compiler 集成到自己的开发系统中，并使用 dINNE Runtime 对结果进行验证。

- 方案部署阶段

在方案部署阶段，用户可以将 dINNE Runtime 集成到自己的应用里。

#### 2.1.1 工作流

工作流如下图所示：



#### 2.1.2 集成编译器

用户可以在模型训练阶段和方案开发阶段集成 dINNE Compiler。

##### 模型导入

dINNE 提供以下方式导入静态模型文件（.pb, .onnx）：

- 使用命令行工具将用户模型文件编译为登临运行时格式 (需要重命名)。
- 使用 C++ / Python API 将模型文件编译为登临运行时格式。
- 使用 converter 将静态模型文件编译为登临模型格式，再使用 C++/Python API 将其编译为登临运行时格式。

##### Framework 整合

通过使用 TensorFlow dINNE Plugin，用户可以在 TensorFlow 中自动使用 dINNE 实现子图的加速。

### 2.1.3 集成 Runtime

用户可以在模型训练，方案开发和方案部署三个阶段使用 dINNE Runtime 提供的 C/C++ 或 Python API 执行已编译好的模型。

#### Framework 整合

TensorFlow dINNE Plugin 会使用 dINNE runtime 来执行加速。

#### 方案开发阶段的整合

在方案开发阶段，dINNE Runtime 可以用于 dINNE Compiler 结果的验证。

#### 方案部署阶段的整合

在方案部署阶段，dINNE Runtime 直接作为部署方案的一部分来执行模型推理任务。

## 2.2 如何量化模型

### 通过 TensorFlow Quantization Plugin 进行 Quantization Aware Training。

dINNE 可以编译 TensorFlow Quantization Plugin 输出的模型。请参考 dINNE Quantization Toolkit 相关文档《dINNE-TVM-Quantization》，《dINNE-Quant-Introduction\_To\_Quantization》，《dINNE-Quant-Quantization\_Tutorial\_With\_ResNet-V1-50》，《dINNE-Quant-TU-Operator》。

## 2.3 使用 C++ API

详情请参考 dINNE C++ API 相关文档 *dINNE-Developer-Guide.pdf* 的内容。

## 2.4 使用 Python API

详情请参考 dINNE Python API 相关文档中的内容。

## 2.5 使用 Custom op

用户可以在 dINNE 中注册自己实现的 Custom op，这个 op 对应的 Kernel 可以用 TVM 标准方式实现，也可以用 dICU 实现 (即标准 CUDA 编程方式)。

### 2.5.1 实现 Custom Relay op

#### 在 C++ 中实现 Custom Relay op

1. 注册一个名为 `custom_op` 的 op，并实现相应的 attribute。

```
RELAY_REGISTER_OP("custom_op")
```

2. 注册一个构造函数，构造函数的名字格式需要符合 `relay.op._make.${name}`。

```
TVM_REGISTER_GLOBAL("relay.op._make.custom_op")
```

3. 将代码编译为一个 so。

### 在 Python 中注册 Frontend 转换函数

dINNE 提供一个 decorator 将一个转换函数注册进 Frontend。

```
@op.register_frontend_converter("TensorFlow", "OpName")
def converter(inputs, attrs, params, mod):
```

## 2.5.2 为 custom op 注册 custom kernel

如果 custom op 使用了 dICU 实现的 kernel，那么该 kernel 也必须注册到 dINNE 中。请参考 dINNE 相关 API 文档 [dINNE-API.pdf](#)和[dINNE-Build-Modulator-API.pdf](#)。

## 2.5.3 将定义好的 custom op 动态导入 dINNE

dINNE 提供了一个 Python API 用来导入编译好的 so。导入之后，dINNE 将产生一个名为 `dlnne.op.custom_op` 的构造函数。

```
dlnne.op.load_op_library(%{DIR_OF_SO}/libcustom_op.so)
```

### 3. 使用 dICU

详情请参考文档 *dICU-Programming-Guide.pdf* , *dICU-Extended-API.pdf*, *dICU-Extended-Math-API.pdf* , *Denglin Hamming V2 Texture Feature Guide*。

登临科技保密材料

## 4. 使用命令行工具dlsmi

DengLin System Management Interface (dlsmi) 提供了监控和管理 Goldwasser AI 加速卡的功能。

dlsmi 提供的主要选项如下：

- `list` 选项：列出系统中的 GPU 和被屏蔽的 GPU。
- `summary` 选项：显示系统中的 GPU 的综合信息。
- `query` 选项：显示系统中的 GPU 的详细信息，如 MEMORY，UTILIZATION，ECC，TEMPERATURE，POWER，CLOCK，COMPUTE，PIDS，PERFORMANCE，SUPPORTED\_CLOCKS，PAGE\_RETIREMENT，ACCOUNTING，ENCODER\_STATS 等。
- `mig` 选项：用于管理 MIG 模式。

文档 **DengLin-System-Management-Interface-User-Guide.pdf** 详细描述了各选项，返回值，query 选项返回的 GPU 属性和使用样例。