寒武纪 BANG C 性能调优指南

版本 *2.15.0*

1 月 28 日

Cambricon 寒武纪

目录

插图目录

1 版权声明

免责声明

中科寒武纪科技股份有限公司（下称“寒武纪”）不代表、担保（明示、暗示或法定的）或保证本文件所含信息，并明示放弃对可销售性、所有权、不侵犯知识产权或特定目的适用性做出任何和所有暗示担保，且寒武纪不承担因应用或使用任何产品或服务而产生的任何责任。寒武纪不应对因下列原因产生的任何违约、损害赔偿、成本或问题承担任何责任：（1）使用寒武纪产品的任何方式违背本指南；或（2）客户产品设计。

责任限制

在任何情况下，寒武纪都不对因使用或无法使用本指南而导致的任何损害（包括但不限于利润损失、业务中断和信息损失等损害）承担责任，即便寒武纪已被告知可能遭受该等损害。尽管客户可能因任何理由遭受任何损害，根据寒武纪的产品销售条款与条件，寒武纪为本指南所述产品对客户承担的总共和累计责任应受到限制。

信息准确性

本文件提供的信息属于寒武纪所有，且寒武纪保留不经通知随时对本文件信息或对任何产品和服务做出任何更改的权利。本指南所含信息和本指南所引用寒武纪文档的所有其他信息均“按原样”提供。寒武纪不担保信息、文本、图案、链接或本指南内所含其他项目的准确性或完整性。寒武纪可不经通知随时对本指南或本指南所述产品做出更改，但不承诺更新本指南。

本指南列出的性能测试和等级要使用特定芯片或计算机系统或组件来测量。经该等测试，本指南所示结果反映了寒武纪产品的大概性能。系统硬件或软件设计或配置的任何不同会影响实际性能。如上所述，寒武纪不代表、担保或保证本指南所述产品将适用于任何特定用途。寒武纪不代表或担保测试每种产品的所有参数。客户全权承担确保产品适合并适用于客户计划的应用以及对应用程序进行必要测试的责任，以期避免应用程序或产品的默认情况。

客户产品设计的脆弱性会影响寒武纪产品的质量和可靠性并导致超出本指南范围的额外或不同的情况和/或要求。

知识产权通知

寒武纪和寒武纪的标志是中科寒武纪科技股份有限公司在中国和其他国家的商标和/或注册商标。其他公司和产品名称应为与其关联的各自公司的商标。

式复制、重制、修改、出版、上传、发布、传输或分发本指南。除了客户使用本指南信息和产品的权利，根据本指南，寒武纪不授予其他任何明示或暗示的权利或许可。未免疑义，寒武纪不根据任何专利、版权、商标、商业秘密或任何其他寒武纪的知识产权或所有权对客户授予任何（明示或暗示的）权利或许可。

- 版权声明
- © 2021 中科寒武纪科技股份有限公司保留一切权利。

2 前言

2.1 版本记录

表 2.1: 版本记录表

文档名称	寒武纪 BANG C 性能调优指南
版本号	V 2.15.0
作者	Cambricon
修改日期	1 月 28 日

2.2 反馈

如果您发现软件 CNCC 有任何错误，请发送电子邮件到 compiler@cambricon.com，并请同时提供：

- 导致错误的源码文件；
- CNCC 的版本号；
- 其他使用过程中可能导致错误的环境信息；
- 问题的简要说明。

欢迎您对需要改进之处提出建议。

3 概述

3.1 BANG C 应用场景

BANG C 是由寒武纪公司为 MLU 硬件打造的编程语言，用户可以使用 BANG C 做通用计算和机器学习高性能计算。它提供高效的编程接口来充分利用 MLU 的硬件特性。

3.2 性能调优简介

BANG C 的性能调优的主要策略是充分利用硬件资源和软件编译优化，至少包括五个方面：

- 对访存进行优化以充分利用片上存储。
- 对计算逻辑进行优化以充分削减计算量，充分利用张量运算器。
- 充分利用多核并行（计算任务拆分）以及提供的循环自动流水来提升并行度以及隐藏访存延迟。
- 充分利用编译器提供的各种编译优化选项。
- 利用低精度运算提升计算性能并补偿由此带来的精度损失。

Cambricon
寒　武　纪

4 访存优化

对于 BANG C 的任务来说，NRAM，WRAM，LDRAM 都是私有数据。当一个任务执行完成之后，所有的私有数据都变为不可用。同一个 Kernel 任务中 Cluster 间和 Cluster 内部的共享数据可以存放在 SRAM 中。Kernel 任务之间共享的数据存放在 GDRAM 中，GDRAM 中的数据也可以被不同的 Kernel 共享使用。

4.1 变量声明

要注意由于 CNCC 的设定，当在变量声明前指定地址空间时，语义为将该变量同时声明为全局变量。当用户想在函数中定义局部变量的话，则不要在变量声明前指定地址空间。这样变量就默认在栈上，是真正的局部变量。

目前，CNCC 只支持声明静态可确定大小的数组，不能支持 malloc 或者 alloc 等动态空间分配。因此对于可变规模的计算，用户需要自己计算好空间大小，可能要对空间进行切分。

4.2 传参优化

对于 __mlu_func__ 或者 __mlu_device__ 的函数，推荐用户采用引用的参数传递方式，示例如下：

```
__mlu_func__ void myFunc(int& a, half& b) {
  ......
}
```

4.3 算子融合

当用 BANG C 写整个网络或者应用时，可以通过将多个算子进行融合的方式减少启动 Kernel 的次数，同时算子的融合也可以带来访存的减少，中间结果可以不需要存储到片外再加载回片上，以此达到性能提高。

融合前，scale 和 relu 的函数示例：

```
#define N 1024
__mlu_entry__ void scale(half* dst, half* src, half scale) {
```

```
  __nram__ half dst_nram[N];
  __memcpy(dst_nram, src, N * sizeof(half), GDRAM2NRAM);
  __bang_mul_const(dst_nram, dst_nram, scale, N);
  __memcpy(dst, dst_nram, N *sizeof(half), NRAM2GDRAM);
}

__mlu_entry__ void relu(half* dst, half* src) {
  __nram__ half dst_nram[N];
  __memcpy(dst_nram, src, N * sizeof(half), GDRAM2NRAM);
  __bang_active_relu(dst_nram, dst_nram, N);
  __memcpy(dst, dst_nram, N *sizeof(half), NRAM2GDRAM);
}
```

融合后，scale_relu 的函数示例：

```
#define N 1024
__mlu_entry__ void scale_relu(half* dst, half* src, half scale) {
  __nram__ half dst_nram[N];
  __memcpy(dst_nram, src, N * sizeof(half), GDRAM2NRAM);
  __bang_mul_const(dst_nram, dst_nram, scale, N);
  __bang_active_relu(dst_nram, dst_nram, N);
  __memcpy(dst, dst_nram, N *sizeof(half), NRAM2GDRAM);
}
```


4.4 GDRAM

GDRAM 容量大，访问速度慢，一般只用来在 Host 端和 MLU 端传输数据，而不直接参与大量运算。GDRAM 中的数据也可以被不同的 Kernel 共享使用。GDRAM 内存支持在 MLU 端和 Host 端管理。尤其在多线程时，由于 bank conflict 的影响，对于 GDRAM 的频繁访问更容易造成随机访问延迟，造成整体平均性能下降。

可以将 Host 端对于 GDRAM 的拷贝 cnrtMemcpy 合并，尽可能减少 cnrtMemcpy 的次数，提高性能。

由于 cnrtInvokeKernel 时有一次参数的 cnrtMemcpy，所以当追求高效时，要尽可能减少 cnrtInvokeKernel 次数。

4.5 LDRAM

由于 LDRAM 是片外存储空间，访存速度较慢，因此只有当片上存储空间不满足需求时，才会用 LDRAM 空间。

4.6 NRAM

NRAM 为片上存储空间，访存速度快，因此用户应尽可能使用 NRAM 空间完成运算。但是由于 NRAM 空间有限（如 MLU100，MLU270 约为 512KB），可能不能够直接满足运算需求，此时需要用户尽可能对 NRAM 手动进行空间重用使得能够满足运算需求。

下方给出 MLU100 上的 NRAM 空间重用的示例。观察重用前的代码，则可以看到，由于 MLU100 上 NRAM 大小限制为 512KB，声明的 NRAM 空间大小会超过 NRAM 可用空间，导致编译错误。但是假设 src0 和 src1 的活跃区间不重叠，则可以将代码改写成后者形式，那么可以编译通过并且两者都在 NRAM 高速空间上。

重用前:

```
#define LEN 250 * 1024
......
__nram__ half src0[LEN];
__nram__ half src1[LEN];
```


重用后:

```
#define LEN 250 * 1024
......
__nram__ half src0[LEN];
half* src1 = src0;
```

4.6.1 数据轮转

如果 NRAM 的空间不够放下全部数据，但是 SRAM 的空间可以放下数据，那么可以先把数据拆分加载到 SRAM，然后通过 Cluster 间通信的方式来获取全部的数据，称之为数据轮转。

这个过程需要精确计算 SRAM 的空间分配。

如图所示的例子中：Data 分成四块，可以分别放入到四个 SRAM 中，之后加载到 Cluster 内的四个 IPU Core 上（此时可以复制，也可以继续拆分，具体实现可灵活处理）。IPU Core 计算完毕后，SRAM 之间交换数据，交换的方法可以自行设计，在这个例子中是轮转。轮转完毕之后，SRAM0 中是 Data3, SRAM1 中是 Data0，SRAM2 中是 Data1，SRAM3 中是 Data2。然后 IPU Core 再取部分数据计算，三轮过后，Data0~3 四块数据，在所有 IPU Core 上都轮过一遍了。

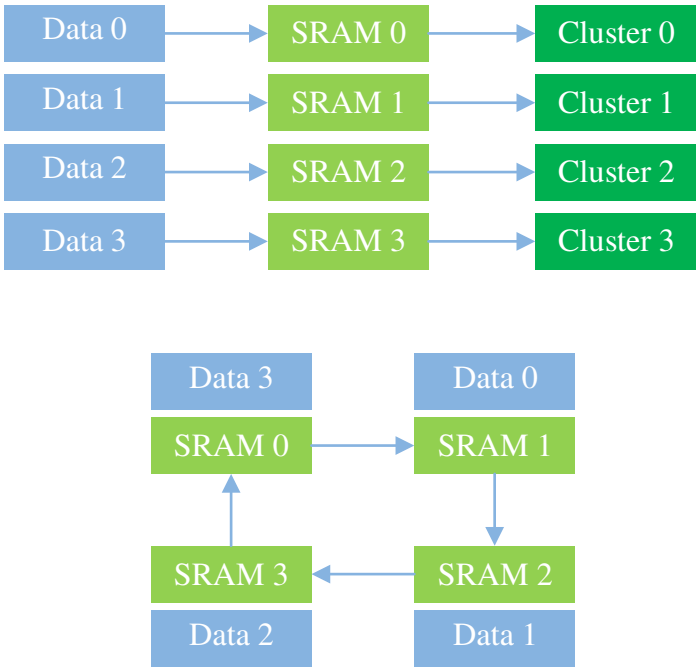

图 4.1: 数据流转示意图

4.7 WRAM

WRAM 也是片上高速访存空间，一般用于存储 __bang_conv 和 __bang_mlp 等指令的输入数据。

对于 DDR 带宽不足的情况，针对数据可以采用数据驻留的方法解决。即在单核 WRAM 空间足够放到全部数据时，一次将输入数据加载到 WRAM 上，然后一直存放在 WRAM 上直到计算结束。

当数据过多不能在 WRAM 空间放下的情况下，可以根据 WRAM 的空间大小对数据进行拆分，并将拆分出的数据分到每个核各自独有的 WRAM 上分别进行计算，再将多个核上的计算结果拼起来即为最终结果。

如果这样空间还是不够，可以考虑将数据进行拆分，先加载到 WRAM 上，剩下的数据放在 SRAM 上，在计算过程中去做一个数据的轮转来完成整个计算。这个过程需要精确计算 SRAM 上的空间分配。

4.8 SRAM

SRAM 是片上的共享内存，每个 Cluster 共享一个 SRAM，不同 Cluster 之间可以进行 SRAM 通信。它的主要作用是存储和缓存片上数据的中间结果，减少片外访存次数。SRAM 用 __mlu_shared__ 来修饰。

4.8.1 SRAM 使用示例

下面给出规约求和计算示例。

使用 SRAM 之前，规约求和之后的 val[0] 需要存储到 GDRAM，最后不同任务的值相加，得到输出值 out_sum。

```
#define CLUSTER_DIM 1
#define CORE_DIM 4
#define TASK_DIM (CLUSTER_DIM * CORE_DIM)
#define STRIDE 1024
__mlu_device__ float tmp[TASK_DIM];
__mlu_entry__ void reductionKernel(float* out_sum,
                                   float* arr,
                                   int num_elems) {
  __nram__ float sum[STRIDE];
  __nram__ float val[STRIDE];

  int task_stride = (num_elems + taskDim - 1) / taskDim;
  int start_offset = taskId * task_stride;
  int end_offset = (taskId + 1) * task_stride;
  if (end_offset > num_elems) {
    end_offset = num_elems;
  }
  __nramset_float(sum, STRIDE, 0.0f);
  for (int task_offset = start_offset; task_offset < end_offset;
       task_offset += STRIDE) {
    __memcpy(val, arr + task_offset, STRIDE * sizeof(float),
             GDRAM2NRAM);
    __bang_add(sum, sum, val, STRIDE);
  }
  for (int i = 0; i < 32; i ++) {
    __bang_reduce_sum(val, sum + i * 32, 32);
    sum[i] = val[0];
  }
  __bang_reduce_sum(val, sum, 32);
```



```
  //! Store the result into the GDRAM
  __memcpy(&tmp[taskId],&val[0],sizeof(float), NRAM2GDRAM);
  __sync_cluster();
  if(taskId == 0) {
    float sum = 0.0f;
    for(int i = 0; i < taskDim; i ++) {
      sum += tmp[i];
    }
    out_sum[0] = sum;
  }
}
```

使用 SRAM 之后，程序示例如下：规约求和之后的计算结果暂存在 SRAM 上，不用存放在 GDRAM，减少了访存的次数，减少了程序执行的时间，执行时间减少了约 18%。

```
#define CLUSTER_DIM 1
#define CORE_DIM 4
#define TASK_DIM (CLUSTER_DIM * CORE_DIM)
#define STRIDE 1024
__mlu_entry__ void reductionKernel(float* out_sum,
                                   float* arr,
                                   int num_elems) {
  __nram__ float sum[STRIDE];
  __nram__ float val[STRIDE];
  __mlu_shared__ float partial_sum[TASK_DIM];

  int task_stride = (num_elems + taskDim - 1) / taskDim;
  int start_offset = taskId * task_stride;
  int end_offset = (taskId + 1) * task_stride;
  if (end_offset > num_elems) end_offset = num_elems;

  __nramset_float(sum, STRIDE, 0.0f);

  for (int task_offset = start_offset; task_offset < end_offset;
     task_offset += STRIDE) {
    __memcpy(val, arr + task_offset, STRIDE * sizeof(float),
             GDRAM2NRAM);
    __bang_add(sum, sum, val, STRIDE);
  }

  for (int i = 0; i < 32; i ++) {
    __bang_reduce_sum(val, sum + i * 32, 32);
```

```
    sum[i] = val[0];
  }
  __bang_reduce_sum(val, sum, 32);

  //! Store the result into the shared memory
  partial_sum[taskId] = val[0];

  __sync_cluster();

  if(taskId == 0) {
    float sum = 0.0f;
    for(int i = 0; i < taskDim; i ++) {
      sum += partial_sum[i];
    }
    out_sum[0] = sum;
  }
}
```

4.8.2 SRAM 使用限制

下面说一下 SRAM 使用的限制。单核任务不能支持 SRAM 的使用，UNION1 任务和单核任务不支持 SRAM 通信，UNION2 以上任务才支持 SRAM 通信。

4.8.3 SRAM 通信库

SRAM 在 Cluster 之间可以通信，传递数据，CNSCCL 通信库就是寒武纪公司提供的 SRAM 通信库。CNSCCL 通信库是 Cambricon Neuware Shared-Memory Collective Communication Library 的缩写，具体使用方法可以参考《Cambricon BANG C Developer Guide》。

5 向量优化

对 BANG C 进行优化的手段中，最重要、最复杂的就是对计算逻辑进行优化以充分削减计算量，充分利用张量运算器。BANG C 提供了很多常用的向量指令，下面给出一些常用的向量化方法和相应的选择。从下面的例子可以看出，对于同一运算逻辑的向量化方法可能有多种组合，用户需要根据自己的需求选择其中一种。本文档中给出的向量化方法仅为其中的一种或几种，不代表为最优的向量化方法。

5.1 直接向量化方法

由于向量操作速度较快，建议用户尽可能使用向量操作代替标量运算。向量操作的优化速度与向量数据长度正相关。

5.1.1 向量相加

对于下方代码，在 MLU100 平台上，数据长度为 8192 时，性能提升约 40 倍。当数据长度为 65535 时，性能提升约 150 倍。

标量版：

```
#define CHANNELS  8192
......
for (int i = 0; i < CHANNELS; i++) {
    dst_nram[i] = src0_nram[i] + src1_nram[i];
}
```

向量版：

```
#define CHANNELS  8192
......
__bang_add(dst_nram, src0_nram, src1_nram, CHANNELS);
```


5.1.2 relu 激活

标量版：

```
#define CHANNELS  8192
......
for (int i = 0; i < CHANNELS; i++) {
    dst_nram[i] = src0_nram[i] > 0 ? src0_nram[i] : 0;
}
```

向量版：

```
#define CHANNELS  8192
......
__bang_active_relu(dst_nram, src0_nram, CHANNELS);
```

5.1.3 置零操作

当需要进行写零操作时，可以选择 __bang_write_zero 指令，或者 __nramset 指令。由于底层选择的指令实现不同，__bang_write_zero 指令的性能是要优于 __nramset 指令的，因此优先选择 __bang_write_zero 指令去实现置零操作。这里可以看出，可能对于某种计算逻辑，可以用不同的 BANG C 函数去实现，要根据实际情况，如数据大小等，去选择一种性能最优的方法。

5.2 间接向量化方法

由于 BANG C 提供的向量指令有限，所以除了一些显而易见的向量化方法之外，还有一些间接的，需要拼凑的向量化方法。下面提供几个例子仅供参考。需要注意的是，下例中提供的拼凑的向量化方法仅仅为该计算逻辑向量化的一种方式，并不保证为最优的向量化方法。

5.2.1 对位最大值

对位最大值 element-wise max 即为两个数组对应位置取最大值作为结果，其公式表示为 dst[i] = src0[i] > src1[i] ? src0[0] : src1[i]。

标量版：

```
#define LEN 1024
......
for (int i = 0; i < LEN, ++i) {
  dst[i] = src0[i] > src1[i] ? src0[0] : src1[i];
}
```


向量版：

```
#define LEN 1024
......
// max(a,b) ~ max(a - b, 0) + b
__bang_sub(dst, src0, src1, LEN);
__bang_active_relu(dst, dst, LEN);
__bang_add(dst, dst, src1, LEN);
```

5.2.2 向量加常数

由于 BANG C 只提供了 __bang_mul_const 指令，但是会经常用到数组的每个元素都加上一个常数的操作。而 BANG C 又未提供 __bang_add_const 接口，因此可以用 __nramset 和 __bang_cycle_add 拼凑成相应的语义。

标量版：

```
#define LEN 1024
......
for (int i = 0; i < LEN, ++i) {
  dst[i] = src[i] + value;
}
```


向量版：

```
#define LEN 1024
......
__nram__ half tmp[64];
__nramset(tmp, 64, value);
__bang_cycle_add(dst, src, tmp, LEN, 64);
```

5.2.3 向量 split

标量版：

```
#define LEN 256
......
for (int i = 0; i < LEN; ++i) {
  dst0[i] = src[2 * i];
  dst1[i] = src[2 * i + 1];
}
```


向量版：

```
#define LEN 512
……
__nram__ half mask0[LEN] = {1, 0, 1, 0, ……, 1, 0, 1, 0};
__nram__ half mask1[LEN] = {0, 1, 0, 1, ……, 0, 1, 0, 1};
__bang_maskmove(dst0, src, mask0, LEN);
__bang_maskmove(dst1, src, mask1, LEN);
```

5.2.4 gather-scatter 操作

gather-scatter 为一种稀疏线性代数中常见的内存寻址操作。gather 用作索引读，scatter 用作索引写。例如要将 {0, 1，……，63} 扩充为 {0, 0, 0, 1, 1, 1, ……, 63, 63, 63}，而 BANG C 并未直接提供这样的接口，用户可以用 __bang_conv 指令进行实现。

$$[0,1,2,\cdots,64]*\begin{bmatrix}1&0&0&\cdots&0\\1&0&0&\cdots&0\\1&0&0&\cdots&0\\0&1&0&\cdots&0\\0&1&0&\cdots&0\\0&1&0&\cdots&0\\\cdots&&&&\\0&0&0&\cdots&1\end{bmatrix}$$

标量版：

```
#define N 64
……
for (int i = 0; i < N, ++i) {
  dst[i * 3] = src[i];
  dst[i * 3 + 1] = src[i];
  dst[i * 3 + 2] = src[i];
}
```

向量版：


```
#define N 64
......
__bang_conv(dst, src, weight, N, 1, 1, 1, 1, 1, 1, N * 3);
```

5.2.5 矩阵乘法

对于 C = A * B，其中 A 为 M * N 矩阵，B 为 N * P 矩阵，则 C 为 M * P 矩阵。用户可以用 __bang_mlp 去实现该矩阵乘，可以看到此时，是有一次 for 循环的。

标量版：

```
#define M 64
#define N 64
#define P 64
......
for (int i = 0; i < M, ++i) {
  __bang_mlp(output, input + i * N, bias, weight, N, P, 0);
}
```

用户也可以用 __bang_conv 去实现矩阵乘，此时是没有循环的。

向量版：


```
#define M 64
#define N 64
#define P 64
......
__bang_conv(output, input, weight, N, M, 1, 1, 1, 1, 1, P, 0);
```

而后者的效率在 MLU 上显然是高于前者的。

5.3 常数预处理

编译时可以确定的数据写为常量，以减少编译生成的指令条数，提升效率。

常数预处理前：

```
#define N 1024
__mlu_entry__ void scale_relu(half* dst, half* src, int size, half scale) {
  __nram__ half dst_nram[N];
  __memcpy(dst_nram, src, size * sizeof(half), GDRAM2NRAM);
  __bang_mul_const(dst_nram, dst_nram, scale, size);
```



```
  __bang_active_relu(dst_nram, dst_nram, size);
  __memcpy(dst, dst_nram, size *sizeof(half), NRAM2GDRAM);
}
```

假设已知 size 与 N 是相同的。常数预处理后：

```
#define N 1024
__mlu_entry__ void scale_relu(half* dst, half* src, half scale) {
  __nram__ half dst_nram[N];
  __memcpy(dst_nram, src, N * sizeof(half), GDRAM2NRAM);
  __bang_mul_const(dst_nram, dst_nram, scale, N);
  __bang_active_relu(dst_nram, dst_nram, N);
  __memcpy(dst, dst_nram, N * sizeof(half), NRAM2GDRAM);
}
```

对于一些可以提前计算的，不随输入数据变化而变化的一些数据，可以提前计算好传入，以减少计算量。

6 多核并行优化

MLU 为多核异构平台，用户可以使用其进行多核加速计算。在 MLU 的一个 Cluster 内部的不同 Core 之间，以及不同 Cluster 之间，可以进行数据并行并通过 SRAM 进行数据通信。

6.1 UNION 模式

MLU 为多核异构平台，用户可以使用其进行多核加速计算。下方给出使用多核并行进行两个数组相除操作的示例代码。在 MLU100 平台上，当使用 32 核，任务类型为 UNION4 时，与单核相比，性能提升约 10 倍。

BLOCK 版：

```
#define LEN 65536
__mlu_entry__ void kernel(half* dst, half* src1, half *src2) {
  __nram__ half src1_nram[LEN];
  __nram__ half src2_nram[LEN];
  __memcpy(src1_nram, src1, LEN * sizeof(half), GDRAM2NRAM);
  __memcpy(src2_nram, src2, LEN * sizeof(half), GDRAM2NRAM);
  for ( int i = 0; i < LEN; i++) {
    src2_nram[i] = src1_nram[i] / src2_nram[i];
  }
  __memcpy(dst, src2_nram, LEN * sizeof(half), NRAM2GDRAM);
}
```


UNION 版：

```
#define CORE_NUM 32
#define LEN 65536
#define PER_CORE_LEN  (LEN / CORE_NUM)
__mlu_entry__ void kernel(half* dst, half* src1, half *src2) {
  __nram__ half src1_nram[PER_CORE_LEN]; // use macro for constant
  __nram__ half src2_nram[PER_CORE_LEN]; // use macro for constant
  __memcpy(src1_nram, src1 + taskId * PER_CORE_LEN,
           PER_CORE_LEN * sizeof(half), GDRAM2NRAM);
```

```
  __memcpy(src2_nram, src2 + taskId * PER_CORE_LEN,
           PER_CORE_LEN * sizeof(half), GDRAM2NRAM);
  for ( int i = 0; i < PER_CORE_LEN; i++) {
      src2_nram[i] = src1_nram[i] / src2_nram[i];
  }
  __memcpy(dst + taskId * PER_CORE_LEN, src2_nram,
           PER_CORE_LEN * sizeof(half), NRAM2GDRAM);
}
```

6.2 增加核间并行度

现在的任务多以 Cluster 级任务为主，因此在编程过程中需要注意核间的并行度。例如，在 UNION1 任务中，Core0 的计算结果需要给 Core1 使用，首先要分析 Core0 和 Core1 上的数据依赖情况。对于没有数据依赖的计算过程可以提前计算，即当 Core0 在计算的时候，Core1 可以进行没有数据依赖的计算，掩盖一部分计算时间。

以下给出了 Core0 和 Core1 的一种增加核间并行度的示例。

原始版本：


```
if (coreid == 0) {
  __memcpy(nram_data, ldram_data, len * sizeof(half), LDRAM2NRAM);
  __bang_mul_const(nram_data, nram_data, 2, len);
  __memcpy(gdram_data, nram_data, len * sizeof(half), NRAM2GDRAM);
}
__sync_all();
if (coreid == 1) {
  __memcpy(nram_data2, ldram_data, len * sizeof(half), LDRAM2NRAM);
  __bang_max(nram_data2, nram_data2, len);
  __memcpy(nram_data3, gdram_data, len * sizeof(half), GDRAM2NRAM);
  __bang_mul_const(nram_data2, nram_data3, nram_data2[0], len);
  __memcpy(gdram_data, nram_data2, len * sizeof(half), NRAM2GDRAM);
}
```

优化后：

```
if (coreid == 0) {
  __memcpy(nram_data, ldram_data, len * sizeof(half), LDRAM2NRAM);
  __bang_mul_const(nram_data, nram_data, 2, len);
  __memcpy(gdram_data, nram_data, len * sizeof(half), NRAM2GDRAM);
}
if (coreid == 1) {
```



```
  __memcpy(nram_data2, ldram_data, len * sizeof(half), LDRAM2NRAM);
  __bang_max(nram_data2, nram_data2, len);
}
__sync_all();
if (coreid == 1) {
  __memcpy(nram_data3, gdram_data, len * sizeof(half), GDRAM2NRAM);
  __bang_mul_const(nram_data2, nram_data3, nram_data2[0], len);
  __memcpy(gdram_data, nram_data2, len * sizeof(half), NRAM2GDRAM);
}
```

可以看出，第二个 if 判断之内的计算为 Core1 中不依赖于 Core0 的部分，可以将其提升到 __sync_all 之前，将这一部分的计算时间与 Core0 的计算进行重叠。

6.3 Cluster 间流水

下面给出一个 Cluster 间流水的例子。Cluster 间的流水是任务上的流水，首先拆解任务，例如 Cluster0/1/2 完成任务 1，Cluster3 完成任务 2，在这里最好使用 SRAM 进行 Cluster 之间的数据交换，利用 SRAM 可以一定程度上缓解带宽的限制。示例代码如下：

```
if (clusterId == 0 || clusterId == 1 || clusterId == 2) {
  __memcpy(nram_data, ldram_data, len * sizeof(half), LDRAM2NRAM);
  __bang_mul(nram_data, nram_data, nram_w1, len);
  __memcpy(gram_data, nram_data, len * sizeof(half), NRAM2GRAM);
  __sync_cluster();
}
if (clusterId == 3) {
  __memcpy(nram_data, gdram_data, len * sizeof(half), GDRAM2NRAM);
  __bang_add(nram_data, nram_data, nram_w1, len);
  __memcpy(gdram_data, nram_data, len * sizeof(half), NRAM2GDRAM);
  __sync_cluster();
}
```


7 循环自动流水

当任务规模比较大，需要循环加载、计算和存储数据的时候，由于访存和计算可以并行执行，可以通过循环流水的方式用计算时间掩盖访存时间，或者相反。编译器提供了循环自动流水的优化，在满足一定条件下，会自动进行流水操作。

7.1 循环自动流水简介

循环自动流水分为三个部分：Prolog, Kernel 和 Epilog。Prolog 和 Epilog 为循环流水开始和结束阶段，Kernel 为循环部分，在此部分的计算指令和访存指令可以达到很好的并行，如图所示，其中 L，C，S 分别表示加载（Load）, 计算（Compute）和存储（Store)。

图 7.1: 循环自动流水

7.2 循环自动流水步骤

7.2.1 编写 MLU 程序

用户的内存申请和使用可以分为连续和非连续两种形式。连续形式下，用户可以申明一块大的 NRAM 区域，所有的访存和计算操作通过该区域内的偏移来实现。非连续形式下，用户可以申明不同的 NRAM 变量进行操作，这种形式思路清晰但是可能会造成内存的浪费。循环自动流水对这两种形式都支持，后续例子均采用连续形式。

连续形式：

```
#define BUFFER_SIZE 523008
#define ALL_DATA_NUM BUFFER_SIZE/sizeof(float)
#define DATA_NUM ALL_DATA_NUM/2
__mlu_entry__ void kernel(uint32_t size, float* c, float* a, float* b) {
  __nram__ float tmp[ALL_DATA_NUM];
```

```
  for (int i = 0; i <= 100; i++) {
    __memcpy(tmp, a + i * DATA_NUM, DATA_NUM * sizeof(float), GDRAM2NRAM);
    __memcpy(tmp + DATA_NUM, b + i * DATA_NUM, DATA_NUM * sizeof(float), GDRAM2NRAM);
    __bang_add(tmp, tmp, tmp + DATA_NUM, DATA_NUM);
    __memcpy(c + i * DATA_NUM, tmp, DATA_NUM * sizeof(float), NRAM2GDRAM);
  }
}
```

非连续形式：

```
#define BUFFER_SIZE 512
#define DATA_NUM BUFFER_SIZE/sizeof(float)
__mlu_entry__ void kernel(uint32_t size, float* c, float* a, float* b) {
  __nram__ float tmp1[DATA_NUM];
  __nram__ float tmp2[DATA_NUM];
  __nram__ float tmp3[DATA_NUM];
  for (int i = 0; i < 100; i++) {
    __memcpy(tmp1, a + i * DATA_NUM, DATA_NUM * sizeof(float), GDRAM2NRAM);
    __memcpy(tmp2, b + i * DATA_NUM, DATA_NUM * sizeof(float), GDRAM2NRAM);
    __bang_add(tmp3, tmp1, tmp2, DATA_NUM);
    __memcpy(c + i * DATA_NUM, tmp3, DATA_NUM * sizeof(float), NRAM2GDRAM);
  }
}
```


7.2.2 检查是否符合流水要求

现阶段自动流水有如下要求：

- 为了防止循环次数过低时被循环展开，请在 CNCC 编译时加入-fno-unroll-loops 选项，编译时采用至少 O2 选项；
- 尽量按照 Load、Compute，Store（以下简称为 LCS 模式）的顺序排布需要自动流水的指令，否则有可能不会流水；
- LCS 模式时使用严格小于一半 NRAM 大小的 NRAM 空间；
- 循环内不要有分支；
- 访问的 NRAM 变量寻址时不要和循环变量（比如 i) 有关；
- 循环步长需要是常数。

7.2.3 按照要求改写程序

假设需要支持可变规模的 add 算子，原.mlu 程序如下：

```
#define BUFFER_SIZE 523008
#define ALL_DATA_NUM BUFFER_SIZE/sizeof(float)
#define DATA_NUM ALL_DATA_NUM/2
__mlu_entry__ void kernel(uint32_t size, float* c, float* a, float* b) {
  int repeat = size / DATA_NUM;
  int rem = size % DATA_NUM;
  __nram__ float tmp[ALL_DATA_NUM];
  for (int i = 0; i < repeat; i++) {
    if (i == repeat - 1) {
      __memcpy(tmp, a + i * DATA_NUM, (rem + DATA_NUM) * sizeof(float), GDRAM2NRAM);
      __memcpy(tmp + DATA_NUM, b + i * DATA_NUM, (rem + DATA_NUM) * sizeof(float), GDRAM2NRAM);
      __bang_add(tmp, tmp, tmp + DATA_NUM, DATA_NUM);
      __memcpy(c + i * DATA_NUM, tmp, (rem + DATA_NUM) * sizeof(float), NRAM2GDRAM);
    } else {
      __memcpy(tmp, a + i * DATA_NUM, DATA_NUM * sizeof(float), GDRAM2NRAM);
      __memcpy(tmp + DATA_NUM, b + i * DATA_NUM, DATA_NUM * sizeof(float), GDRAM2NRAM);
      __bang_add(tmp, tmp, tmp + DATA_NUM, DATA_NUM);
      __memcpy(c + i * DATA_NUM, tmp, DATA_NUM * sizeof(float), NRAM2GDRAM);
    }
  }
}
```


使用一半以下的 NRAM 空间作为 BUFFER，需要考虑去除编译器保留空间以及栈在 NRAM 上时的栈空间，可以不断地尝试更改数值直到流水成功为止：

```
#define BUFFER_SIZE 523008/2
```

消除循环内部分支，可以将分支提出循环外：

```
for (int i = 0; i < repeat; i++) {
  __memcpy(tmp, a + i * DATA_NUM, DATA_NUM * sizeof(float), GDRAM2NRAM);
  __memcpy(tmp + DATA_NUM, b + i * DATA_NUM, DATA_NUM * sizeof(float), GDRAM2NRAM);
  __bang_add(tmp, tmp, tmp + DATA_NUM, DATA_NUM);
  __memcpy(c + i * DATA_NUM, tmp, DATA_NUM * sizeof(float), NRAM2GDRAM);
}
if (rem > 0) {
  __memcpy(tmp, a + repeat * DATA_NUM, rem * sizeof(float), GDRAM2NRAM);
  __memcpy(tmp + DATA_NUM, b + repeat * DATA_NUM, rem * sizeof(float), GDRAM2NRAM);
  __bang_add(tmp, tmp, tmp + DATA_NUM, DATA_NUM);
```



```
  __memcpy(c + repeat * DATA_NUM, tmp, rem * sizeof(float), NRAM2GDRAM);
}
```

7.2.4 验证流水是否成功

CNCC 提供 O1，O2，O3 和 Os 优化选项，可以比较 O1、O2、O3 的性能，如果 O2 和 O3 差不多并且都明显好于 O1，通常流水成功。也可以调大 BUFFER 的大小，大于 NRAM 一半时，由于不能流水，性能反而会跌落，出现这种情况也能反映流水成功。

8 编译优化选项

8.1 优化级别

CNCC 提供 O1,O2,Os,O3 编译优化选项。在大部分情况下，用户可以使用 O2 优化。当开发者追求极致性能的时候，可以开启激进的 O3 优化。

8.2 –fmlu-fast-math

由于编译器不会默认做浮点运算的优化，所以当用户的程序中含有较多的浮点运算时，可以打开–fmlu-fast-math 编译选项，以提升性能。

Cambricon
寒　武　纪

9 精度补偿

由于半精度浮点数采用 2 字节存储，只适合用来存储对精度要求不高的数字，而不适合计算。与单精度浮点数相比，虽然只需要一半的存储空间和带宽，但是代价为精度和数值范围。

用浮点表示就一定会产生精度损失 epsilon。如果想表达一个 0.000488 的数字，在目前的硬件平台上是无法表示的，只会得到 1.0 这种值。可以通过下面方法计算 epsilon：

```
epsilon = 1.0;
while ((1.0 + 0.5 * epsilon) != 1.0) {
  epsilon = 0.5 * epsilon;
}
```

在 MLU100 上只支持半精度浮点，而不支持单精度浮点或双精度浮点。MLU270 虽然支持单精度浮点，但是单精度浮点的计算单元要少于半精度浮点，计算效率低。因此可能存在半精度浮点运算不满足运算精度的情况。

在 MLU 中精度损失主要表现为半精度浮点类型计算结果与单精度浮点类型计算结果不一致，甚至差别很大的情况。例如，half 的数值范围有限，为-65504-65504，当超出其范围时在 MLU 上做饱和处理，即划分到-65504 或 65504。以及进行运算时，由于 half 的可表示精度问题，导致计算结果出现较大误差，如累加，加乘等运算。常见的为大数吃小数的情况。比如 half 中 1000 + 0.5 = 1000, 这就是大数吃小数的情况。

以下以累加误差补偿为例介绍精度补偿方案。

9.1 累加误差补偿

假设数组内的数字范围相仿，在对数组所有数求和时，可以通过将数字两两相加的方法，来避免相加数之间差距过大，从而有效避免大数吃小数的现象出现。也可以采用 Kahan 求和算法来获得更为精确的结果。Kahan 求和是保存较大数与较小数相加过程中，较小数丢失的有效数字，然后将丢失的有效数字一次补偿到求和结果中。

以下是对数组求和精度补偿的示例。

补偿前:


```
#define LEN 1024
__mlu_func__ half sum (half* src) {
  __nram__ half result = 0;
  for (int i = 0; i < LEN; ++i) {
    result += src[i];
  }
  return result;
}
```

累加误差补偿共两种方法。

方法一两两相加：

```
#define LEN 1024
__mlu_func__ half sum (half* src) {
  for (int k = LEN / 2; k > 0; k /= 2) {
    for (int i = 0; i < k; ++i) {
      src[i] = src[2 * i] + src[2 * i + 1];
    }
  }
  return src[0];
}
```


方法二 Kahan 算法：

```
#define LEN 1024
__mlu_func__ half sum (half* src) {
  half result = 0;
  half y;
  half temp;
  half comp = 0;
  for (int i = 0; i < LEN; ++i) {
    y = src[i] - comp;
    temp = result + y;
    comp = (temp - result) - y;
    result = temp;
  }
  return result;
}
```

9.2 转数损失

在 MLU270 上，__bang_conv 和 __bang_mlp 指令只支持 fix 类型，因此需要进行转数，如从 float 或 half 转为 fix8，fix16 等类型。虽然 fix 的计算没有精度损失，但是此时转数时需要注意 pos 域的选取和相应带来的转数的精度损失。

10 FAQ

问题 1：barrier 和 sync_cluster 的区别?

sync_cluster 是对 barrier 的封装。

问题 2：建议手动排流水还是自动排流水?

编译器自动排流水对用户程序有很多限制，如用户使用的 NRAM 空间必须小于总空间的一半等限制，因此用户需要确认自己的程序是否被自动排流水。如果没有自动，则还需要手动排流水。

问题 3：结构体中的参数在内联函数中被频繁的取出使用，在代码内联后，编译过程中是否会对其读写行为进行优化?

对传值结构体参数的读操作可以被优化, 写操作不行。对传指针和传引用的读写都可以优化。

问题 4：是否建议使用指针?

不建议。原因如下：

- 指向多个地址空间的指针，编译器会生成一些额外的指令来保证访存空间的正确性，这可能会造成程序性能的下降。
- 编译器对于指针的分析通常比较复杂，为了保证正确性会放弃一些优化机会，从而可能会造成程序性能的下降。
- 指针使用不当容易出错，尤其是多级指针，错误很难定位。

问题 5：大量的小批量数据随机寻址运算是否影响性能?

会。大量的小批量数据随机寻址运算会导致性能降低。

