南开大学 Nankai University

南　　开　　大　　学

计　算　机　学　院

“先导杯”校内赛赛题实现报告

基于矩阵乘法优化实现的多层感知机性能优化

姓名：申健强

学号：2313119

专业：计算机科学与技术

指导教师：师建新 张金

2025 年 6 月 3 日

目录

一、 赛题背景

低轨（LEO）卫星网络因其低时延、高覆盖的优势，正成为未来全球广域网络服务的重要补充。目前，SpaceX、OneWeb 等公司已部署数千颗卫星，初步形成星座网络；我国星网工程也在加快推进，积极构建天地一体化信息网络。LEO 卫星网络具备动态拓扑、链路多变、频繁切换等特点，使其网络服务面临带宽波动性大、链路预测难等挑战。因此，提升服务质量的关键之一在于精准的网络带宽预测。借助机器学习模型，可实现对历史网络状态的深度建模与未来网络带宽的有效预测，但如何实现高效且实时的预测，要求对机器学习的计算过程进行深度优化。

机器学习过程的核心计算单元是矩阵乘法运算。在实际应用中，如何高效利用加速硬件（如曙光 DCU，英伟达 GPU 等）和并行计算算法完成大规模矩阵乘，成为智能计算系统设计的关键问题。

二、 赛题题面

(一) 基础题：智能矩阵乘法优化挑战

已知两个矩阵：矩阵 A(大小 $N \times M$)，矩阵 B(大小 $M \times P$)。

问题一：完成标准的矩阵乘算法，并支持浮点型输入，输出矩阵为 $C = A \times B$ 并对随机生成的浮点数矩阵输入（$N = 1024$，$M = 2048$，$P = 512$），验证输出是否正确。

问题二：采用至少两种方法加速以上矩阵运算，鼓励采用多种优化方法和混合优化方法；理论分析优化算法的性能提升，并可通过 hipprof、hipgdb、rocm - smi 等工具进行性能分析和检测。

(二) 进阶题 1：基于矩阵乘法的多层感知机（MLP）实现和性能优化挑战

基于矩阵乘法，实现 MLP 神经网络计算，可进行前向传播、批处理，要求使用 DCU 加速卡，以及矩阵乘法优化方法，并进行全面评测。输入、权重矩阵由随机生成的浮点数组成：输入层：一个大小为 B×I 的随机输入矩阵（B 是 batch size = 1024，I 是输入维度 = 10）。

以三层 MLP 为例，假设：输入层：1024×10 隐藏层：10×20（ReLU）输出层：20×5（无激活）

变量定义：X 是输入矩阵（Batch× 输入维度），W 是权重矩阵（输入维度 × 输出维度），B 是偏置向量。

第一层前向传播：$H_1 = ReLU(X@W_1 + B_1)$, 其中 X : [1024×10], W_1 : [10×20], B_1 : [1×20], H_1 : [1024×20]。

第二层前向传播：$Y = H_1@W_2 + B_2$，其中 $ReLU(x) = \begin{cases} x & if x > 0 \\ 0 & if x \leq 0 \end{cases}$

(三) 进阶题 2：基于 MLP 的低轨卫星网络带宽预测性能优化挑战

完成 MLP 网络设计，要求能够进行前向传播，反向传播和通过梯度下降方法训练，并实现准确的 LEO 卫星网络下行带宽预测，需使用 DCU 加速卡，并对训练和推理性能进行全面评测。输入：每次输入 $t_0, t_1, \ldots, t_N$ 时刻的网络带宽值（$N = 10$）。数据集：一维的带宽记录，每个数据对应一个时刻的带宽值（已上传到测试环境中），文件为 starlink_bw.json。

(四) 实验环境和代码

实验环境：

- DCU：曙光异构加速卡 1/16GB
- 线上测试平台：
- IDE：Vscode
- 编译器：hipcc

实验代码：在本报告中我们将展示算法思路和关键的代码片段，具体实现可以在https://github.com/sjq0098/MatSatMLP.git中查看。

三、 优化原理

(一) 矩阵乘法优化原理

矩阵乘法的原始算法（即三重循环）在计算上具有极高的算术强度（Arithmetic Intensity），其时间复杂度为

$$T_{\text{baseline}} = O(N \times M \times P)$$

并且总浮点运算量为

$$\text{FLOPs} \approx 2\,N\,M\,P.$$

然而，单纯地依赖这一算法往往无法在现代 CPU/GPU 架构上充分发挥硬件性能，瓶颈常常来自于内存层次结构的访问开销。为了提升性能，需要从以下几个数学原理角度进行优化：

1. 数据局部性与块划分（Blocking/Tiling）

空间局部性与时间局部性 在经典的三重循环算法中，内层循环访问形如 $A[i, m\cdot k+k]$ 和 $B[k, p\cdot j+j]$ 的内存地址，其中对于固定的 i 和 j，k 从 0 遍历到 $M-1$。此时对于矩阵 A，一行元素被连续访问，可以较好利用行优先存储（Row-major order）带来的空间局部性；但对于矩阵 B，访问模式为跳跃式读取，无法充分利用缓存的空间局部性，导致频繁的缓存失效率（Cache Miss）。

块划分的思想 若将矩阵分块，将矩阵 A 和 B 划分为若干个大小为 $b \times b$ 的子块，则可以将三个循环等价改写为对块（Block）的遍历，再在块内部进行常规的矩阵乘法。设块大小为 b，则有：

$$A = \begin{pmatrix} A_{11} & A_{12} & \cdots \\ A_{21} & A_{22} & \cdots \\ \vdots & \vdots & \ddots \end{pmatrix}, \quad B = \begin{pmatrix} B_{11} & B_{12} & \cdots \\ B_{21} & B_{22} & \cdots \\ \vdots & \vdots & \ddots \end{pmatrix}, \quad C = \begin{pmatrix} C_{11} & C_{12} & \cdots \\ C_{21} & C_{22} & \cdots \\ \vdots & \vdots & \ddots \end{pmatrix},$$

其中每个 A_{pq}, B_{pq}, C_{pq} 均为 $b \times b$ 大小的子矩阵。对应的乘法可写为：

$$C_{ij} \mathrel{+}= \sum_k A_{ik} \times B_{kj}, \quad \text{其中} A_{ik},\, B_{kj},\, C_{ij} \in \mathbb{R}^{b\times b}.$$

这样做能够使得每次对一个 $b \times b$ 子块进行运算时：1. 将 A_{ik} 子块和 B_{kj} 子块从主存或更低级缓存（如 L2/L3）加载到更高效的缓存（如 L1）。2. 在 L1 缓存中连续地对这两个子块进行 b^3 次浮点计算，直到用尽，再将结果写回。

其核心数学收益在于：

- **减少缓存缺失（Cache Miss）次数**传统三重循环中，每次 k 的迭代都会针对 B 进行跨行访问，而块化后，在读取 B_{kj} 的整个 $b \times b$ 子块后，可重用其中的所有元素 b 次，从而提升空间局部性。
- **提高缓存重用率（Reuse Factor）**假设矩阵维度较大，整行或整列无法全部容纳于 L1 缓存，则将 b 选取为与缓存容量相匹配的合适大小，使得一个块在一次加载后能被充分重复利用，降低内存带宽需求。

若令 b 为合理选取的块大小，则在 $N \times M \times P$ 的总体工作量中，块划分后内存访问的浮点数近似为:

$$\mathrm{Mem}_{\mathrm{blocked}} \approx \frac{N\,M + M\,P + N\,P}{b} \times b = N\,M + M\,P + N\,P,$$

相比未经分块时频繁的缓存失效数目大大减少。

2. 转置（Transpose）

访问模式重排 在基于内存行优先（Row-major）布局的环境下，直接访问 B 的 $B[k][j]$ 会导致跨行跳转，无法在一次 cache line 加载后复用多个元素。为此可以先将矩阵 B 转置为 $\widetilde{B}$，使得原本按列访问的模式转换为按行访问。设

$$B \in \mathbb{R}^{M \times P}, \qquad \widetilde{B} = B^T \in \mathbb{R}^{P \times M},$$

其元素对应关系为

$$\widetilde{B}[j][k] = B[k][j], \quad k = 0, \ldots, M-1,\ j = 0, \ldots, P-1.$$

则在计算内层循环时，可直接令

$$\sum_{k=0}^{M-1} A[i][k] \times B[k][j] \;\equiv\; \sum_{k=0}^{M-1} A[i][k] \times \widetilde{B}[j][k],$$

其中 $\widetilde{B}[j][k]$ 在内存中相邻存储。这样，对于固定的 i, j，k 的循环访问只涉及两行数据：A 的第 i 行和 $\widetilde{B}$ 的第 j 行，均为顺序访问，能够极大提高缓存命中率。转置操作本身的开销为 $O(MP)$，但当 $M, P \gg$ 缓存大小时，通过重排访问能显著降低内层循环的内存延迟，从而整体上减少运行时间。

3. SIMD 向量化（Single Instruction Multiple Data）

向量寄存器与批量浮点运算 现代处理器支持向量寄存器（如 AVX2 的 256-bit 寄存器），一次可以并行处理 v 个相同类型的数据（例如 $v = 4$ 个 double-precision）。在数学上，将标量乘加运算

$$c_j = \sum_{k=0}^{M-1} a_{i,k}\, b_{k,j} \quad \text{(标量模式)}$$

拓展为向量模式时，可在一次指令中完成

$$(c_j,\, c_{j+1},\, \ldots,\, c_{j+v-1}) \;=\; \sum_{k=0}^{M-1} a_{i,k} \times (b_{k,j},\, b_{k,j+1},\, \ldots,\, b_{k,j+v-1}),$$

其中 $(b_{k,j}, \ldots, b_{k,j+v-1})$ 为 v 个连续位置的元素。同样，$a_{i,k}$ 被广播（broadcast）到向量寄存器，赋值给每个分量。这样，每完成一次 k 的迭代，实际同时更新了 v 个不同的 j 下标对应的累加值，从而将原本的 M 次标量乘法变为 M 次向量乘法，整体浮点运算吞吐提高约 v 倍。

对齐与捕获内存局部性 为了最大化性能，需要保证内存访问的对齐（aligned load/store）或至少利用 `loadu` 进行无对齐访问，以减少对内存子块的额外开销。此外，SIMD 向量化常常与块划分结合，以保证 B 在连续内存上的 v 个元素能够被一次读入向量寄存器。

4. 多线程并行（Parallelism）

循环并行化 矩阵乘法天然具有 $\mathcal{O}(N \times P)$ 个独立输出元素，每个元素对应于一个向量点积。可通过对最外层的循环（例如迭代 i 或迭代 (i, j)）进行并行，令不同线程负责计算不同的行或行列组合。在 OpenMP 中可使用类似

```
#pragma omp parallel for collapse(2) schedule(static)
```

的指令，将二维循环 (i, j) 打平后在多个线程间静态分配任务，从而获得接近线性加速。

负载均衡与内存冲突 并行化时，还需关注线程间负载均衡：若将循环分配为等量的迭代次数，则可确保各线程工作量相近；此外，在写入结果矩阵时，避免多个线程同时写入同一缓存行，从而产生伪共享（false sharing）。通过合适的索引映射与块划分，可降低此类冲突。

5. GPU 上的块（Tiling）与共享内存（Shared Memory）

分层存储访存模型 GPU（如使用 HIP/DCU）拥有更深的存储层次：全局内存（Global Memory）、片上共享内存（Shared Memory）及寄存器（Registers）。全局内存带宽虽高，但延迟也大；而共享内存带宽更高、延迟更低，适合多次重用。

线程块与瓦片（Tile）策略 在 GPU 中，通常将二维输出矩阵 C 分为 $(T \times T)$ 大小的瓦片（Tile），每个线程块（block）负责计算一个瓦片中的所有元素。设：

$$T = 16, \quad \text{则瓦片大小为} 16 \times 16.$$

每个线程块中，先从全局内存加载对应的 A 子矩阵 $T \times T$ 块以及 B 子矩阵 $T \times T$ 块到共享内存（Shared Memory），然后在共享内存中对这两个小块执行 T^3 次标量乘累加运算。数学上，每个瓦片遍历 $\lceil M/T \rceil$ 个阶段，每阶段完成

$$T \times T \times T = T^3$$

次乘加操作。全部阶段结束后，得到该线程块负责区域的最终结果。

共享内存重用（Data Reuse） 假设 $A_{\text{tile}} \in \mathbb{R}^{T \times T}$ 与 $B_{\text{tile}} \in \mathbb{R}^{T \times T}$ 被加载后，可在共享内存中重复使用 T 次内层迭代，直到该瓦片完成对 $\sum_t A_{\text{tile}}^{(t)} \times B_{\text{tile}}^{(t)}$ 的汇总。相比直接从全局内存加载，使用共享内存能将对同一数据的访问次数从 $\lceil M/T \rceil$ 减少为仅一次加载并在共享内存中重复 T 次提取，从而显著降低全局内存带宽压力。

6. 算术强度与 Roofline 模型

算术强度（Arithmetic Intensity） 定义算术强度为

$$\text{AI} = \frac{\text{总浮点运算量 (FLOPs)}}{\text{总内存访问量 (Bytes)}}.$$

对于矩阵乘法，若仅按最简模型假设每个元素仅被读取一次、写入一次，则

$$\text{FLOPs} = 2\,N\,M\,P, \qquad \text{Bytes} \approx 8\,(N\,M + M\,P + N\,P).$$

因此

$$\mathrm{AI}_{\mathrm{ideal}} = \frac{2\,N\,M\,P}{8\,(N\,M + M\,P + N\,P)}.$$

当 N, M, P 相近且很大时，AI 会趋近于常数。但在真实场景中，由于缓存缺失、重复载入等因素导致的内存访问远超理想情况，从而降低了实际算术强度。通过块划分、转置及共享内存重用，可加大 AI，使得更多的浮点运算在数据被加载之后立即进行，从而更接近计算带宽限制，而非由内存带宽瓶颈所主导。

Roofline 模型 在 Roofline 模型中，性能上限取决于较低的两者：处理器峰值计算能力 $\times$ 软件实现效率，或内存带宽 $\times$ 算术强度。随着优化手段（块划分、转置、向量化、并行化、共享内存等）的引入，算术强度提高，计算性能逐渐从“带宽受限”向“计算受限”转移，最终可能达到硬件的顶点性能。

后续我们将会介绍上述方法的具体实现。

(二) MLP 前向传播原理

1. MLP 前向传播

设批量大小为 B，输入维度为 I，隐藏层维度为 H，输出维度为 O。定义：

$$X \in \mathbb{R}^{B\times I}, \quad W_1 \in \mathbb{R}^{I\times H}, \quad b_1 \in \mathbb{R}^{H}, \quad W_2 \in \mathbb{R}^{H\times O}, \quad b_2 \in \mathbb{R}^{O}.$$

前向传播分两步计算：

1. **第一层（线性变换 + 偏置 + ReLU）**
   对于第 b 个样本（$b = 1, \ldots, B$）和第 h 个隐藏元（$h = 1, \ldots, H$），

   $$H_{b,h} = \max\Big(0,\ \underbrace{\sum_{i=1}^{I} X_{b,i}\,W_{1,i,h}}_{\text{线性变换}} + b_{1,h}\Big).$$

   矩阵形式可写为

   $$H \;=\; \mathrm{ReLU}\big(X\,W_1 \;+\; \mathbf{1}_B\,b_1^T\big), \quad H \in \mathbb{R}^{B\times H},$$

   其中 $\mathbf{1}_B \in \mathbb{R}^B$ 为全 1 列向量，$\mathbf{1}_B\,b_1^T$ 对应对每一行加同一个偏置向量。

2. **第二层（线性变换 + 偏置）**
   对于第 b 个样本和第 o 个输出神经元（$o = 1, \ldots, O$），

   $$Y_{b,o} = \sum_{h=1}^{H} H_{b,h}\,W_{2,h,o} \;+\; b_{2,o}.$$

   矩阵形式为

   $$Y \;=\; H\,W_2 \;+\; \mathbf{1}_B\,b_2^T, \quad Y \in \mathbb{R}^{B\times O}.$$

综上，MLP 的前向传播即

$$H = \mathrm{ReLU}\big(X\,W_1 + \mathbf{1}_B\,b_1^T\big), \qquad Y = H\,W_2 + \mathbf{1}_B\,b_2^T.$$

其中 ReLU 激活函数定义为

$$\mathrm{ReLU}(z) \;=\; \max(0,\,z) \quad \text{(逐元素操作)}.$$

2. MLP 前向传播优化

在 GPU 上实现 MLP 前向传播时，主要优化思路包括：将偏置加法与激活函数融合、利用常量内存存储偏置、以及优化网格和线程配置，以减少内存访问并提高吞吐率。具体数学原理如下：

1. **融合偏置与激活（Bias + ReLU）**
   对第一层隐藏层计算而言，单独执行矩阵乘法后再分别进行偏置加法和 ReLU 激活会产生两次对隐藏矩阵 H 的全局内存读写。设中间结果为

   $$Z_{b,h} = \sum_{i=1}^{I} X_{b,i} W_{1,i,h} + b_{1,h}, \quad H_{b,h} = \max(0, Z_{b,h}).$$

   如果将偏置加法与 ReLU 激活合并为一个操作，则可在读出 $Z_{b,h}$ 的寄存器后，直接执行

   $$H_{b,h} = \max\big(0, Z_{b,h} + b_{1,h}\big),$$

   并将结果一次写回全局内存。这样，矩阵乘法内核只负责计算 $\sum XW_1$ 并将中间结果保存在全局内存，后续融合内核一次性完成“加偏置 + ReLU”：

   $$H_{b,h} = \max\big(0, H_{b,h} + b_{1,h}\big).$$

   由于融合操作只需要对每个元素做一次读—算—写，减少了额外的内存往返，降低了带宽开销。

2. **利用常量内存存储偏置（Constant Memory）**
   偏置向量 $b_1 \in \mathbb{R}^H$ 和 $b_2 \in \mathbb{R}^O$ 的长度远小于激活矩阵，且每个偏置在所有线程中频繁读取。将它们拷贝到 GPU 常量内存（Constant Memory）后，各线程读取偏置时可共享同一个 L1/L2 缓存副本，访问延迟更低，并且带宽需求更小。数学上，相当于在计算

   $$Z_{b,h} + b_{1,h} \quad \text{和} \quad Y_{b,o} + b_{2,o}$$

   时，$b_{1,h}$, $b_{2,o}$ 的读取成本被降至常量。此时融合内核仅需在每个线程中通过：

   $$A_{idx} = A_{idx} + c_B[\,\mathrm{col}(idx)\,],$$

   读取对应列的常量偏置，完成更新。

3. **优化网格与线程配置（Grid & Block）**
   对于每个 GEMM 内核（$H = XW_1$ 和 $Y = HW_2$），设线程块大小为 (16×16)，则需要满足：

   $$\text{grid.x} = \lceil K/16 \rceil, \qquad \text{grid.y} = \lceil M/16 \rceil,$$

   其中对于第一层，$M = B$（批量大小）、$K = H$；对于第二层，$M = B$、$K = O$。这样可保证每个线程对应输出矩阵中 (row, col) 的一个元素，且能对连续列或行进行 coalesced 访问，数学上减少了线程间因“非对齐”导致的内存分裂访问。

4. **内存访问与算术密集度提升**
   将偏置与激活融合后，每个隐藏元素仅需一次全局写回，原本的额外写-读过程被消除。记第一层计算产生的浮点运算量为

   $$\text{FLOPs}_1 = 2\,B\,I\,H,$$

而融合后对 H 的内存访问量从原先

$$8\,B\,H\,(\text{写入}+\text{读取})+8\,H\;=\;16\,B\,H+8\,H$$

减少为

$$8\,B\,H\quad(\text{一次写入})+8\,H,$$

从而算术强度（Arithmetic Intensity）得以提升。第二层类似，只需一次写入：

$$\text{FLOPs}_2=2\,B\,H\,O,\quad \text{Bytes}_2=8\,B\,O+8\,O.$$

总体来看，融合操作使得整个前向传播的内存带宽需求在全局内存层面降低近 2×，更接近于计算带宽平衡点。

以上优化手段在数学上共同作用，减少了不必要的全局内存往返、提高了偏置读取效率，并通过合理的线程映射实现了对连续内存的共聚访问，从而提升了 GPU 上 MLP 前向传播的整体性能。“‘

(三) 基于 MLP 的低轨卫星网络带宽预测性能优化原理

针对低轨卫星带宽时序预测的 MLP 训练与推理过程，主要优化思路可概括如下：

- **数据预处理与滑动窗口**对原始时序 $\{d_t\}$ 做最小—最大归一化：

  $$d'_t=\frac{d_t-d_{\min}}{d_{\max}-d_{\min}},$$

  然后以窗口大小 I 构造样本

  $$\mathbf{x}^{(i)}=[\,d'_i,\ldots,d'_{i+I-1}\,],\quad y^{(i)}=d'_{i+I}.$$

  归一化保证训练数值尺度一致，有助于梯度稳定。

- **小批量并行（Batch）**将训练集划分为大小为 $b=\text{BATCH\_SIZE}$ 的小批量，

  $$\mathbf{X}_{\text{batch}}\in\mathbb{R}^{b\times I},\quad \mathbf{y}_{\text{batch}}\in\mathbb{R}^{b\times 1}.$$

  在 GPU 上仅调用一次显存分配，并在每个批次中复用缓冲区，减少 `hipMalloc` 开销。

- **前向传播：矩阵乘法与并行化**第一层线性变换：

  $$\mathbf{H}_{\text{lin}}=\mathbf{X}_{\text{batch}}\,W_1,\quad W_1\in\mathbb{R}^{I\times H}.$$

  使用自定义 GEMM 内核 `matmul_kernel`，令线程块大小 (16×16)，并行计算 $b\times H$ 个元素。加偏置与 ReLU 分两步：

  $$H_{\text{act},\,ij}=\max\bigl(0,\;H_{\text{lin},\,ij}+b_{1,j}\bigr),$$

  通过 `add_bias_kernel` 和 `relu_kernel` 并行完成。第二层同理：

  $$\mathbf{Y}_{\text{pred}}=\mathbf{H}_{\text{act}}\,W_2+b_2,\quad W_2\in\mathbb{R}^{H\times 1}.$$

- **反向传播：并行梯度计算**损失为均方误差 (MSE)：

  $$\mathcal{L}=\frac{1}{b}\sum_{i=1}^{b}(\hat{y}_i-y_i)^2.$$

1. 输出层梯度：

$$\delta_i^{(2)} = \frac{2}{b}\big(\hat{y}_i - y_i\big), \quad i = 1, \dots, b.$$

2. 隐层误差：

$$\delta_{ij}^{(1)} = \delta_i^{(2)}\, W_{2,j}\, \mathbf{1}(H_{\text{lin},\,ij} > 0), \quad j = 1, \dots, H.$$

均由 `compute_output_grad` 和 `compute_relu_backward` 内核并行计算。

3. 梯度归约：将 $\mathbf{H}_{\text{act}}$、$\boldsymbol{\delta}^{(2)}$ 及 $\boldsymbol{\delta}^{(1)}$ 拷回 Host 后，用双重循环计算

$$\nabla W_{2,j} = \sum_i H_{\text{act},\,ij}\, \delta_i^{(2)}, \quad \nabla b_2 = \sum_i \delta_i^{(2)},$$

$$\nabla b_{1,j} = \sum_i \delta_{ij}^{(1)}, \quad \nabla W_{1,kj} = \sum_i X_{ik}\, \delta_{ij}^{(1)}.$$

- **参数更新：SGD 并行内核**使用简单的随机梯度下降：

$$W \leftarrow W - \eta\, \nabla W, \quad b \leftarrow b - \eta\, \nabla b, \quad \eta = \text{LEARNING\_RATE}.$$

在 GPU 上调用 `sgd_update`，让每个线程负责一个参数元素，完成并行更新。

- **数据传输优化**
  - 一次性分配所有 GPU 缓冲区，避免频繁 `hipMalloc`。
  - 每个小批量只拷贝 $\mathbf{X}_{\text{batch}}$ 和对应标签到设备，完成所有计算后，再分批将必要向量拷回 Host 进行归约。
  - 所有矩阵、向量按行优先存储，保证内核中对连续元素的访问可合并，提高全局内存带宽利用。

- **Xavier 初始化与数值稳定**权重 W_1, W_2 采用 Xavier 均匀分布：

$$W_{1,ij} \sim \mathcal{U}\Big[-\sqrt{\tfrac{6}{I+H}}, \sqrt{\tfrac{6}{I+H}}\Big], \quad W_{2,j1} \sim \mathcal{U}\Big[-\sqrt{\tfrac{6}{H+1}}, \sqrt{\tfrac{6}{H+1}}\Big],$$

使前向激活和反向梯度方差均衡，加速收敛并减少数值不稳定风险。

简而言之，通过归一化、批处理并行、定制化 GEMM 内核、逐元素并行操作、Host 归约与 GPU 并行更新，以及一次性显存分配和连续内存访问等策略，最大化利用 GPU 计算单元与内存带宽，从而显著提升低轨卫星带宽预测任务的训练与推理性能。

四、 优化实现

(一) 矩阵乘法算法优化实现

首先我们实现基线矩阵乘法方法，这也将作为我们对后续方法正确性验证的基础，对于正确性的检验，我们验证两个矩阵的差值是否在我们的误差允许范围内：

```
1  void matmul_baseline(const std::vector<double>& A,
2                       const std::vector<double>& B,
3                       std::vector<double>& C,
4                       int n, int m, int p) {
5      for (int i = 0; i < n; ++i) {
6          for (int j = 0; j < p; ++j) {
```

```
            double sum = 0.0;
            for (int k = 0; k < m; ++k) {
                sum += A[i * m + k] * B[k * p + j];
            }
            C[i * p + j] = sum;
        }
    }
}
```

```
bool validate(const std::vector<double>& A, const std::vector<double>& B,
    int rows, int cols, double tol = 1e-6) {
    for (int i = 0; i < rows * cols; ++i) {
        if (std::abs(A[i] - B[i]) > tol) return false;
    }
    return true;
}
```

我们自然能想到使用 openmp 并行后面的两层循环，于是就有:

```
    void matmul_openmp(const std::vector<double>& A,
                       const std::vector<double>& B,
                       std::vector<double>& C,
                       int n, int m, int p) {
    #pragma omp parallel for collapse(2) schedule(static)
    for (int i = 0; i < n; ++i) {
        for (int j = 0; j < p; ++j) {
            double sum = 0.0;
            for (int k = 0; k < m; ++k) {
                sum += A[i * m + k] * B[k * p + j];
            }
            C[i * p + j] = sum;
        }
    }
}
```

在 openmp 的基础上我们进一步对矩阵进行展开并行计算:

```
    void matmul_block(const std::vector<double>& A,
                      const std::vector<double>& B,
                      std::vector<double>& C,
                      int n, int m, int p, int block_size = BLOCK_SIZE) {
    std::fill(C.begin(), C.end(), 0.0);
    #pragma omp parallel for schedule(static)
    for (int ii = 0; ii < n; ii += block_size) {
        int i_max = std::min(ii + block_size, n);
        for (int kk = 0; kk < m; kk += block_size) {
            int k_max = std::min(kk + block_size, m);
            for (int jj = 0; jj < p; jj += block_size) {
                int j_max = std::min(jj + block_size, p);
                for (int i = ii; i < i_max; ++i) {
```

```
                    for (int k = kk; k < k_max; ++k) {
                        double a_val = A[i * m + k];
                        for (int j = jj; j < j_max; ++j) {
                            C[i * p + j] += a_val * B[k * p + j];
                        }
                    }
                }
            }
        }
    }
}
```

基于 cpu 的架构，我们优化访问顺序，对 cache 的访问进行了优化：

```
void matmul_transpose(const std::vector<double>& A,
                      const std::vector<double>& B,
                      std::vector<double>& C,
                      int n, int m, int p) {
    // 先转置 B -> Bt (维度 p x m)
    std::vector<double> Bt(p * m);
    for (int k = 0; k < m; ++k) {
        for (int j = 0; j < p; ++j) {
            Bt[j * m + k] = B[k * p + j];
        }
    }
    std::fill(C.begin(), C.end(), 0.0);
    #pragma omp parallel for collapse(2) schedule(static)
    for (int i = 0; i < n; ++i) {
        for (int j = 0; j < p; ++j) {
            const double* a_row = &A[i * m];
            const double* b_row = &Bt[j * m];
            double sum = 0.0;
            for (int k = 0; k < m; ++k) {
                sum += a_row[k] * b_row[k];
            }
            C[i * p + j] = sum;
        }
    }
}
```

在并行程序设计的课程中我们学习到 simd 方法：

```
void matmul_simd(const std::vector<double>& A,
                 const std::vector<double>& B,
                 std::vector<double>& C,
                 int n, int m, int p) {
    std::fill(C.begin(), C.end(), 0.0);
    for (int i = 0; i < n; ++i) {
        const double* a_row = &A[i * m];
        for (int j = 0; j < p; j += 4) {
```

```
            __m256d c_vec = _mm256_setzero_pd();
            for (int k = 0; k < m; ++k) {
                __m256d a_vec = _mm256_set1_pd(a_row[k]);
                __m256d b_vec = _mm256_loadu_pd(&B[k * p + j]);
                c_vec = _mm256_fmadd_pd(a_vec, b_vec, c_vec);
            }
            _mm256_storeu_pd(&C[i * p + j], c_vec);
        }
    }
}
```

GPU（DCU）加速实现基于 HIP 框架，我们实现了高效 tile-based 并行矩阵乘法。核函数如下：

```
// TILE 为线程块尺寸
__global__ void matmul_kernel(const double* A, const double* B, double* C,
                              int n, int m, int p) {
    __shared__ double tileA[TILE][TILE];
    __shared__ double tileB[TILE][TILE];

    int row = blockIdx.y * TILE + threadIdx.y;
    int col = blockIdx.x * TILE + threadIdx.x;

    double sum = 0.0;
    for (int t = 0; t < (m + TILE - 1) / TILE; ++t) {
        if (row < n && t * TILE + threadIdx.x < m)
            tileA[threadIdx.y][threadIdx.x] = A[row * m + t * TILE +
                threadIdx.x];
        else
            tileA[threadIdx.y][threadIdx.x] = 0.0;

        if (col < p && t * TILE + threadIdx.y < m)
            tileB[threadIdx.y][threadIdx.x] = B[(t * TILE + threadIdx.y) * p
                + col];
        else
            tileB[threadIdx.y][threadIdx.x] = 0.0;

        __syncthreads();

        for (int i = 0; i < TILE; ++i)
            sum += tileA[threadIdx.y][i] * tileB[i][threadIdx.x];

        __syncthreads();
    }

    if (row < n && col < p)
        C[row * p + col] = sum;
}
```

主机端通过如下调用流程完成执行：

- 使用 `hipMalloc` 分配显存；
- 调用 `hipMemcpy` 拷贝主机数据至设备；
- 配置 `dim3 gridDim, blockDim` 启动核函数；
- 使用 `hipEventRecord` 记录执行时间；
- 使用 `hipMemcpy` 拷回计算结果并与 CPU 验证；

在这里我们分配资源后，对 DCU 进行分配计算任务，调用核函数进行任务，同时也对运算进行计时，最终我们将结果写回，获得结果。

我们也测试了这些方法的杂交方法，在后续实验结果部分将会对他们进行分析。

(二) MLP 前向传播优化实现

在本节中，我们介绍一个两层全连接神经网络（MLP）的前向传播在 GPU 上的优化实现。该网络包含一个隐藏层，并使用 ReLU 激活函数。我们利用 GPU 的并行计算能力加速计算过程。以下是实现中的关键部分，主机端的数据准备和性能度量等非核心部分将通过文字描述衔接。

1. GPU 优化实现

GPU 实现基于 HIP API，设计了三个核心内核函数来完成矩阵乘法、偏置相加和 ReLU 激活。

1. 矩阵乘法内核 矩阵乘法是前向传播中最耗时的操作。我们实现了一个通用的矩阵乘法内核，计算 $\mathbf{C} = \mathbf{A} \times \mathbf{B}$，适用于输入到隐藏层和隐藏层到输出层的计算。

```
__global__ void matmul_kernel(const double* __restrict__ A,
                               const double* __restrict__ B,
                               double* __restrict__ C,
                               int M, int N, int K)
{
    int row = blockIdx.y * blockDim.y + threadIdx.y;
    int col = blockIdx.x * blockDim.x + threadIdx.x;

    if (row < M && col < K) {
        double sum = 0.0;
        for (int n = 0; n < N; ++n) {
            sum += A[row * N + n] * B[n * K + col];
        }
        C[row * K + col] = sum;
    }
}
```

该内核使用二维线程块组织，每个线程计算结果矩阵 $\mathbf{C}$ 的一个元素，通过并行化行和列充分利用 GPU 并行性。

2. 加偏置内核 矩阵乘法后需为每列添加偏置值，这一操作高度并行化。

```
__global__ void add_bias_kernel(double* A, const double* bias, int rows, int
    cols)
{
    int idx = blockIdx.x * blockDim.x + threadIdx.x;
    int total = rows * cols;
    if (idx < total) {
        int col = idx % cols;
        A[idx] += bias[col];
    }
}
```

此内核采用一维线程组织，每个线程根据列索引添加对应的偏置值。

3. ReLU 激活内核 ReLU 函数对隐藏层输出应用 $\mathrm{ReLU}(x) = \max(0, x)$，同样适合并行执行。

```
__global__ void relu_kernel(double* A, int size)
{
    int idx = blockIdx.x * blockDim.x + threadIdx.x;
    if (idx < size) {
        A[idx] = A[idx] > 0.0 ? A[idx] : 0.0;
    }
}
```

与加偏置内核类似，每个线程处理一个元素，完成激活操作。

主机端代码负责数据初始化、内存分配和数据传输。例如，输入数据和网络参数使用随机值初始化，并通过 `hipMalloc` 和 `hipMemcpy` 管理设备内存。内核启动时，配置二维网格和块尺寸（如 16×16）以计算隐藏层和输出层结果，随后同步设备并拷回结果。性能通过 GFLOPS 和内存带宽评估，验证则比较 CPU 和 GPU 输出的一致性。

在后续的过程中，我们也将展示将 relu 核与偏置内核进行融合对性能的优化。

(三) 基于 MLP 的低轨卫星网络带宽预测实现及性能优化

我们接下来实现多层感知器（MLP）预测低轨卫星网络带宽的 GPU 优化。通过滑动窗口生成输入输出对，使用 Xavier 初始化权重，并采用带动量的 SGD 优化器训练模型。以下聚焦核心 GPU 内核实现。

1. 核心 GPU 实现

GPU 实现利用 HIP API，关键内核包括矩阵乘法、ReLU 激活和带动量的参数更新。

1. 矩阵乘法内核 矩阵乘法处理 $\mathbf{H} = \mathbf{X} \times \mathbf{W1}$ 和 $\mathbf{Y} = \mathbf{H} \times \mathbf{W2}$，并行计算提升效率。

```
__global__ void matmul_kernel(const double* __restrict__ A,
                              const double* __restrict__ B,
                              double* __restrict__ C,
                              int M, int N, int K)
{
    int row = blockIdx.y * blockDim.y + threadIdx.y;
```

```
    int col = blockIdx.x * blockDim.x + threadIdx.x;
    if (row < M && col < K) {
        double sum = 0.0;
        for (int n = 0; n < N; ++n) {
            sum += A[row * N + n] * B[n * K + col];
        }
        C[row * K + col] = sum;
    }
}
```

2. ReLU 激活内核 ReLU 函数 $\mathrm{ReLU}(x) = \max(0, x)$ 对隐藏层输出并行应用。

```
__global__ void relu_kernel(double* A, int size)
{
    int idx = blockIdx.x * blockDim.x + threadIdx.x;
    if (idx < size) {
        A[idx] = (A[idx] > 0.0) ? A[idx] : 0.0;
    }
}
```

3. 带动量的 SGD 更新内核 参数更新结合动量加速梯度下降, 计算 $v = \mathrm{momentum} \cdot v + \mathrm{lr} \cdot \mathrm{grad}$ 并更新权重。

```
__global__ void momentum_sgd_update(double* weights, double* velocities,
    const double* grad,
                                    double lr, double momentum, int size)
{
    int idx = blockIdx.x * blockDim.x + threadIdx.x;
    if (idx < size) {
        double v_old = velocities[idx];
        double v_new = momentum * v_old + lr * grad[idx];
        velocities[idx] = v_new;
        weights[idx] -= v_new;
    }
}
```

数据从 JSON 文件加载并归一化后，通过滑动窗口生成训练和测试样本。权重采用 Xavier 初始化，训练过程分批执行前向传播、梯度计算和参数更新，损失以 MSE 评估。测试阶段计算归一化和反归一化后的 MSE，验证预测精度。

五、 实验结果与分析

(一) 矩阵乘法优化实验结果分析

我们利用下列的方法对矩阵乘法进行优化的结果列表格如下:

将表格进行可视化:

由图和表我们可以很明显的看出: 所有的方法都通过了正确性检验。基线实现 (Baseline) 未采用任何并行或优化技术，计算耗时最长，性能最差。引入 OpenMP 并行后性能有明显提升，而

表 1: Matrix Multiplication Benchmark Results

Method	Time (ms)	GFLOP/s	BW (GB/s)	Valid	Speedup
CPU (Baseline)	10875.7	0.197	0.0027	–	1.00×
CPU (OpenMP)	1407.37	1.526	0.0209	True	7.73×
CPU (Block Tiling)	61.89	34.699	0.4744	True	175.74×
CPU (Transpose)	299.24	7.176	0.0981	True	36.35×
CPU (SIMD)	2605.42	0.824	0.0113	True	4.17×
CPU (SIMD+OpenMP)	360.4	5.959	0.0815	True	30.17×
CPU (SIMD+Block)	386.33	5.559	0.0760	True	28.15×
GPU (HIP/DCU)	3.27	655.928	8.9678	PASS	3325.58×

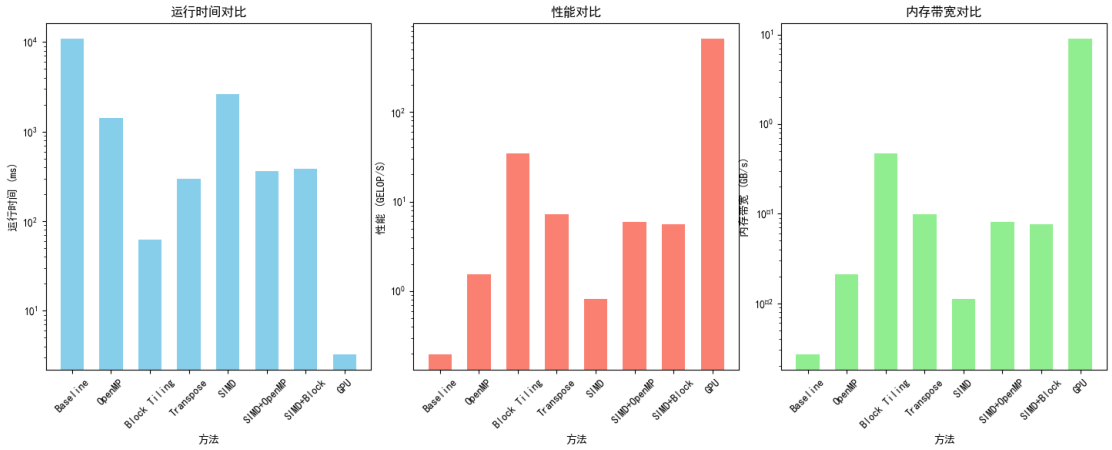

图 1: 矩阵乘法优化可视化，采用对数坐标轴

块划分 (Block Tiling) 策略则进一步优化了缓存局部性，使得性能大幅提升至约 34.7 GFLOP/s。转置优化改善了访问模式，但整体效果不及块划分。

使用 SIMD 指令集后，性能有所提升，但若未结合并行或块划分，其优势未能充分发挥。将 SIMD 与 OpenMP 或 Block 技术联合使用时，能有效兼顾向量化与并行性，取得更优性能。

相比之下，基于 HIP 的 GPU 实现充分利用了 DCU（Data Center GPU）的高吞吐能力，计算时间仅为 3.27 毫秒，性能达到 655.9 GFLOP/s，远超 CPU 上的各类优化方案。此外，所有 CPU 和 GPU 实现均通过了结果验证，保证了数值正确性。

(二) MLP 前向传播过程优化结果分析

我们首先将我们实现的 MLP 前向传播过程与对应的 CPU 实现对比, 我们一开始使用了参考代码中的尺寸进行实现和对比，但我们发现一个问题：**在这个情形下 DCU 的计算表现居然不如 CPU 串行实现**

GPU 性能未达预期的原因分析 如表 2 所示，在小规模输入（例如 1024×10 与 1024×20 的矩阵乘法）下，CPU 的前向传播耗时仅为 0.14 ms，而 GPU 执行同样操作的总时间却高达 0.65 ms，导致其 GFLOP/s 与内存带宽均远低于 CPU。究其原因，主要包括以下几点：

- **问题规模太小，GPU 并未充分利用：**当前矩阵乘操作的维度过小，GPU 无法调度足够的

表 2: CPU 与 DCU 在不同矩阵尺寸下的性能对比

Size	Time (ms)		Performance (GFLOP/s)		Bandwidth (GB/s)	
	CPU	DCU	CPU	DCU	CPU	DCU
1024	0.14133	0.64957	4.35	0.95	3.21	0.70
4096	0.56410	0.69469	4.36	3.54	3.20	2.60
8192	1.13315	0.69853	4.34	7.04	3.18	5.16

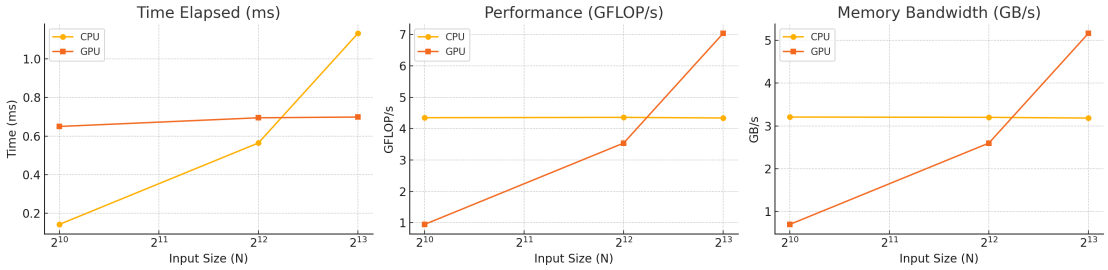

图 2: CPU 与 DCU 在不同矩阵尺寸下的性能变化对比，GPU 代指 DCU

线程参与计算，导致并行度严重不足。相反，CPU 更能在缓存中完成小矩阵乘法，并通过向量化等优化方式获得更优性能。

- **核函数启动与同步开销显著：** 每个 GPU 操作（如矩阵乘、加偏置、ReLU）都需单独启动一个 kernel，并进行一次同步。由于每个 kernel 仅进行少量计算，启动与同步成本在总时间中占据主导，导致效率低下。
- **带宽利用率不高：** 尽管理论上 GPU 拥有远高于 CPU 的显存带宽，但由于问题规模小、访问不连续、操作频繁切换等原因，未能实现流式高带宽访问，实测仅为 0.7 GB/s，远低于 CPU 的 3.2 GB/s。
- **算力密度不足：** 与卷积操作或大规模矩阵乘法不同，全连接层的计算密集度较低。GPU 需要足够大的工作负载才能让大量计算单元并行运行，而当前任务不足以激发其峰值性能。

于是我们尝试修改 batch 大小得到了如上的表格中的数据，可视化如下：我们不难发现：

- 运行时间：随着尺寸增大，CPU 用时线性增加，而 GPU 用时变化不大，主要受启动与同步开销影响。
- 性能（GFLOP/s）：GPU 性能随尺寸明显上升，CPU 基本持平，说明 GPU 对问题规模更敏感。
- 内存带宽：GPU 在大尺寸下带宽大幅提升，而 CPU 相对稳定，表明 GPU 在数据量大时能更有效地利用内存带宽。

1. 前向传播过程的进一步优化

我们尝试把“加偏置+ ReLU”融合到每个 matmul Kernel 里,使得“第一层 matmul+bias+ReLU”合成一个 kernel，“第二层 matmul+bias”合成一个 kernel。这样仅需 2 次 kernel 启动，能省下 3 次启动/同步开销。与原本的实现进行对比，实验数据表格如下：

表 3: MLP 前向传播各阶段耗时对比（单位：ms）

阶段	融核后时间	融核前时间
第一层（GEMM + Bias + ReLU）	0.1467	0.7438
第二层（GEMM + Bias）	0.0789	0.0570
Memcpy / 总前向	0.0370	0.1003

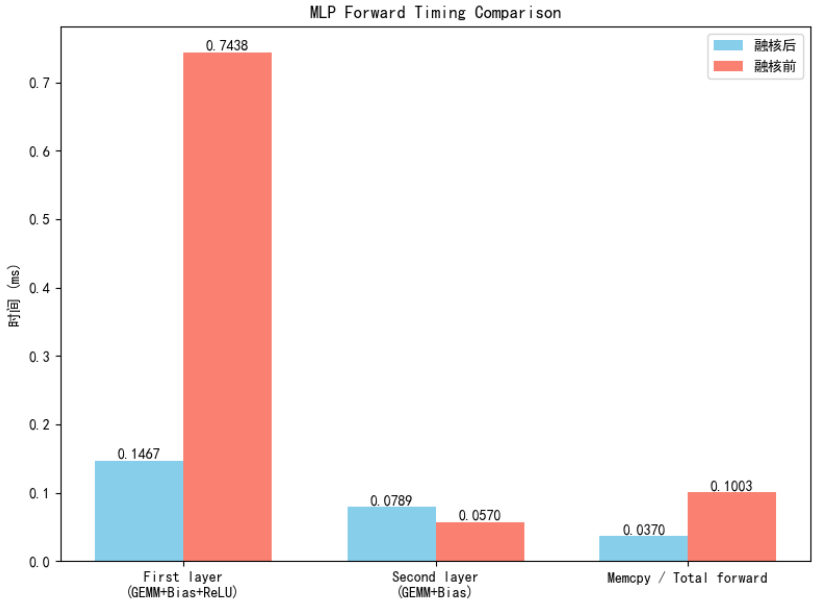

图 3: MLP 前向传播各阶段耗时对比（单位：ms）

可视化如下:

我们可以很明显地看到性能有所提升。

(三) MLP 的低轨卫星网络带宽预测结果分析

我们首先完全参考代码框架中的 MLP 实现，使用

原始数据 → 滑动窗口 → Minmax 归一化 → [-1,1] 均匀初始化 → GPU 前向传播 → MSE 损失计算 → GPU 反向传播 → 普通 SGD 更新 → 反归一化评估

来搭建我们的 MLP 模型，模型收敛时，**MSE=0.0456248**，我们随后对其进行优化尝试。

我们对初始化过程进行修改，我们使用 Xavier 初始化来作为新的初始化方式替代 [-1,1] 均匀初始化，这可以加速模型的收敛，模型表现有所优化，**MSE=0.0130282**，进一步地，我们替换 SGD 更新方式，变为带动量的 SGD，模型再次优化，**MSE=0.00841459**。

于是，我们得到了一个表现尚为不错的 MLP 模型，其训练过程可视化如下:

可以看到模型迅速收敛，而在 DCU 计算加速卡的作用下，我们得以快速地对模型进行调整优化，减少了时间成本。

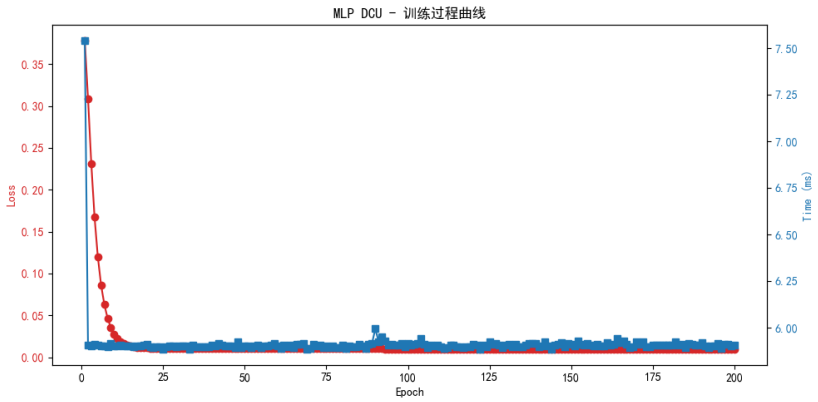

图 4: 最终模型的训练过程可视化

六、 总结

在本赛题中，我们通过实现和优化多层感知机（MLP）模型，探索了低轨（LEO）卫星网络带宽预测的高效解决方案：

- **基础任务完成：矩阵乘法优化**
  - 成功实现了标准矩阵乘法算法，支持浮点输入，针对 $N = 1024$、$M = 2048$、$P = 512$ 的随机矩阵验证了正确性。
  - 采用多种优化策略（块划分、转置、SIMD 向量化、OpenMP 并行、DCU 加速），显著提升性能：CPU 优化后最高达 34.7 GFLOP/s，GPU（HIP/DCU）实现 655.9 GFLOP/s，远超基线（0.197 GFLOP/s）。
- **进阶任务一：MLP 前向传播优化**
  - 基于矩阵乘法实现了三层 MLP（输入层 1024×10，隐藏层 10×20，输出层 20×5），支持前向传播和批处理（batch size = 1024）。
  - 通过融合“加偏置 + ReLU”操作、优化线程配置和内存访问，GPU 实现大幅减少 kernel 启动和同步开销，前向传播总耗时从 0.7438 ms 优化至 0.1467 ms（第一层），性能随矩阵规模增大而提升。
- **进阶任务二：LEO 带宽预测优化**
  - 设计并实现了支持前向传播、反向传播和梯度下降训练的 MLP 模型，基于 starlink_bw.json 数据集完成 LEO 卫星网络下行带宽预测。
  - 优化措施包括数据归一化、Xavier 初始化、带动量的 SGD 更新和 GPU 并行计算，最终模型 MSE 从 0.0456248 优化至 0.00841459，展现快速收敛和高预测精度。
  - 利用 DCU 加速卡提升训练和推理效率，显著降低时间成本，验证了硬件加速在实践中的价值。
- **未来展望**
  - **模型复杂度提升：**探索更深层次或更复杂模型（如 LSTM、Transformer），以捕捉 LEO 网络带宽的时序动态特性。

- **数据扩展：**结合真实卫星运行数据（如轨道参数、干扰因素），进一步验证和优化模型的泛化能力。
- **硬件与算法协同：**深入研究 DCU 架构特性，结合高级优化技术（如张量核心、混合精度计算），进一步提升计算效率。
- **实时应用：**开发实时带宽预测系统，针对动态拓扑和链路切换场景，测试模型在实际网络环境中的鲁棒性。

本实验我们通过软硬件结合优化，成功提升了矩阵运算效率和 LEO 卫星带宽预测性能，从而体会到了计算机系统设计时的一系列深刻的设计思想和历练。

致谢：特别感谢师建新老师和张金老师的悉心指导与支持。感谢曙光 DCU 训练平台的计算资源支持和实训平台支持。