

快速傅里叶变换：原理、GPU 加速与应用

快速傅里叶变换（Fast Fourier Transform, FFT）是**高效计算离散傅里叶变换（Discrete Fourier Transform, DFT）**的一类算法。它本身不是一种新的数学变换，而是把 DFT 从直接计算的 $O(N^2)$ 降低到通常的 $O(N \log N)$ 。

FFT 的价值可以概括为：把一个离散信号从“按时间或空间排列的样本”转换成“按频率排列的成分”，并且让这个转换足够快，从而可以用于实时音频、图像处理、通信、雷达、科学计算和深度学习。

时间或空间样本



FFT 计算



频率、幅度、相位

Figure 1: FFT 的输入、计算和输出

1. 先用三个例子建立直觉

1.1. 例 1：4 个数字也可以做频率分析

先看一个非常小的序列：

$$x = [1, 0, 1, 0].$$

它每隔一个样本就在 1 和 0 之间交替，因此直觉上应该包含一个“平均值”和一个“快速交替”的成分。对它做 4 点 DFT，结果如下：

频率索引 k	结果 X_k	怎么理解
0	2	平均值为 $\frac{2}{4} = 0.5$ ，称为直流分量
1	0	没有对应的周期成分
2	2	每两个样本重复一次，属于最高可见频率
3	0	实值输入的共轭对称位置

这里的“频率索引”不是直接的赫兹值。若采样率是 $f_s = 4$ Hz，则 $k = 1$ 代表 1 Hz， $k = 2$ 代表 2 Hz。这个例子说明：FFT 输出不是另一组“更好看的数字”，而是告诉我们原始数字由哪些周期成分组成。

1.1.1. 这个例子如何用蝶形算出来

把偶数下标和奇数下标分开：

$$x_{\text{even}} = [1, 1], \quad x_{\text{odd}} = [0, 0].$$

两个 2 点变换分别得到 $E = [2, 0]$ 、 $O = [0, 0]$ 。因为 $W_4^0 = 1$ 、 $W_4^1 = -j$ ，合并时只需要：

$$X_0 = E_0 + W_4^0 O_0 = 2, \quad X_2 = E_0 - W_4^0 O_0 = 2.$$

另一个蝶形给出 $X_1 = X_3 = 0$ 。这就是 FFT 所说的“先拆小，再合并”：没有改变答案，只是避免重复计算。

1.2. 例 2：识别一个 1 Hz 的正弦波

假设采样率为 $f_s = 8$ Hz，连续采样 1 秒，得到 8 个样本：

$$x_n = \sin\left(2\pi \frac{n}{8}\right), \quad n = 0, 1, \dots, 7.$$

这个信号每秒振荡一次，所以频谱应该在 1 Hz 附近出现峰值。8 点 FFT 的幅度可以用下面的简化柱状图表示（这里把幅度归一化到 1）：

索引 k	频率	归一化幅度	解释
0	0 Hz		没有直流偏置
1	1 Hz		正频率峰值
2	2 Hz		无明显成分
3	3 Hz		无明显成分
4	4 Hz		奈奎斯特频率

Table 1: 1 Hz 正弦波的单边频谱示意

对于实值信号, $k = 7$ 处还有一个与 $k = 1$ 对称的负频率峰值。实际代码中常用 `rfft` 只保留 $k = 0$ 到 $k = 4$ 的部分。

1.3. 例 3: GPU 像同时做很多个蝶形

把一次蝶形想象成下面的两行工作:

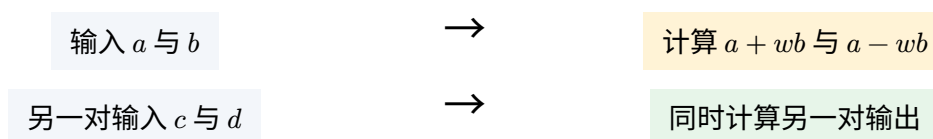


Figure 2: 一个阶段中彼此独立的蝶形可以并行

CPU 可以依次做这些蝶形; GPU 则让成百上千个线程同时处理许多蝶形。难点在于下一阶段必须等上一阶段写完结果, 所以 GPU 加速不仅是“线程更多”, 还要安排好共享内存、全局同步和连续内存访问。

2. 先回答三个问题

2.1. 它解决什么问题

给定 N 个等间隔采样值

$$x_0, x_1, \dots, x_{N-1},$$

我们经常想知道: 信号里有哪些周期成分、各自的强度和相位是多少。DFT 给出这些答案: 第 k 个输出表示频率索引 k 对应的复数频谱系数。

FFT 主要解决的是**计算量太大**的问题。直接计算每个频率的内积, 需要 N 个输出、每个输出 N 次乘加, 总计约 N^2 次操作; FFT 利用对称性和重复子问题, 只需约 $N \log_2 N$ 级别的操作。

2.2. 它不自动解决什么问题

FFT 不会凭空提高采样率、消除噪声或区分超过奈奎斯特频率的成分。它只能分析已经采集的数据, 并且结果受以下因素限制:

- 采样率决定可观察的最高频率;
- 采样时长决定频率分辨率;
- 有限长度截断会产生谱泄漏;
- 噪声、量化误差和时钟抖动会污染频谱;
- 频域结果通常是复数, 幅度和相位都可能有意义。

2.3. FFT 与 DFT 的关系

DFT 是数学定义, FFT 是计算 DFT 的算法族。可以使用不同的 FFT 结构, 例如基 2、基 4、混合基、分裂基、Bluestein 和 Rader 算法。输入长度不是 2 的幂时, 仍然可以做 FFT, 但实现可能采用混合基、补零或其他算法, 性能不一定最好。

3. 从傅里叶级数到 DFT

3.1. 连续信号的频率分解

傅里叶分析的核心想法是：复杂信号可以看成许多不同频率的正弦波（或复指数）的叠加。

连续时间傅里叶变换写作：

$$X(f) = \int_{-\infty}^{\infty} x(t) \exp(-j2\pi ft) dt.$$

复指数 $\exp(j2\pi ft)$ 同时包含正弦和余弦，因此一个频率分量需要两个信息：幅度和相位。

3.2. 采样后的 DFT

对长度为 N 的离散序列，DFT 定义为：

$$X_k = \sum_{n=0}^{N-1} x_n e^{-j2\pi k \frac{n}{N}}, \quad k = 0, 1, \dots, N-1.$$

记

$$W_N = e^{-j2\pi \frac{1}{N}},$$

则

$$X_k = \sum_{n=0}^{N-1} x_n W_N^{kn}.$$

逆变换为：

$$x_n = \frac{1}{N} \sum_{k=0}^{N-1} X_k e^{j2\pi k \frac{n}{N}}.$$

不同库对归一化的位置约定不同：有的库只在逆变换除以 N ，有的库使用正逆变换各除以 $\sqrt{N}$ 。比较结果时必须先确认这个约定。

3.3. 频率索引如何对应真实频率

若采样率为 f_s ，则第 k 个频点对应

$$f_k = k \frac{f_s}{N}.$$

对实值输入， $X_{N-k} = \overline{X_k}$ ，所以正频率和负频率形成共轭对称。通常只看 0 到 $\frac{f_s}{2}$ 的单边谱。频率分辨率为

$$\Delta f = \frac{f_s}{N} = \frac{1}{T},$$

其中 $T = \frac{N}{f_s}$ 是观测时间。补零可以让频谱曲线看起来更平滑，但不会真正提高由观测时长决定的分辨能力。

3.3.1. 例：点数如何影响频率分辨率

采样率 f_s	点数 N	$\Delta f = \frac{f_s}{N}$	能否区分相近频率
1000 Hz	100	10 Hz	只能粗略区分 100 Hz 和 110 Hz
1000 Hz	1000	1 Hz	可以较好区分 100 Hz 和 101 Hz
1000 Hz	100 个样本补零到 1000	1 Hz 频率网格	曲线更密，但实际分辨率仍约 10 Hz

4. FFT 为什么能做到 $O(N \log N)$

下面以最经典的 Cooley-Tukey 基 2 FFT 为例，假设 $N = 2^m$ 。

4.1. 偶奇分解

把 DFT 求和按输入下标的奇偶拆开：

$$X_k = \sum_{r=0}^{\frac{N}{2}-1} x_{2r} W_N^{k(2r)} + \sum_{r=0}^{\frac{N}{2}-1} x_{2r+1} W_N^{k(2r+1)} = E_k + W_N^k O_k. \quad (1)$$

其中 E_k 是偶数下标样本的 $\frac{N}{2}$ 点 DFT, O_k 是奇数下标样本的 $\frac{N}{2}$ 点 DFT。

利用旋转因子的周期性, 可以同时得到两个输出:

$$X_k = E_k + W_N^k O_k,$$

$$X_{k+\frac{N}{2}} = E_k - W_N^k O_k, \quad 0 \leq k < \frac{N}{2}.$$

这两个式子构成一个**蝶形 (butterfly)**: 一次复乘和两次复加, 就产生两个输出。

4.2. 递归规模

一个 N 点问题被拆成两个 $\frac{N}{2}$ 点问题, 再做 $O(N)$ 的合并:

$$T(N) = 2T\left(\frac{N}{2}\right) + O(N).$$

由此得到

$$T(N) = O(N \log_2 N).$$

例如, $N = 1,048,576 = 2^{20}$ 时, FFT 只有 20 个蝶形阶段; 直接 DFT 则需要约一万亿数量级的乘加。

4.3. 迭代实现与位反转

工程实现通常不递归, 而是分成 $\log_2 N$ 个阶段, 每个阶段扩大蝶形跨度。为了让原地 (in-place) 计算成立, 输入经常需要按**位反转排列**: 把下标的二进制位倒过来作为新下标。

位反转并不是 FFT 的数学要求, 而是某些迭代实现的数据布局要求。另一类实现使用 Stockham 结构, 通过交替读写两个缓冲区避免显式位反转, 更适合 GPU, 但会增加内存读写或缓冲区管理成本。

5. 一个蝶形到底做什么

设上支路为 a , 下支路为 b , 旋转因子为 w , 蝶形输出为:

$$y_0 = a + wb, \quad y_1 = a - wb.$$

这里的 $w = e^{-j\theta}$ 只是单位圆上的一个复数。由于很多蝶形反复使用同一组旋转因子, 实际库会:

- 预计算 twiddle factors (旋转因子);
- 利用共轭、周期性和对称性减少存储;
- 对小规模变换使用手写的 radix-2、radix-4 或更高基数内核;
- 使用融合乘加、向量指令和 SIMD 计算复数运算。

FFT 的主要代价不仅是复乘加, 还包括数据重排、跨层级内存访问, 以及多批次变换时的步长 (stride) 处理。

6. 采样、窗函数与谱泄漏

6.1. 奈奎斯特限制

采样率为 f_s 时, 能无混叠表示的最高频率约为 $\frac{f_s}{2}$ 。频率高于这个范围的成分会折叠到低频, 这就是混叠 (aliasing)。抗混叠低通滤波器应放在 ADC 之前; FFT 之后再滤波无法恢复已经折叠的信息。

6.1.1. 例: 为什么 6 Hz 可能被看成 2 Hz

如果采样率只有 $f_s = 8$ Hz, 奈奎斯特频率是 4 Hz。一个真实频率为 6 Hz 的正弦波超过了这个上限, 采样后会折叠到

$$|6 - 8| = 2 \text{ Hz}.$$

所以 FFT 看到的峰值可能在 2 Hz, 而不是 6 Hz。提高 FFT 点数不能修复这个问题, 必须在采样前提高采样率或使用抗混叠滤波器。

6.2. 有限窗口导致泄漏

FFT 实际上只观察有限时间窗口，相当于把无限长信号乘以一个矩形窗。时域相乘对应频域卷积，因此一个不落在频点上的正弦波会扩散到邻近频点，这叫谱泄漏。

常见窗函数：

- Hann：通用、旁瓣较低，常用于音频和振动分析；
- Hamming：主瓣与旁瓣折中；
- Blackman：泄漏抑制更强，但主瓣更宽；
- Flat-top：幅值测量更准确，但频率分辨率较差；
- 矩形窗：主瓣最窄，但旁瓣和泄漏较明显。

窗函数不是越强越好。它在降低旁瓣的同时会扩大主瓣，实际选择取决于“分辨相近频率”还是“准确测幅值”。

6.3. 实值输入的优化

输入为实数时，频谱具有共轭对称，库可以使用 real-to-complex (R2C) 变换只保存约 $\frac{N}{2} + 1$ 个复数。逆变换则是 complex-to-real (C2R)。这能减少算力和内存，但输出布局与普通复数 FFT 不同，不能只靠类型名猜测数组长度。

7. GPU 如何加速 FFT

GPU 加速的本质是：同时执行大量彼此独立或局部独立的蝶形，并尽量让数据访问合并、同步次数少、片上存储利用率高。

7.1. 并行化结构

对一个一维 FFT：

1. 每个 FFT 实例（或 batch 中的每一行）可以分配给一个线程块、多个线程块，甚至多个 GPU。
2. 在同一个阶段中，不同蝶形之间互不依赖，可以并行计算。
3. 一个阶段完成后，下一阶段需要上一阶段的结果，因此阶段之间存在全局同步或 kernel 边界。
4. 对二维或三维 FFT，通常先沿一个维度做批量一维 FFT，再转置或改变 stride，沿下一个维度继续做。

GPU 版本经常采用 Stockham 或 tiled FFT，使每个阶段的读写模式更规则；对很大的变换，也可能采用分块 FFT、转置和多次 kernel 调度。

7.2. 内存层次是关键

FFT 不是单纯的算术密集型问题。GPU 实现通常按以下层次组织数据：

- **寄存器**：保存线程当前负责的少量复数和旋转因子，延迟最低；
- **共享内存**：在一个线程块内交换蝶形数据，适合小块 FFT，但容量有限且需要同步；
- **L2/显存**：保存完整输入、输出和大规模 batch，带宽高但延迟大；
- **常量或只读缓存**：保存被大量线程重复读取的旋转因子。

如果每个阶段都把数据从显存读回、写回，内核会被显存带宽和 kernel 启动开销限制。高性能库会融合多个阶段、使用共享内存 tile，并优化 stride，使一次全局内存事务服务多个相邻线程。

7.3. 蝶形映射到线程

最简单的映射是“一个线程计算一个蝶形”。但随着阶段变化，蝶形距离会增大，线程访问可能不再连续，导致非合并访问和 bank conflict。实际实现会根据变换大小选择不同策略：

- 小 FFT：一个线程块把整个变换放进共享内存，连续做多阶段蝶形；
- 中等 FFT：一个线程处理多个元素，增加寄存器复用；
- 大 FFT：分成多个 tile，阶段间写回显存，并配合转置；
- 批量 FFT：把 batch 维度映射给线程块，隐藏单个 FFT 规模不足的问题。

7.4. cuFFT、rocFFT 与框架接口

生产环境通常不手写 FFT kernel，而使用经过优化的库：

- NVIDIA GPU：cuFFT；
- AMD GPU：rocFFT；
- Intel GPU：oneMKL/oneAPI 相关 FFT 实现；
- CUDA 生态的深度学习框架：PyTorch、JAX、CuPy 等会调用底层库或自己的融合内核。

这类库通常先创建一个 plan。plan 会根据尺寸、类型、batch、stride 和设备情况选择算法并分配临时 workspace。因此应当：

- 重复使用相同尺寸时缓存 plan；
- 把 plan 创建时间与实际执行时间分开测量；
- 预热 GPU 后再 benchmark；
- 使用 CUDA event 或框架提供的同步计时，避免异步提交导致测量偏小；
- 同时报告端到端时间，包括主机到设备和设备到主机拷贝。

7.5. GPU 加速并不总是更快

以下情况 CPU 可能更合适：

- FFT 很小，GPU kernel 启动和数据传输占主导；
- 只有一次变换，无法摊薄初始化和拷贝成本；
- 数据本来就在 CPU 内存中；
- batch 太小，GPU 并行度不足；
- 需要很高精度，而 GPU 的双精度吞吐较低；
- stride 很差，导致全局内存访问不连续。

如果后续步骤也在 GPU 上进行，例如频域滤波、卷积、谱分析或神经网络，那么可以保留数据在显存中，减少往返拷贝，GPU 的优势通常会明显得多。

一个便于记忆的性能判断表：

场景	CPU	GPU	原因
一次 256 点 FFT	通常更合适	通常不划算	启动 kernel 和拷贝的固定成本太大
1 万次 1K 点 FFT	可以	通常更快	batch 足够大，可以隐藏延迟
一张 4096×4096 图像	较慢	通常更快	二维 FFT 有大量并行蝶形
FFT 后还要做 GPU 卷积	需要来回拷贝	适合	数据可以一直留在显存中

因此 benchmark 时不要只问“GPU 的 FFT 多快”，而要问“从输入到最终结果的整条流水线多快”。

8. 卷积为什么常用 FFT

离散卷积定义为：

$$(x * h)[n] = \sum_m x[m]h[n - m]. \quad (2)$$

卷积定理说明：

$$\mathcal{F}\{x * h\} = \mathcal{F}\{x\}\mathcal{F}\{h\}.$$

因此可以按以下步骤计算线性卷积：

1. 将 x 和 h 补零到至少 $N + M - 1$ 的长度；

2. 分别做 FFT；
3. 逐点相乘频谱；
4. 做逆 FFT；
5. 取出有效的 $N + M - 1$ 个样本。

复杂度约为 $O(L \log L)$ ，其中 $L \geq N + M - 1$ 。当卷积核很大时，FFT 卷积通常优于直接卷积；当卷积核很小或只处理很短的信号时，直接卷积、Winograd 或专用 SIMD 内核可能更快。

实时长信号常用 overlap-add 或 overlap-save，把输入分块并复用卷积核的 FFT。GPU 上则常把多个块组成 batch，以提高占用率。

8.1. 一个可以手算的卷积例子

取信号 $x = [1, 2, 1]$ 和卷积核 $h = [1, 1]$ 。直接卷积为：

$$y_0 = 1 \times 1 = 1,$$

$$y_1 = 1 \times 1 + 2 \times 1 = 3,$$

$$y_2 = 2 \times 1 + 1 \times 1 = 3,$$

$$y_3 = 1 \times 1 = 1.$$

所以 $y = [1, 3, 3, 1]$ 。如果用 FFT，至少要补零到 $3 + 2 - 1 = 4$ 个点：

序列	补零后的数据	FFT	频域逐点相乘
x	[1, 2, 1, 0]	X	$X_k H_k$
h	[1, 1, 0, 0]	H	Y_k

对 Y_k 做逆 FFT，就会得到同一个 $[1, 3, 3, 1]$ 。这个小例子里 FFT 反而更麻烦；当 x 和 h 都很长时， $O(L \log L)$ 才会超过直接卷积的优势。

9. 能解决哪些实际问题

9.1. 音频与语音

- 频谱、声谱图、音高和谐波分析；
- 均衡器、降噪、混响和长 FIR 滤波；
- 语音识别前端的短时傅里叶变换（STFT）；
- 音频编码中的频域变换。

短时傅里叶变换本质上是“加窗、分帧、逐帧 FFT”。窗口越长，频率分辨率越高，但时间定位越差，这是时频分辨率的基本权衡。

9.2. 图像与视频

- 二维频域滤波、去周期噪声和模糊或锐化；
- 图像配准中的相位相关；
- 大卷积和全景或体数据处理；
- 纹理和周期结构检测。

二维 FFT 通常沿行和列分别做一维 FFT，中间需要转置或处理 stride。图像边界若不处理，会产生环绕卷积；补零或边界延拓可减少这种影响。

9.3. 通信、雷达与控制

- OFDM 调制和解调：IFFT 生成时域符号，FFT 恢复子载波；
- 雷达距离或速度估计和脉冲压缩；
- 频谱监测、信道估计和同步；
- 振动、旋转机械和电力系统的故障诊断。

9.4. 科学计算

- 利用卷积定理加速偏微分方程的谱方法；
- 天文学、地震学和流体模拟中的相关计算；
- 计算成像、MRI 和断层重建中的频域算子；
- 大规模随机过程和周期边界问题。

10. 常见错误与排查方法

10.1. 把 FFT 的幅度当成真实幅值

DFT 输出的幅度通常与 N 成比例。单边幅度谱还需要根据归一化约定对非直流、非奈奎斯特频点乘 2。应先用一个已知频率、已知幅值的正弦波校准代码。

10.2. 忘记直流分量和负频率

X_0 是直流分量，反映信号平均值。对实值信号，负频率不是“额外的物理频率”，而是复数表示中的共轭部分；但在复数基带信号、IQ 数据和通信系统中，正负频率都可能具有直接含义。

10.3. 把补零当成提高分辨率

补零会增加频谱采样点，让峰值位置更容易估计，但不能创造新的观测信息。真正提高频率分辨能力需要更长的有效观测时间，或使用额外的模型估计方法。

10.4. 忽略线性卷积与循环卷积的区别

频域逐点相乘对应的是循环卷积。要得到线性卷积，必须补零到至少 $N + M - 1$ ，否则尾部会绕回开头。

10.5. 只看 GPU kernel 时间

端到端性能应至少包括：数据准备、主机与显存拷贝、plan 创建（如果不是缓存的）、FFT 执行、后处理和同步。只测异步 kernel 提交时间会得到误导性的结果。

11. 一个实用决策流程

面对一个新问题，可以按以下顺序判断：

1. **目标是什么：**找频率、做滤波、做相关，还是计算卷积？
2. **数据是否等间隔采样：**不等间隔数据不能直接套普通 FFT，应考虑重采样、NUFFT 或其他方法。
3. **窗口和采样率是否合适：**先检查混叠、观测时长和窗函数，再谈算法优化。
4. **问题规模和批量多大：**小问题可能用 CPU；大 batch、二维数据或长卷积更适合 GPU。
5. **数据在哪个设备上：**如果后续计算已在 GPU，尽量让 FFT 留在 GPU，减少拷贝。
6. **精度要求是什么：**比较 FP32、FP64、实数 FFT 和复数 FFT 的误差与吞吐。
7. **如何验证：**用直接 DFT 或高精度参考结果检查误差；用已知正弦、冲激和卷积例子检查幅值、相位、归一化和边界。

12. 最小心智模型

可以把 FFT 记成下面这条链：

采样信号 → 加窗或分帧 → FFT → 频率索引与幅相解释 → 滤波、检测或卷积 → 必要时逆 FFT

其中：

- DFT 定义了“频率分解”是什么；
- Cooley-Tukey 通过偶奇分解复用计算；
- 蝶形是局部的加减与旋转；
- GPU 通过并行蝶形、批处理、共享内存和高带宽提升吞吐；
- 采样率、窗口、归一化和边界条件决定结果是否有物理意义。

理解这几层之后，FFT 就不只是一个库函数，而是一套从信号建模、数值计算到硬件执行的完整方法。