2023 年 CCFSys 定制计算挑战赛
CCFSys Customized Computing Challenge 2023
FFT 设计报告

参赛学校：华东师范大学
指导教师：徐飞老师
参赛队员：白卓岩、堵仪萱、徐珑珊

2023 年 7 月 9 日

目录

1 工作概述

本文为 2023 年 CCFSys 定制计算挑战赛的参赛项目设计报告。小组选择的是 FFT 赛题，赛题的完成情况如表 1所示，包含以下部分：

- 根据对基础非递归的 FFT 算法和 AIE 的理解，使用 AIE API 完成了对 1,024 点 cint16、cfloat 小规模 FFT 算法的单核设计、优化和仿真。这一部分使用了循环优化技术，并利用 ping-pong buffer 来优化访存。
- 基于分布式的 FFT 算法，探索 AIE kernel 之间的连接，扩展更大点数（4K、8K）的 cint16 数据类型的 FFT 实现。利用 AIE 提供的 sliding multiplication 方法优化矩阵向量的乘累加操作。
- 实现 PL 和 AIE 的数据连接，在 Vitis IDE 上尝试了系统级仿真，可以在 VCK5000 硬件上运行。

表 1: 赛题完成情况

点数	AIE 数量	数据类型	twiddle 类型	AIE 运行时间（us）
1,024	1	cint16	cint16	11.072
		cfloat	cfloat	30.033
4,096	5	cint16	cint16	14.275
8,192	9	cint16	cint16	20.661

2 算法设计

傅里叶变换（Fourier transform，FT）是一种重要的数字信号处理技术。傅里叶变换可以将时域信号转换至频域，以便查看不同频率下信号的强度，实现去除低频、高频噪声信号等功能，在音频、图像、雷达、通信等领域中都有着广泛的应用。

快速傅里叶变换（fast Fourier transform，FFT）是一种优化算法，可以高效计算傅里叶变换，将点数为 N 的计算复杂度从 $O(N^2)$ 降低到 $O(N\log N)$。FFT 在具体实现上可以采用不同的优化方案，例如使用分治算法将一个大规模的傅里叶变换分解成若干个小规模的傅里叶变换进行并行计算，这使得 FFT 在实际应用中可以高效地实现。

本节介绍本项目所采用的 FFT 算法设计。第 2.1 小节介绍 FFT 的非递归实现方法，用于计算基础的 1024 点的 FFT。第 2.2 小节介绍分布式的 FFT 实现方法，侧重于点数规模的扩展。

2.1 非递归 FFT

2.1.1 FFT 基本原理

傅里叶变换的基本计算公式为：

$$Z_k = \sum_{n=0}^{N-1} \omega_N^{nk} x_n, \quad k = 0, 1, \cdots, N. \tag{1}$$

其中，N 为采样点数，ω_N^{nk} 叫做旋转因子（twiddle factor），可看作频域下的基。该式子的作用就是通过线性变换，将时域下的采样点转换至频域下。由于傅里叶变换所做的是线性变换，其公式也可以写成矩阵向量乘法的形式，即：

$$\begin{bmatrix} Z_0 \\ Z_1 \\ \vdots \\ Z_{N-1} \end{bmatrix} = \begin{bmatrix} \omega_N^0 & \omega_N^0 & \cdots & \omega_N^0 \\ \omega_N^0 & \omega_N^1 & \cdots & \omega_N^{N-1} \\ \vdots & \vdots & \ddots & \vdots \\ \omega_N^0 & \omega_N^{N-1} & \cdots & \omega_N^{(N-1)^2} \end{bmatrix} \cdot \begin{bmatrix} x_0 \\ x_1 \\ \vdots \\ x_{N-1} \end{bmatrix}. \tag{2}$$

可以看出该矩阵向量乘法的计算复杂度为 $O(N^2)$。而 FFT 利用了两个技巧，将傅里叶变换的复杂度降低为 $O(N \log N)$。

技巧 1. twiddle factors 均匀分布在复平面单位圆上。

如图 1所示，当 $N = 8$ 时，8 个 twiddle factor 均匀地落在复平面的单位圆上，它们的模长均为 1。

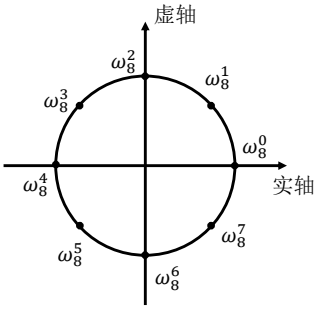


图 1: 复平面单位圆

根据均匀分布的特点，可以得到 twiddle factor 的三个性质：

- 性质一：$\omega_{2N}^{2k} = \omega_N^k$；
- 性质二：$\omega_N^{k+N} = \omega_N^k$；
- 性质三：$\omega_N^k = -\omega_N^{\frac{N}{2}+k}$。

从图 1中可以看出，twiddle factor 的幂具有规律性，随着幂的增加，是呈逆时针均匀分布的，因此可得性质一。例如，考虑 $N = 4$ 的情况，从图上易知 $\omega_4^0 = \omega_8^0$、$\omega_4^1 = \omega_8^2$、$\omega_4^2 = \omega_8^4$、$\omega_4^3 = \omega_8^6$。同理，幂乘 N 次，也就是逆时针旋转 N 后又会回到原来的位置，取值不变，可得性质二。此外，对角线上的两个 twiddle factor 符号相反，因此可得性质三。

技巧 2. Z_k 可以分成奇偶两部分。

对于

$$Z(y) = x_0 y^0 + x_1 y^1 + x_2 y^2 + \cdots + x_{N-1} y^{N-1}, \quad y = \omega_N^0, \omega_N^1, \cdots, \omega_N^{N-1}, \tag{3}$$

可以将它分为偶部 $Z_e(y)$ 和奇部 $Z_o(y)$，

$$
\begin{aligned}
Z_e(y) &= \sum_{j=0}^{\frac{N}{2}-1} x_{2j} y^j = x_0 y^0 + x_2 y^1 + \cdots + x_{N-2} y^{\frac{N}{2}-1}, \\
Z_o(y) &= \sum_{j=0}^{\frac{N}{2}-1} x_{2j+1} y^j = x_1 y^0 + x_3 y^1 + \cdots + x_{N-1} y^{\frac{N}{2}-1}.
\end{aligned} \tag{4}
$$

那么

$$
Z(y) = Z_e(y^2) + y Z_o(y^2). \tag{5}
$$

根据 twiddle factor 的性质一可得，当 $k < \frac{N}{2}$ 时，

$$
Z_e(y^2) = Z_e(\omega_N^{2k}) = Z_e(\omega_{\frac{N}{2}}^k), \quad Z_o(y^2) = Z_o(\omega_N^{2k}) = Z_o(\omega_{\frac{N}{2}}^k), \tag{6}
$$

所以

$$
Z(\omega_N^k) = Z_e(\omega_{\frac{N}{2}}^k) + \omega_N^k \cdot Z_o(\omega_{\frac{N}{2}}^k). \tag{7}
$$

再结合性质二和三可得

$$
\begin{aligned}
Z(\omega_N^{k+\frac{N}{2}}) &= Z_e(\omega_N^{2k+N}) + \omega_N^{k+\frac{N}{2}} \cdot Z_o(\omega_N^{2k+N}) \\
&= Z_e(\omega_{\frac{N}{2}}^k) + \omega_N^{k+\frac{N}{2}} \cdot Z_o(\omega_{\frac{N}{2}}^k) \\
&= Z_e(\omega_{\frac{N}{2}}^k) - \omega_N^k \cdot Z_o(\omega_{\frac{N}{2}}^k).
\end{aligned} \tag{8}
$$

于是，原问题 $Z(\omega_N^k), k = 0, \cdots, N-1$ 被拆分成了两个规模缩小了一半的子问题 $Z_e(\omega_{\frac{N}{2}}^k)$ 和 $Z_o(\omega_{\frac{N}{2}}^k), k = 0, \cdots, \frac{N}{2}-1$，实现了加速。根据上述公式，使用递归实现 FFT 是比较直观的，但递归过程开销较大，所以本项目采用非递归的方式来实现。

2.1.2 Shuffle 操作

在执行 FFT 时，需要根据系数下标的奇偶性将系数分成两组。以 $N = 8$ 的情况为例，当拆分到边界为 1 的情况时直接返回，那么奇偶分组的情况如表 2所示。

表 2: FFT 奇偶下标分组情况（$N = 8$）

初始下标	0	1	2	3	4	5	6	7
第一轮递归	0	2	4	6	1	3	5	7
第二轮递归	0	4	2	6	1	5	3	7
第三轮递归	0	4	2	6	1	5	3	7

为了以非递归的方式实现 FFT，首先需要把系数放置到递归情况下最后一轮的位置上，这一步叫做 shuffle 操作，再向上还原。本项目中 shuffle 操作的具体实现方式在第 3.1.1小节中介绍。

2.1.3 Butterfly 操作

在非递归的 FFT 实现中，除 shuffle 外另一个重要的操作就是 butterfly 操作。它将一个长度为 N 的计算分解为两个长度为 $N/2$ 的计算，并将它们组合起来计算出原始的值。该操作是 FFT 算法中实现分治策略的关键步骤。仍旧以 $N = 8$ 的情况为例，在 shuffle 完毕后，自底向上的部分 butterfly 操作如图 2所示。

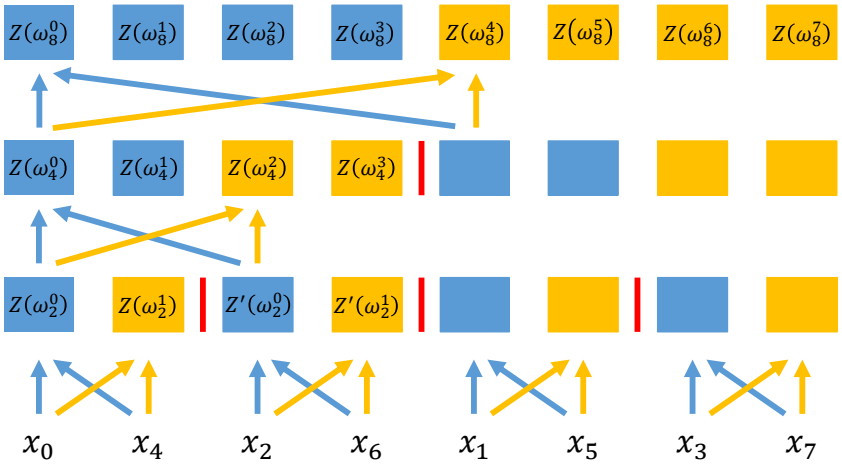


图 2: 蝴蝶操作示意图

图 2第三行 $Z(\omega_2^0)$ 块和 $Z(\omega_2^1)$ 块的计算公式分别为

$$\begin{aligned} Z(\omega_2^0) &= Z_e(\omega_1^0) + \omega_2^0 \cdot Z_o(\omega_1^0) = x_0 + \omega_2^0 \cdot x_4, \\ Z(\omega_2^1) &= Z_e(\omega_1^0) - \omega_2^0 \cdot Z_o(\omega_1^0) = x_0 - \omega_2^0 \cdot x_4. \end{aligned} \tag{9}$$

又因 $\omega_2^0 = (\omega_4^0)^2$，且 $Z'(\omega_2^0) = x_2 + \omega_2^0 \cdot x_6$，可以推至第二行的 $Z(\omega_4^0)$ 块，计算公式为

$$Z(\omega_4^0) = Z(\omega_2^0) + \omega_4^0 \cdot Z'(\omega_2^0) = x_0 + \omega_4^0 \cdot x_2 + (\omega_4^0)^2 \cdot x_4 + (\omega_4^0)^3 \cdot x_6. \tag{10}$$

采用这种迭代的方式所得到的数据下标（0、2、4、6）和表 2中第一轮递归的情况是相符的。从图 2可以看出，计算的数据依赖呈蝴蝶形，按照这种模式计算，就可以求得完整的 FFT。本项目实现的 butterfly 操作以及相关优化见第 3.1.1小节。

2.2 分布式 FFT

分布式 FFT 采用 Cooley-Tukey 技巧将 N-point FFT 分解为 Q-point 和 M-point FFT，给定 FFT 的点数为 $N = Q \times M$，N-point FFT 可以被划分为 Q 个 M-point FFT，对所得结果进行线性相位旋转后，计

算 M 个 Q-point FFT 得到最终结果 ~ [1]。更具体地说，可以将原傅立叶变换公式写为如下形式。

$$\begin{aligned} y_{uM+v} &= \sum_{k=0}^{QM-1} x_k e^{-j\frac{2\pi(uM+v)k}{QM}} = \sum_{q=0}^{Q-1}\sum_{m=0}^{M-1} x_{mQ+q} e^{-j\frac{2\pi(uM+v)(mQ+q)}{QM}} \\ &= \sum_{q=0}^{Q-1} e^{-j\frac{2\pi q}{MQ}v} \left(\sum_{m=0}^{M-1} x_{mQ+q} e^{-j\frac{2\pi mv}{M}} \right) e^{-j\frac{2\pi qu}{Q}} \end{aligned} \tag{11}$$

可以看出，括号内的求和表示对输入数据进行 M-point FFT 变换，而括号外的求和则表示进行 Q-point FFT 变换。这两个变换之间存在一个线性相位旋转，其相位增量为 $-2q/N$，其中 q 代表相位指数。

故我们可以采取以下步骤来进行 N-point FFT 计算。首先，在第一阶段中，我们分配 Q 个处理器，每个处理器计算一个 M-point FFT，并对结果进行线性相位旋转运算。在第二阶段中，收集第一阶段中的 Q 个处理器的计算结果，并对其进行 M 个 Q-point FFT 变换。现在，我们将以矩阵向量乘法的形式来详细解释该算法的执行过程:

1）将输入信号 X 分为 $X_1^{(\text{row})}$ 到 $X_Q^{(\text{row})}$，分别传入 Q 个处理器进行 M-point FFT 运算：

$$X = \begin{bmatrix} x_1 & x_{Q+1} & \cdots & x_{(M-1)Q+1} \\ \vdots & \vdots & \ddots & \vdots \\ x_Q & x_{2Q} & \cdots & x_{QM} \end{bmatrix} = \begin{bmatrix} X_1^{(\text{row})} \\ \vdots \\ X_Q^{(\text{row})} \end{bmatrix} \tag{12}$$

2）在每个处理器中进行 M-point FFT 的计算得到 $Y_i^{(\text{row})} = X_i^{(\text{row})} F_M$，$i = 1, \cdots Q$

3）调整对 Y 进行转置以进行第二阶段运算：

$$Y = \begin{bmatrix} Y_1^{(\text{row})} \\ \vdots \\ Y_Q^{(\text{row})} \end{bmatrix} = \begin{bmatrix} Y_1^{(\text{col})} & \cdots & Y_M^{(\text{col})} \end{bmatrix} \tag{13}$$

4）在每个处理器中将 Y_i 乘以对应的 twiddle factor：

$$Y_i^{(\text{row})} = T_{N,i}^{(\text{row})} \circ Y_i^{(\text{row})} \tag{14}$$

其中 $\circ$ 表示对应元素相乘，twiddle factor 矩阵为

$$\begin{aligned} T_N &= \begin{bmatrix} \omega_N^0 & \omega_N^0 & \cdots & \omega_N^0 \\ \omega_N^0 & \omega_N^1 & \cdots & \omega_N^{M-1} \\ \vdots & \vdots & \ddots & \vdots \\ \omega_N^0 & \omega_N^{Q-1} & \cdots & \omega_N^{(Q-1)(M-1)} \end{bmatrix} \\ &= \begin{bmatrix} T_{N,1}^{(\text{col})} & \cdots & T_{N,M}^{(\text{col})} \end{bmatrix}. \end{aligned} \tag{15}$$

5）在第 $Q+1$ 个处理器中计算出最终结果 Z：

$$Z_i^{(\mathrm{col})} = F_Q Y_i^{(\mathrm{col})} \ , \ i = 1 \ldots M \tag{16}$$

同样地，对于本项目中 FFT 的实现，我们将其分为两个阶段进行计算。具体而言，当 $N = 4K$ 时，如图 3所示，第一阶段（stage-1）将 FFT 输入分为四个部分，并以并行方式输入到四个 Al Engines。每个 Al Engine 对数据执行 1024-point FFT，并随后进行线性相位旋转。在第二个阶段（stage-2），数据被输出到第五个 Al Engine 进行 4-point FFT 计算。

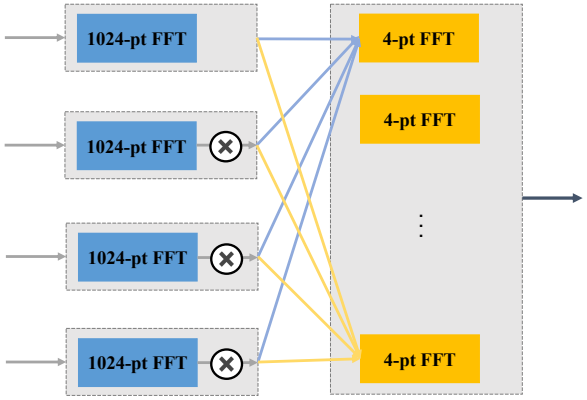


图 3: 分布式 4K-point FFT

类似地，当 $N = 8K$ 时，如图 4所示，FFT 输入将分为 8 个部分，并以并行方式进入 8 个 Al Engines。Al Engine 对数据执行 1024-point FFT，然后进行线性相位旋转，并将数据输出到第 9 个 Al Engine 进行 8-point FFT。

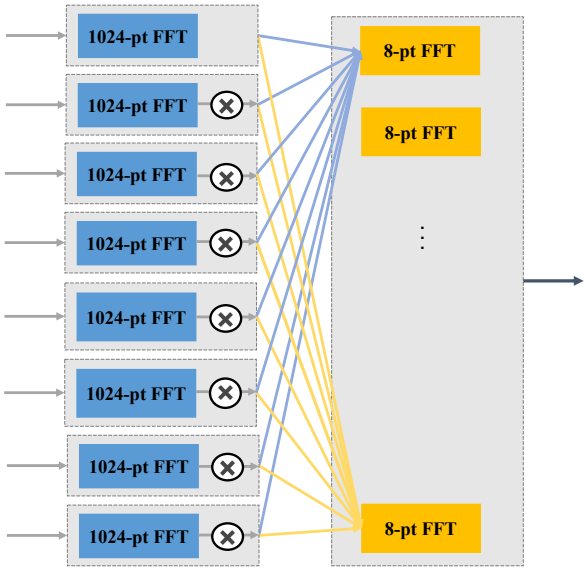


图 4: 分布式 8K-point FFT

3 具体实现

本项目的核心在于优化 AIE 上 FFT 的设计和实现，这一部分将在第 3.1 小节介绍。另外，也尝试了连接 PL 和 AIE，从 host 端进行调用，在 VCK5000 上运行任务，见第 3.2 小节。

3.1 AIE 实现

在 AIE 中实现 4K-point 或 8K-point FFT，主要过程分为两个阶段。第一阶段涉及多个 kernel 并行计算 1K-point FFT，并在计算完成后对结果进行相位旋转，然后将其输入到第二阶段。下文将在第 3.1.1节详细介绍这一阶段。在第二阶段中，我们将利用 AIE 支持的 sliding multiplication 多通道乘法来加速计算 1,024 个 4-point FFT 或 8-point，具体内容参见第 3.1.2节。在计算 1K-point FFT 时，主要涉及到 shuffle 操作和蝶形操作，这些操作将在第 3.1.1.1和第 3.1.1.2节中进行详细介绍。

3.1.1 1K-point FFT（stage-1）

1K-point FFT 为一个基本的计算 kernel，输入和输出都采用 buffer 的方式。该 kernel 首先需要对输入信号进行位置交换，然后用矩阵乘法完成 8-point FFT 的计算，再进行 16-point 到 1024-point 的 bufferly 操作，共 7 次，最后将结果乘以对应的 twiddle factor。为了提高访存效率，该 kernel 将输入和输出 buffer 作为 ping-pong buffer 的方式，以进行上述每一次操作计算结果的读入和写出。

3.1.1.1 Shuffle 本项目的 1K-point FFT 计算采用非递归的 FFT 算法实现，从 8-point FFT 向上迭代。该方法首先需要将输入信号的顺序进行交换，以便后续的迭代计算。为了完成交换步骤，首先要利用递归实现的 FFT 迭代到 8-point 来获取 8-point FFT 开始时各输入值的位置，然后确定如何把输入信号通过交换转变为 8-point FFT 开始时信号序列，需要交换位置的信号值可以分为多个最小的序列组，每组分别进行交换即可完成整个序列的交换。例如第 1 个值最终交换到第 512 个，第 512 个到第 4 个，第 4 个到第 128 个，第 128 个到第 1 个，则第 1、512、4 和 128 个值可分为一个长度为 4 的组进行交换。经过分析，4,096 个输入值一共分为 120 个长度为 2 的组和 384 个长度为 4 的组。在代码实现方面，可以通过两个循环实现长度为 2 和 4 的序列组的交换，同时由于每个组的数据中间没有重复，因此每个迭代间也没有数据依赖，可通过 unroll 来提高循环的并行度。

3.1.1.2 Butterfly 如图 2所示，在传统快速傅立叶变换的实现中，根据傅立叶变换的特殊性质，可将长度为 N 的离散傅立叶变换（discrete Fourier transform，DFT）递归地表述为两个大小为 $N/2$ 的离散傅立叶变换，直至 N 等于 2。通过重用中间计算结果，两个大小为 $N/2$ 的傅立叶变换结果可以同时得到。从而将傅立叶变换的计算复杂度由 $O\left(n^2\right)$ 降低为 $O\left(n \log n\right)$。

在本实现中，为避免大量的函数调用及存储开销，我们将递归操作修改为非递归实现，同时如图 5所示以 8-point FFT 为底开始运算。即首先采用传统傅立叶计算方式（矩阵运算）计算 8-point FFT，再在此计算结果上进行多次蝶形运算的迭代。虽然表面上看 8-point FFT 的计算复杂度有所提升，但将标量运算转化为矢量计算，能够更好地利用 AIE 的单指令多数据流（Single Instruction/Multiple Data，SIMD）特

性，实现对各个数据元素的并行处理，从而大大提升处理速率。同时，选择 8-point 为底具有可接受的存储开销及时间复杂度。

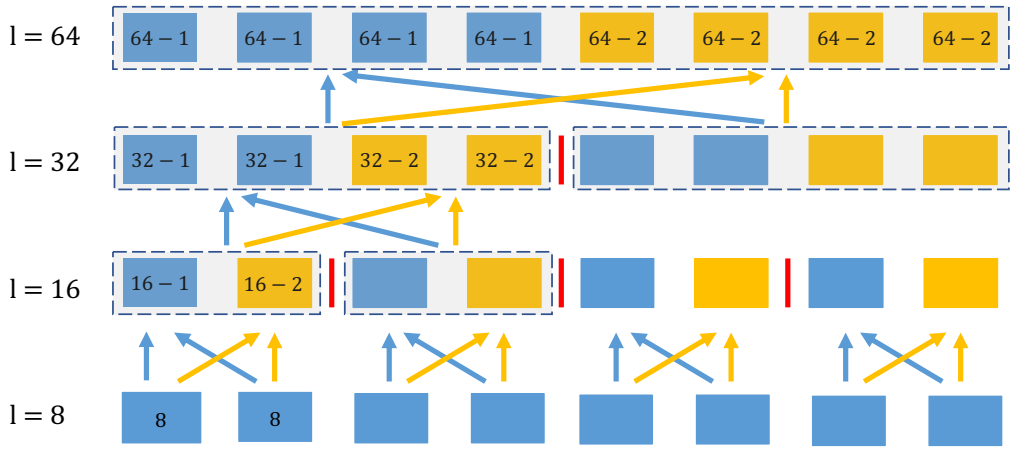


图 5: butterfly 操作在 AIE 里的实现

图 5列出了执行 8-point FFT 计算后，自底向上进行 butterfly 操作的过程。在进行多次蝶形运算的迭代过程中，为充分利用 AIE 的并行计算能力，我们将尽可能多地载入矢量数据进行运算。由图可知，当 l=16 时，我们使用长度为 8 的矢量作为计算单元；当 l=32 时，我们使用长度为 16 的矢量作为计算单元；由于受到矢量寄存器的位宽限制，当 l 大于等于 64 时，我们均使用长度为 32 的矢量为计算单元。此时数据的读入与读出将成为主要的性能瓶颈，为避免读入数据的地址空间不连续所导致的读入耗时过长，我们以可接受的存储冗余开销换取读入时间的大幅度减少。具体而言，我们分别存储了 16-256 以及 1024-point 下所需的 Omega 数组，从而使得蝶形运算过程的访存和计算过程达到相对平衡。

同时，在该过程中我们借鉴了 Ping-Pong Buffer [2] 的思想，如图 6所示，主要考虑避免在一个循环过程中，数据的读取与写入在同一块地址空间中进行，以避免数据冲突和处理延迟。为此，我们利用了两个独立的 buffer，它们在操作过程中会轮流切换角色，即在一个时间片段内，一个 buffer 负责数据的读取，而另一个 buffer 则负责数据的写入。在下一个时间片段，它们的角色互换。该策略大大提高了数据处理的效率。

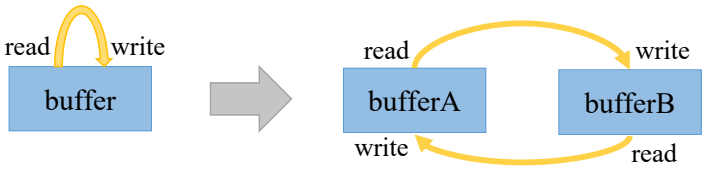


图 6: ping-pong buffer 优化

3.1.1.3 线性相位旋转操作 在前 4 或 8 个 kernel 分别完成 1024-point FFT 后，相应 kernel 需将所得结果分别与对应的 twiddle factor 相乘，之后由最后一个 kernel 汇总所有数据并计算 1,024 个 4-point FFT。

同样，在此过程中为充分利用 AIE 的并行计算能力，我们也尽可能多地载入矢量数据进行运算，使得 32 个数据同时进行乘操作。

3.1.2 4/8-point FFT（stage-2）

4K-point FFT 和 8K-point FFT 的最后一步是将前 4 或 8 个 1K-point FFT 的结果进行组合，并完成 1024 个 4-point FFT 或 8-point FFT。由于点数较小，可直接采用矩阵相乘的方式完成 FFT 的计算。下面以 4K-point FFT 为例介绍 Stage-2 FFT 的具体步骤：

首先，每次从每个输入 buffer 取出 8 个元素组成一个 4×8 的矩阵 D，如图 7，取 8 个元素是为了使该矩阵达到 cint16 类型 vector 的最大长度。

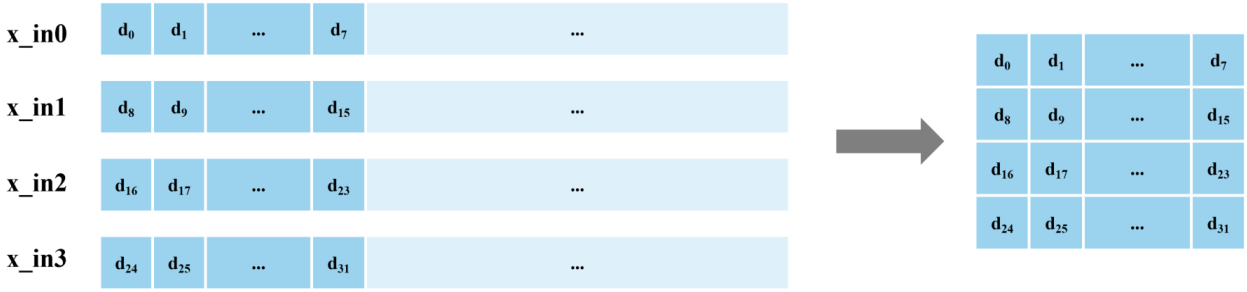


图 7: Stage-2 FFT 读取操作数

用 4×4 的 Omega 矩阵与 D 的每一列相乘即可完成 8 次 4-point FFT 的计算。为了提高计算的并行性，本设计采用 sliding multiplication 的多通道乘法进行计算，AIE 为这种计算模式提供硬件支持来实现加速，它允许多通道并行进行乘累加（multiply accumulate，MAC）操作，符合矩阵乘法中的计算。通过图 8的参数设置，即可将矩阵乘法转换为 sliding multiplication，如图 9。

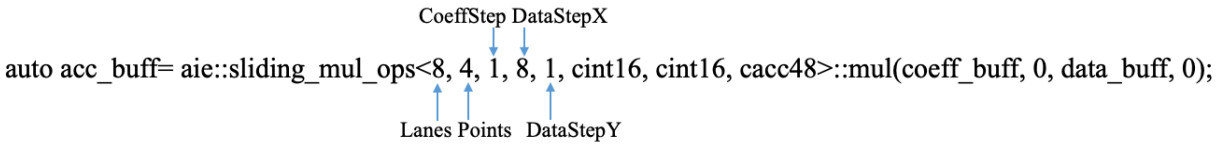


图 8: Sliding Multiplication 参数配置

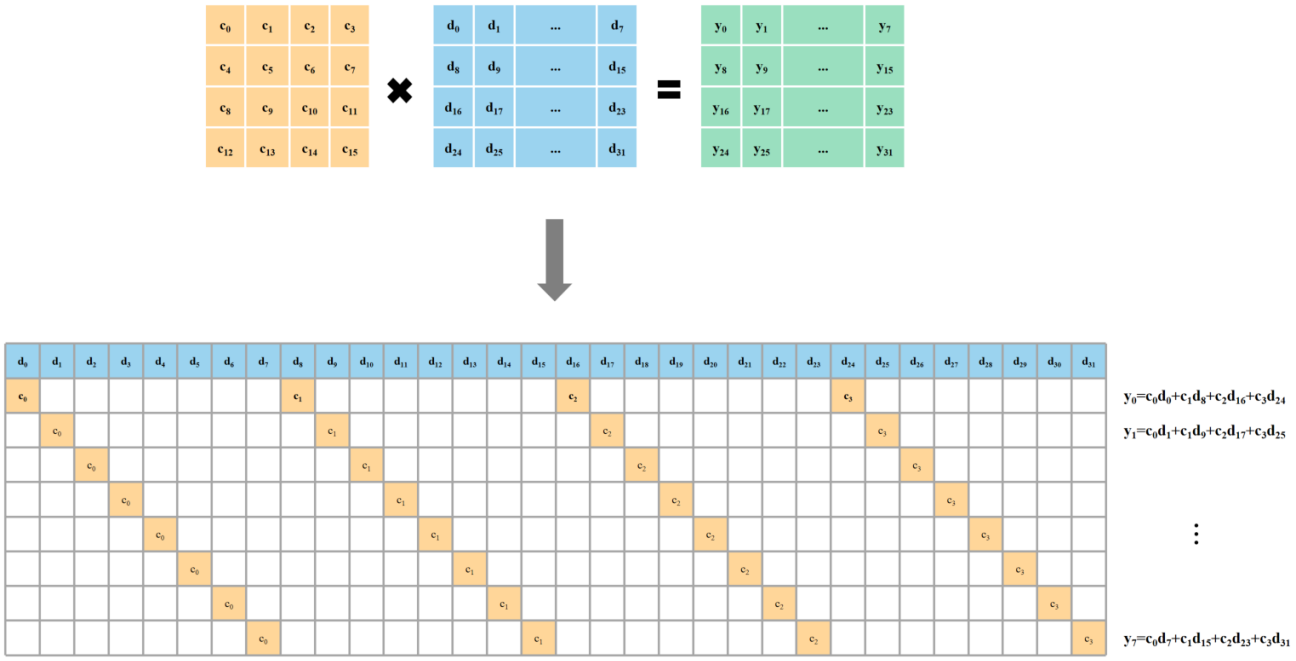


图 9: 矩阵乘法转 Sliding Multiplication

由计算过程可知，一次 sliding multiplication 操作可完成 32 次乘累加操作，并得到 8 个输出作为 4096-point FFT 的最终结果。在后续操作中，只需更新四次系数（coefficient）值，即可得到 32 个输出结果。不断重复操作（循环 128 次），即可得到所有输出结果。经检验，该操作可大大降低运算时长，进而也体现了 AIE 强大的并行运算能力。

3.1.3 4K-point FFT Graph 及 Array 布局

本项目一共使用 5 个 kernel 实现 4K-point FFT，其中 4 个 kernel 用于 1K-point FFT 以及与 twiddle factor 相乘的计算，1 个 kernel 用于 1,024 个 4-point FFT 的计算。如图 10，四个计算 1K-Point FFT 的 kernel 分别为 fft0、fft1、ff2 和 fft3，都通过 buffer 进行输入输出数据的传输，每组输入数据是从原长度为 4,096 的输入信号中每隔 4 个取一个组成。当这 4 个 kernel 将计算结果都写入输出 buffer 后，用于计算 1,024 个 4-point FFT 的 kernel fft_stage2 开始进行计算，并将以 4 个长度为 1024 的结果作为输出，这 4 个输出按顺序拼接即可得出长度为 4096 的结果。

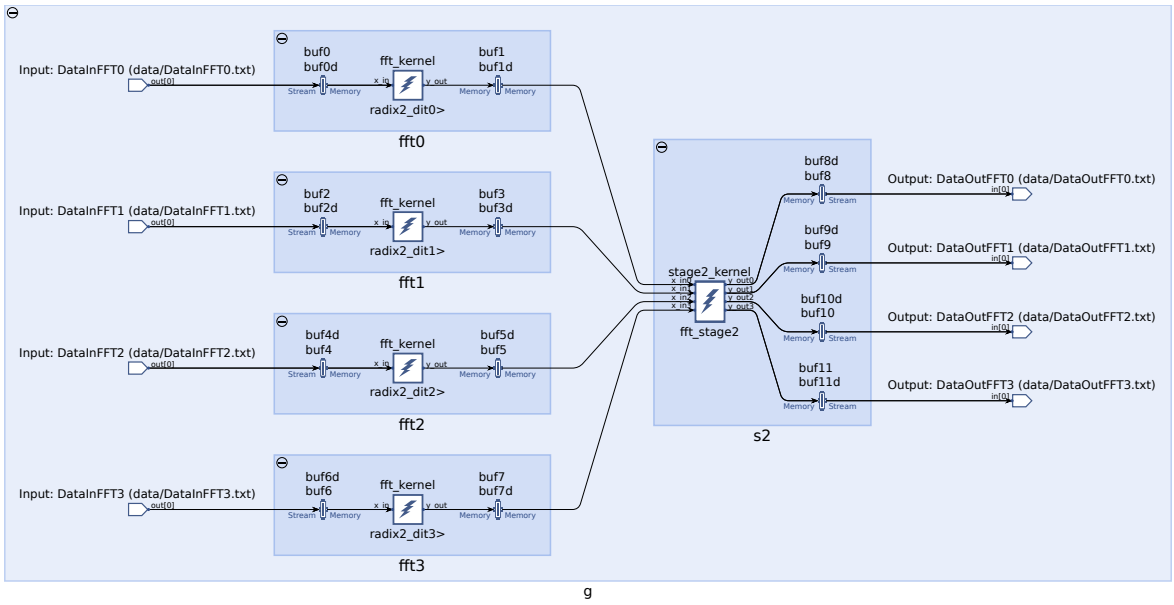


图 10: 4K-point FFT Graph

4K-point FFT 的 Array 布局如图 11所示，stage2-FFT kernel 位于 tile(23, 0)，4 个 1K-point kernel 环绕在 stage2-FFT kernel 周围，分别位于 tile(22, 0)、tile(22, 1)、tile(23, 1) 以及 tile(24, 0)，以便于 stage2-FFT kernel 从本地和相邻的 memory 中读取 4 个 1K-point kernel 的输出，同时 stage2-FFT kernel 也更加靠经 PL 端，减少数据传输的时延。

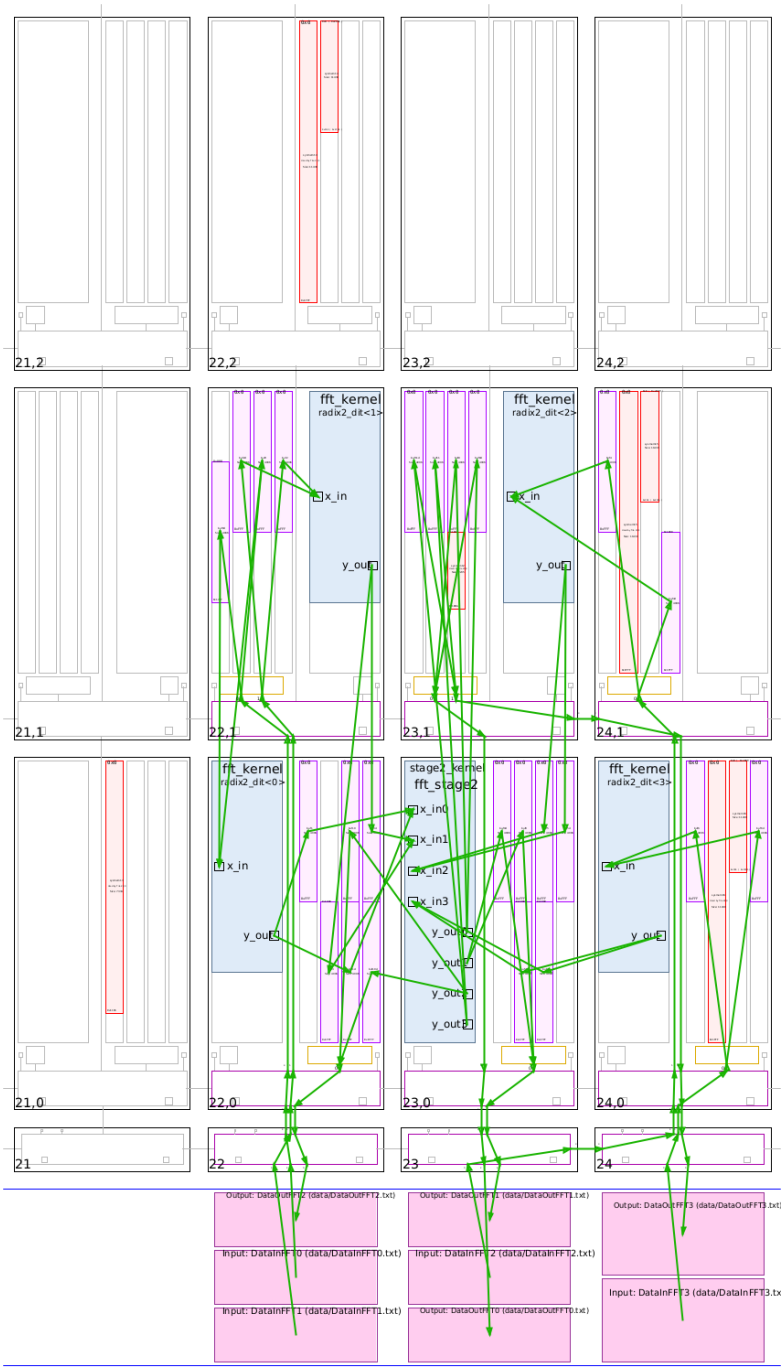


图 11: 4K-point FFT Array 布局

3.1.4 8K-point FFT Graph 及 Array 布局

8K-point FFT 的 graph 结构与 4K 的 graph 相似，如图 12，该 graph 先由 8 个 1K-point kernel 做第一步计算，然后将 8 个输出传入 stage2 kernel 进行 1,024 个 8-point FFT 计算。不同点在于第 6 和 7 个 1K-point kernel 的输出后分别有两个 buffer，因为 memory 空间有限导致第一个 buffer 的位置无法与

stage2 kernel 相邻，需要将其拷贝到与 stage2 kernel 相邻的第二个 buffer。此外，由于 8K-point 的输出数据量较大，stage2 kernel 的输出采用 stream，以节省 memory 空间，输出结果需要重新排列得出最终结果。

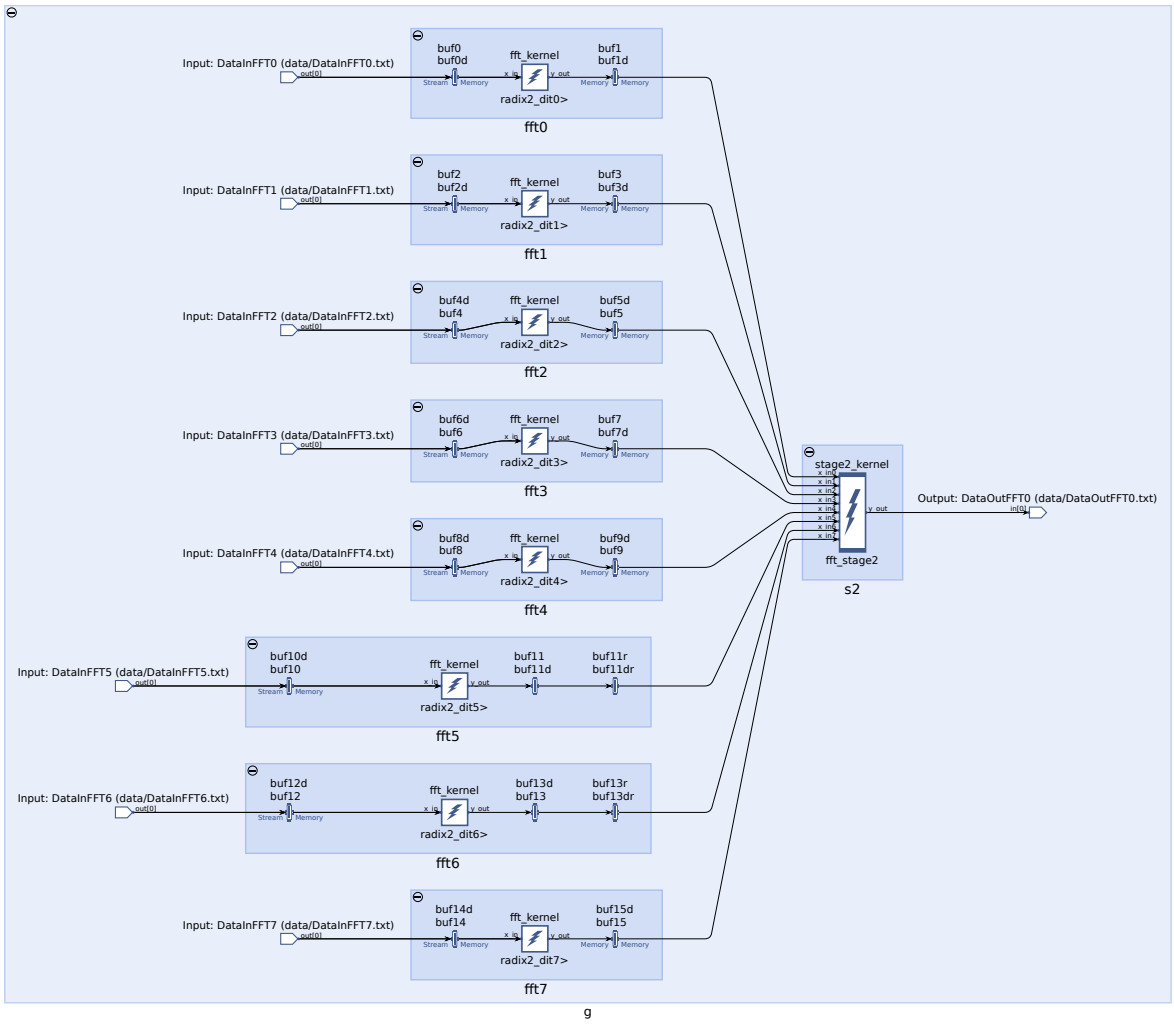


图 12: 8K-point FFT Graph

在 Array 的布局中，9 个 kernel 被放置在一个 9× 9 的 tile 阵列中，stage2 kernel 被放置在阵列中心 tile(23, 1)，如图 13，以减少访问输入 buffer 的时延。第一个 1K-point FFT kernel 被置于 tile(23, 0)，该 kernel 中的 twiddle factor 全为 1 无需存入 memory，因此可以放置在没有相邻空闲 tile 的位置。

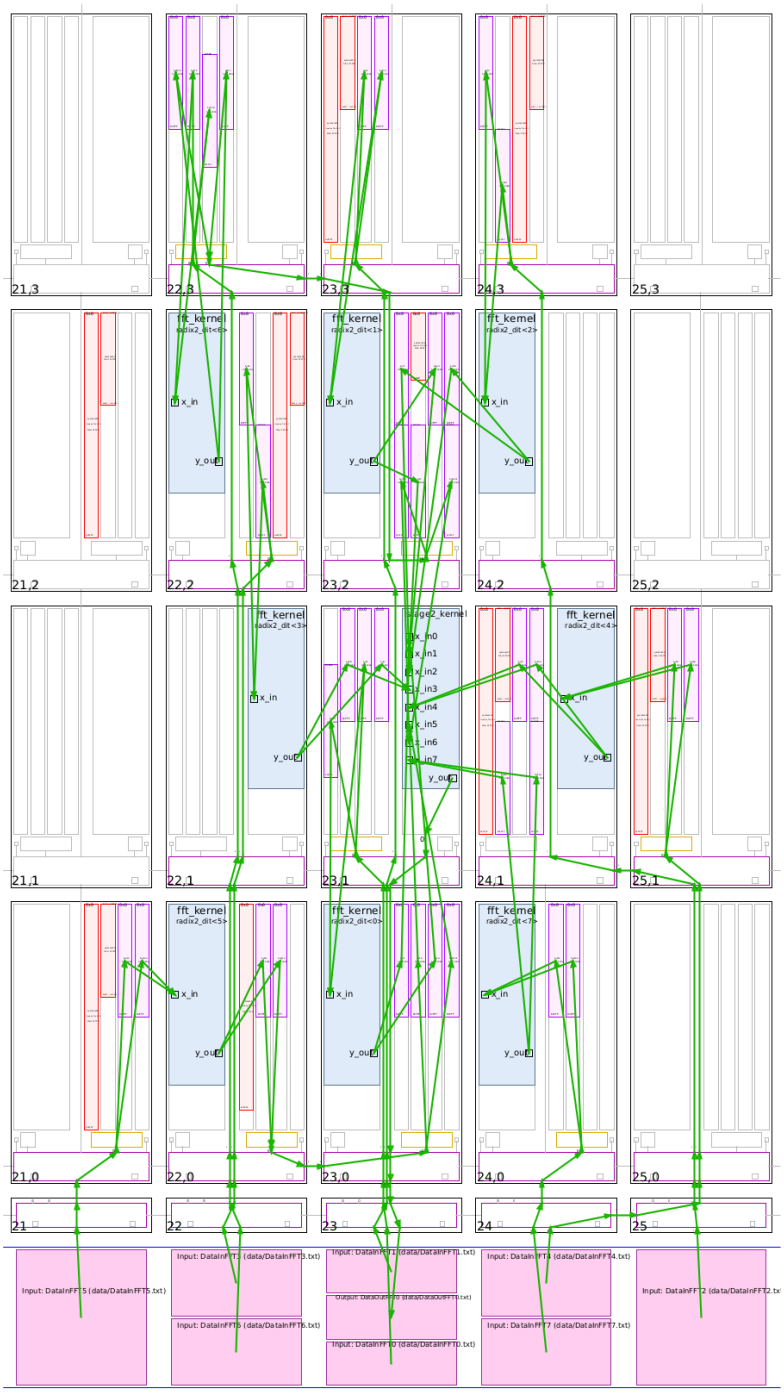


图 13: 8K-point FFT Array 布局

3.2 PL 实现

在 AIE 设计优化的基础上，本项目还尝试了 PL 和 AIE 的数据连接，使得系统可以从 host 端调用，在 VCK5000 板卡上运行。

3.2.1 PL 和 AIE 的数据传输

PL 和 AIE 使用 hls::stream 接口进行数据传输，传输的数据单元位宽是 128 字节。PL 端使用了两个 data mover，mm2s 负责将输入数据从缓冲区通过 stream 接口传输给 AIE，s2mm 负责将 AIE 的输出数据写入缓冲区。数据传输使用了 HLS 的 pipeline pragma 进行优化。

以 4K-point FFT 为例，根据 AIE 的 graph 设计，PL 端的 kernel 连接情况如图 14 所示。

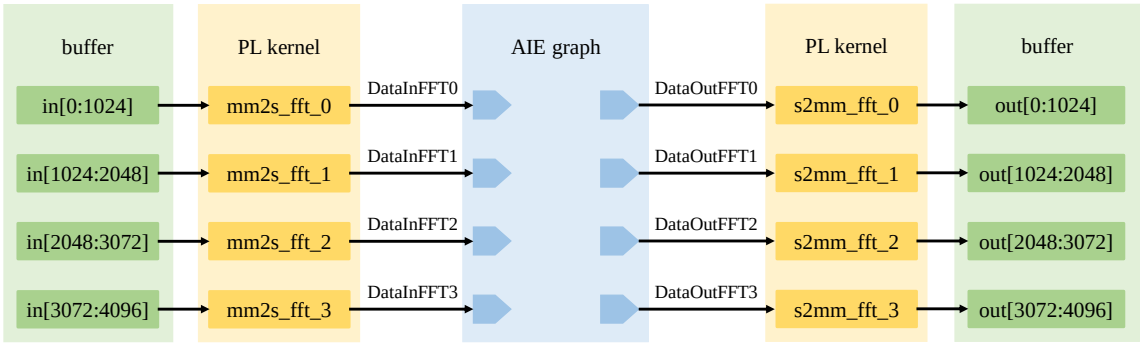


图 14: PL 和 AIE 的连接

3.2.2 Host 应用

Host 应用负责从主机端调用 PL 和 AIE。本项目使用 Xilinx runtime library（XRT）提供的 API 编写主机代码，从而和板卡进行交互。Host 应用的工作流程如下：

1. 获取设备下标，从主机中加载.xclbin 文件和输入文件；
2. 开启计时器；
3. 构建 PL kernel；
4. 为输入数据分配空间，将数据写入计算单元缓冲区，将输入缓冲区的数据与设备全局内存进行同步；
5. 执行计算单元，等待执行完毕；
6. 将输出缓冲区的数据与主机进行同步，将输出缓冲区的数据读取到本地缓冲区中；
7. 停止计时器；
8. 将输出数据写入主机文件中，输出运行时间。

3.2.3 运行结果

完成 PL 和 host 之后，可以利用 Vitis IDE 完成系统级的硬件仿真，本项目最终使用 Makefile 来构建可以在 VCK5000 板卡上运行的完整工程。构建完毕得到.xclbin 文件和.exe 文件后，即可将任务提交到 VCK5000 计算节点。图 15 展示了在 VCK5000 上运行 8K-point FFT 的输出结果。使用 C++ chrono 库提供的计时接口记录得到的 8K-point FFT 运行时长（不包括 host 端的文件 IO）为 8,005 微秒。

```
abuqiqi@hacc-vck5000-3:~/CCC2023/sources/fft_8k/execution$ ./host.exe
Load the point size 8*1024
Open the device
Load the xclbin ./fft.xclbin
TEST PASSED (8005 us)
```

图 15: 运行 8K-point FFT 的主机端输出结果

4 性能分析

由于本项目 FFT 的设计与优化主要针对的是 AIE，而 PL 是实现数据传输，本小节的展示与分析侧重 AIE 部分。此外，在硬件上运行时，在 host 应用中测量了程序除文件 IO 以外部分的运行时长，见图 15。

4.1 实验设置

运行环境

本项目的运行环境如下：

- xilinx_vck5000_gen4x8_qdma_2_202220_1
- XRT 2.14.384
- Vitis 2022.2
- VCK5000

代码仓库

本次参赛最终提交的代码仓库为 `https://github.com/abuqiqi/CCC2023/tree/main`。本仓库中包含了 4K-point 和 8K-point 的 FFT 完整工程代码，包括 AIE、PL、host、可执行文件等，代码运行的说明详见 README.md。

此外，本次参赛过程中完成的各版本代码的存档位于该仓库 backup 分支的 sources 文件夹下，分支中的 README.md 对不同版本实现的功能做了简要的说明。第 4.2 小节将对这些版本实现的 AIE 性能进行分析。

4.2 AIE 性能分析

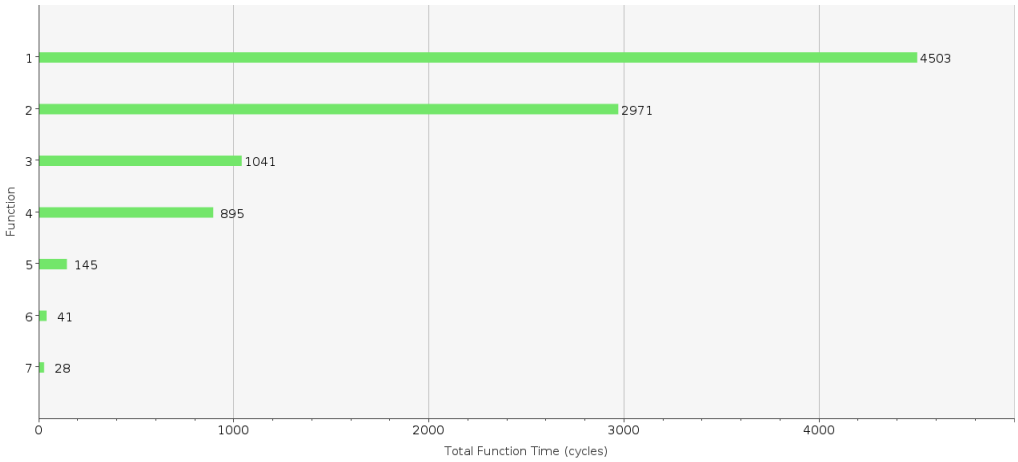
与上述 backup 分支中的提交相匹配，表 3汇总了本次参赛过程中实现的各版本代码的性能情况。其中，fft_1k_pl 在 fft_v2_cint16_aie 的 AIE 设计基础上增加了 PL，因此单独对 AIE 进行仿真的执行时间是相同的。在本节中将进一步分析 fft_1k_pl、fft_4k_v3_pl、fft_8k，也就是点数为 1K、4K、8K 下运行速度最快的 AIE 实现。

表 3: 不同版本 FFT 实现的 AIE 运行结果

工程名	点数	数据类型	AIE 执行时间（us）
fft_v1_cfloat_aie	1K	cfloat	30.033
fft_v2_cfloat_aie			36.691
fft_v2_cint16_aie		cint16	11.072
fft_1k_pl			
fft_4k_v1_pl	4K		24.542
fft_4k_v2_pl			18.591
fft_4k_v3_pl			14.275
fft_8k	8K		20.661

4.2.1 1K-point FFT 性能分析

1K-point FFT 在 AIE Simulation 中的运行总时间为 11.072us，各函数的执行时钟周期数如图 16，其中 memory stall 共 65 次，占 0.1032us，lock stall 共 1 次，占 0.712800us，stall 的时间占总时间的 7.37%。该 graph 的资源使用情况如图 17所示。



Id	Function Name	Number of Calls	Total Function Time (cycles)	Total Function Time (%)	Total Function + Descendants Time (cycles)	Total Function + Descendants Time (%)	Min Function Time (cycles)	Avg Function Time (cycles)	Max Function Time (cycles)
1	radix2_dit	1	4,503	46.789	8,369	26.682	4,503	4,503	4,503
2	butterfly	4	2,971	30.871	2,971	9.472	700	742	835
3	main	1	1,041	10.817	9,410	30.001	1,041	1,041	1,041
4	butterfly_512	1	895	9.3	895	2.853	895	895	895
5	_main_init	1	145	1.507	9,624	30.683	145	145	145
6	__cxa_finalize	1	41	0.426	69	0.22	41	41	41
7	_fini	1	28	0.291	28	0.089	28	28	28

图 16: 1K-point FFT 函数时钟数

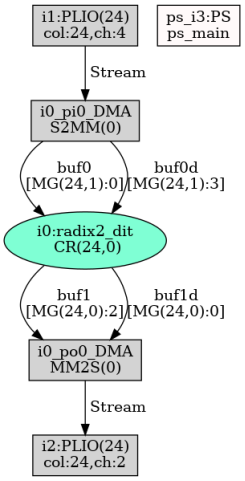


图 17: 1K-point FFT 资源使用图

4.2.2 4K-point FFT 性能分析

4K-point FFT 在 AIE Simulation 中的运行总时间为 14.275us，其中第一阶段执行 radix2 的 DIT，使用了 4 个 kernel 进行分布式计算，memory stall 分别为 493、65、65、482 次，占 0.1112-0.7736us；lock stall 为 1 次，约占 0.68us。第二阶段使用 1 个 kernel 执行数据收集，memory stall 为 133 次，占 0.1112us；lock stall 为 2 次，占 0.85448us。该 graph 的资源使用情况如图 18所示，共使用 5 个 tile，24 个 4096 字节的 buffer，8 个 DMA 以及 8 个 pilo。

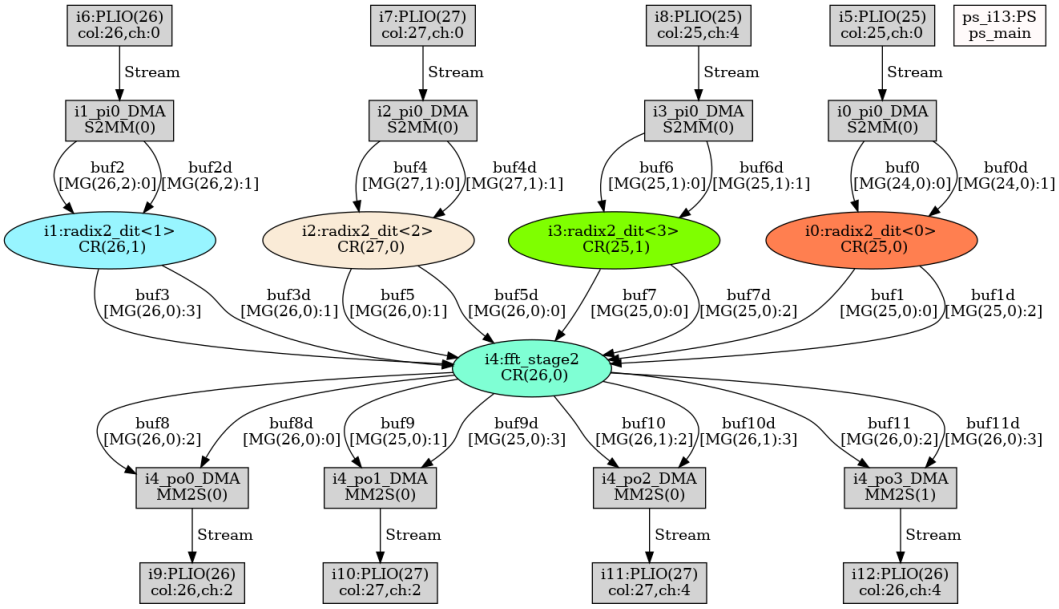


图 18: 4K-point FFT 资源使用图

4.2.3 8K-point FFT 性能分析

8K-point FFT 在 AIE Simulation 中的运行总时间为 20.661us，其中第一阶段执行 radix2 的 DIT，使用了 8 个 kernel，各分别占 1 个 Tile。其中 7 个 Tile 的 memory stall 为 65 次，1 个 Tile 的 memory stall 为 116 次，占 0.103200-0.184800us，lock stall 均为 1 次，约占 0.68us；执行第二阶段的 kernal 的 memory stall 为 8 次，占 0.017600us，lock stall 共 5 次，占 8.752800us。该 graph 的资源使用情况如图 19所示。

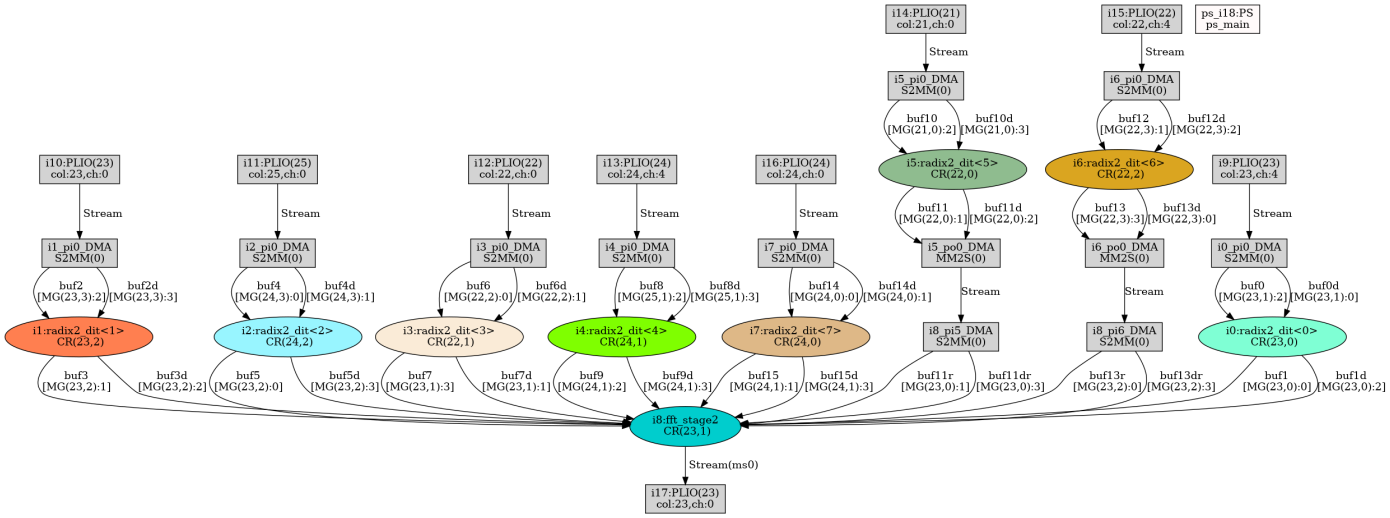


图 19: 8K-point FFT 资源使用图

4.3 输出结果分析

图 20展示了 1K-point FFT 的计算结果，可以看出与 Python Scipy 库的计算相比，二者的输出结果大致相同，但 Scipy 的计算精度更高，所以得到的输出曲线更平滑。

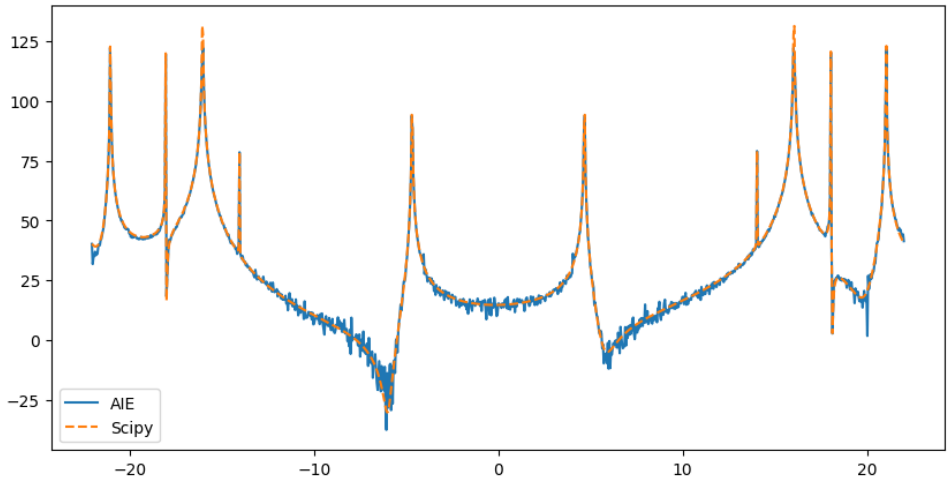


图 20: 1K-point FFT 输出结果

图 21展示了 4K-point FFT 的计算结果，当数值大于 50 时 AIE 输出值误差较小，而小于 50 的值则会出现略大一些的误差，但趋势基本相同。

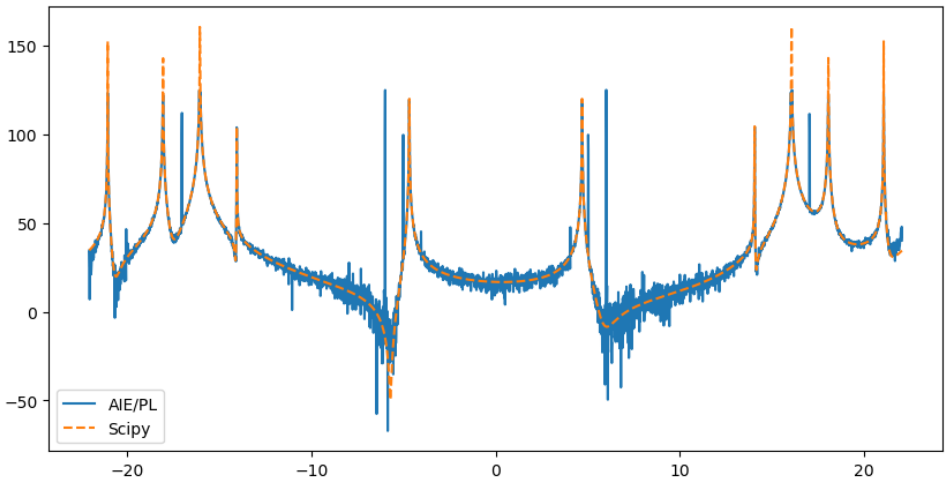


图 21: 4K-point FFT 输出结果

图 22展示了 8K-point FFT 的计算结果。与 1K 和 4K 相比，8K 明显误差较大。根据输出结果可以看出，随着数据点数量的增加，会引起一定程度的精度损失。这可能是由于我们采用了 cint16 数据结构，在计算过程中需要频繁进行移位操作，这一过程中导致了精度的损失。随着数据点数量的增加，对精度的要求也随之增加，因此精度逐渐下降。

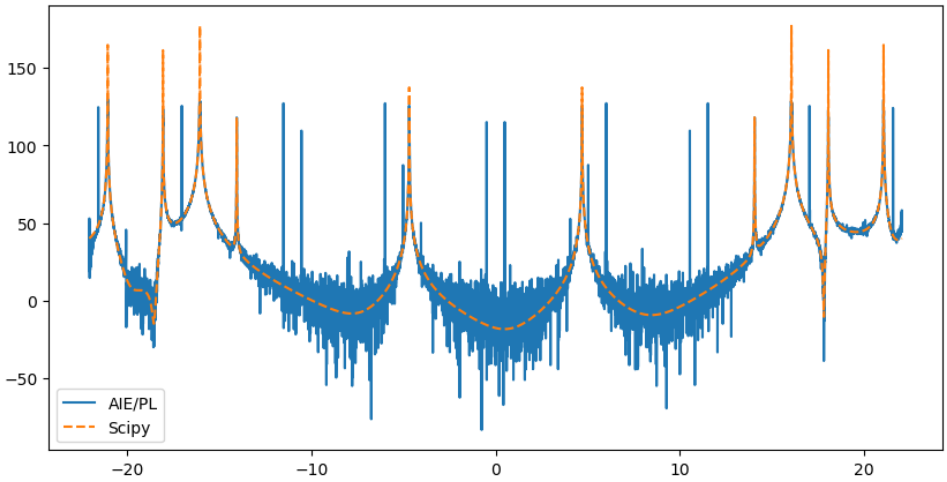


图 22: 8K-point FFT 输出结果

5 总结

本项目使用非递归的算法构造单 kernel 的 1K-point FFT，在此基础上基于分布式技术实现了精度为 16 位整型的 4K-point FFT 和 8K-point FFT，AIE 运行时间分别为 14.275us 和 20.661us。同时，本项目还实现了 PL 端和 host 端的功能，生成了能够在 VCK5000 板卡上运行的可执行文件。

本项目采用的核心优化技术包括：

- 循环优化：使用了 unroll 和 pipeline 的编译优化技术；
- 充分挖掘硬件并行：采用 sliding multiplication 的多通道乘法；
- Ping-Pong Buffer：多次迭代 butterfly 操作通过两个 buffer 实现输入输出 buffer 的转换，提高访存效率。

同时，本项目存在以下可改进方向：

- 当应用于较大点数的 FFT 时，目前 AIE 的实现存在一定的精度问题。使用精度更高的数据或设置 AIE 中的 rounding and saturation modes 可能会有所改善；
- 目前，本项目的 PL 部分仅负责数据传输的工作。可以进一步探索如何更合理地分配 PL 和 AIE 的资源，以提升性能。

参考文献

[1] Haewon Jeong, Tze Meng Low, and Pulkit Grover. Masterless coded computing: A fully-distributed coded fft algorithm. In 2018 56th Annual Allerton Conference on Communication, Control, and Computing (Allerton), pages 887–894. IEEE, 2018.

[2] Y-M Joo and Nick McKeown. Doubling memory bandwidth for network buffers. In Proceedings. IEEE INFOCOM'98, the Conference on Computer Communications. Seventeenth Annual Joint Conference of the IEEE Computer and Communications Societies. Gateway to the 21st Century (Cat. No. 98, volume 2, pages 808–815. IEEE, 1998.

[3] AI engine API user guide: Fast fourier transform (FFT). https://www.xilinx.com/htmldocs/xilinx2022_2/aiengine_api/aie_api/doc/.

[4] AI engine kernel and graph programming guide (UG1079). https://docs.xilinx.com/r/en-US/ug1079-ai-engine-kernel-coding/.

[5] AUP AI engine tutorial. https://xilinx.github.io/xup_aie_training/.

[6] Block-by-block configurable fast fourier transform implementation on ai engine. https://docs.xilinx.com/r/en-US/xapp1356-fft-ai-engine.

[7] CCFSys-ccc2023. https://github.com/ccfsys-ccc/ccc2023.

[8] Vitis™ in-depth tutorials. https://github.com/Xilinx/Vitis-Tutorials.

[9] Xilinx/xup_aie_training: Hands-on experience programming AI engines using vitis unified software platform. https://github.com/Xilinx/xup_aie_training.

[10] XRT native library c++ API. https://xilinx.github.io/XRT/master/html/xrt_native.main.html.

[11] Xtra-computing/hacc_demo. https://github.com/Xtra-Computing/hacc_demo.