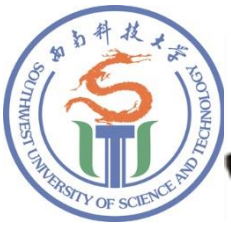

西南科技大学

Southwest University of Science and Technology

本科毕业设计（论文）

题目名称：快速傅里叶变换的并行算法研究及实现

学院名称	计算机科学与技术学院
专业名称	软件工程
学生姓名	肖劲涛
学号	5120184509
指导教师	苏波 讲师

二〇二二年六月

西南科技大学
本科毕业设计（论文）学术诚信声明

本人郑重声明：所呈交的毕业设计（论文），是本人在导师的指导下，独立进行研究工作所取得的成果。除文中已经注明引用的内容外，本论文不包含任何其他个人或集体已经发表或撰写过的作品成果。对本文的研究做出重要贡献的个人和集体，均已在文中以明确方式标明。本人完全意识到本声明的法律结果由本人承担。

作者签名：

日期：　　年　　月　　日

西南科技大学
本科毕业设计（论文）版权使用授权书

本毕业设计（论文）作者同意学校保留并向国家有关部门或机构送交论文的复印件和电子版，允许论文被查阅和借阅。本人授权西南科技大学可以将本毕业设计（论文）的全部或部分内容编入有关数据库进行检索，可以采用影印、缩印或扫描等复制手段保存和汇编本毕业设计（论文）。

保密□，在___年解密后适用本授权书。

本论文属于

不保密□。

（请在以上方框内打“√”）

作者签名：　　指导教师签名：

日期：　　年　　月　　日　　日期：　　年　　月　　日

快速傅里叶变换的并行算法研究及实现

摘要：FFT 在数字信号处理、微分方程求解、图像处理等诸多方面有着广泛的应用。随着计算机技术的不断发展、微电子技术的日益革新，FFT 技术被越来越多地应用在实际的生产生活中。同时，科学与工业界对 FFT 算法的效率的要求也越来越高。受到处理器与存储空间的限制，FFT 在工业领域面对巨大数据的输入时耗时变得十分长。近年来，CPU 的核心数量不断增加，GPU 的流处理器数量也大幅提升。因此，本文针对这种新的趋势，针对并行技术与 FFT 的结合展开了尝试。并针对几种不同的技术、算法和平台做出了相应的实验。

本文通过介绍 FFT 的基本理论、并行算法的相关知识，提出了相应的并行化方法。并且对几种技术进行了分析。我们主要使用三种技术来实现 FFT 的并行化，即，多线程技术、MPI 技术和 GPGPU 技术。其中多线程技术使用 C++的<thread>库和 Windows SDK 的 HANDLE 实现；MPI 技术在 Linux 下使用 OpenMPI 完成；GPGPU 技术则是借助 CUDA 实现。针对多线程技术，我们使用主-从模型，并且使用全局共享内存来优化；对 MPI 技术的优化同样是从“共享内存窗口”出发，对单个计算机上的多个进程之间的消息传递进行优化；CUDA 则是通过存储空间的改变来提高 GPU 与 CPU 之间的通信效率。

我们针对串行和并行算法进行了不同的实现，证明了提出了优化方案的可行性，同时发现了一些问题以及有待改进的地方。

关键词：快速傅里叶变换；并行计算；高性能计算

Research and Implementation of Parallel Algorithm for Fast Fourier Transform

Abstract: FFT has a wide range of applications in digital signal processing, differential equation solving, image processing and many other aspects. With the continuous development of computer technology and the increasing innovation of microelectronic technology, FFT technology is more and more applied in actual production and life. At the same time, science and industry are increasingly demanding the efficiency of FFT algorithms. Due to the limitation of processor and storage space, FFT becomes very time-consuming when faced with huge data input in the industrial field. In recent years, the number of CPU cores has continued to increase, and the number of GPU stream processors has also increased significantly. Therefore, this thesis attempts to combine parallel technology with FFT in view of this new trend. And corresponding experiments are made for several different technologies, algorithms and platforms.

This thesis presents the corresponding parallelization method by introducing the basic theory of FFT and the related knowledge of parallel algorithm. And several techniques are analyzed. We mainly use three techniques to realize the parallelization of FFT, namely, multi-threading technique, MPI technique and GPGPU technique. Among them, the multi-threading technology is implemented via the <thread> library of C++ and the HANDLE of the Windows SDK. The MPI technology is implemented by using OpenMPI under Linux. The GPGPU technology is implemented by CUDA. For multi-threading technology, we use the master-slave model and use global shared memory to optimize. The optimization of MPI technology also starts from the "shared memory window" to optimize the message passing between multiple processes on a single computer. CUDA improves the communication efficiency between GPU and CPU by changing the memory space.

We have carried out different implementations for serial and parallel algorithms, proved the feasibility of the proposed optimization scheme, and found some problems and areas for improvement.

Key words: FFT ; parallel computing; high performance computing

目 录

第 1 章 绪论

1.1 课题的背景

离散傅里叶变换（DFT），是傅里叶变换在时域和频域上都呈离散的形式，是科学与工程领域中一个重要的数学方法[1-3]。给定一个长度为 N 的序列，它的 DFT 也是一个长度为 N 的序列：

$$X(k)=\sum_{n=0}^{N-1}x(n)W_N^{nk},k=\left(0,1,...,N\text{-}1\right) \tag{1-1}$$

其中，$W_N^{nk}=e^{-j2\pi nk/N}$，$j=\sqrt{-1}$。傅里叶变换假设，针对任何连续的时域信号，其可以表示为多个频率不同的正弦波信号的叠加结果。傅里叶变换算法对时序采样信号，以累加的方式来分离其信号中不同正弦波的信号的基本参数（振幅、频率以及相位）。以这种方法，傅里叶变换完成了时域信号到频域的变换，因此可以利用一些工具对变换之后的频域信号做相应的处理。最后，再通过傅里叶逆变换，将频域信号重新映射为时域信号。傅里叶逆变换与傅里叶变换相同，也是一种累加处理算法。

在不同的研究领域里，傅里叶变换具有不同的描述形式。从数学的角度来看，傅里叶变换将满足一定条件的函数，表示为积分或正弦函数的线性组合。由此看来，傅里叶变换可以被看作是一种积分的特殊形式。

离散傅里叶变换（DFT），通常借助于快速傅里叶变换（FFT）来实现，即 FFT 是 DFT 的快速算法。同时 FFT 也能实现逆变换。直接计算 DFT 的时间复杂度是$O\left(N^2\right)$，而使用 FFT 计算的时间复杂度则是$O\left(N\log N\right)$。一般情况下，FFT 要求长度 N 能被因子分解，但不是所有的 FFT 都这么要求（例如 Cooley-Tukey 算法）。同时，对于任意长度为 N 的序列，也都存在复杂度为$O\left(N\log N\right)$的 FFT 算法。

Cooley-Tukey 算法[4]是最常见的 FFT 算法之一，由 Cooley 和 Tukey 两人在 1965 年提出。该方法以分治的方式递归分解一个长度为 $N=N_1N_2$ 的序列，为两个长度分别为 N_1 和 N_2 的子序列，以及 N 个信号与旋转因子的复数乘法。在 1965 年的文章中，Cooley 和 Tukey 举了一个例子，将长度为 2^n 的序列分解为两个长度为 2^{n-1} 的子序列，并递归到子序列长度为 1，该方法和其他基-2FFT 的思路大致相同，但是 Cooley-Tukey 算法可以针对任意长度为 N 的序列。除了使用 Cooley-Tukey 算法本身处理子序列，也可以使用其他的 FFT 算法来实现对子序列的处理，于是产生了混合基 FFT 算法。

离散傅里叶变换的正逆变换以相同的方式完成。逆变换的旋转因子的指数符号与正变换相反，并且多了一个$1/N$因子，除此之外，逆变换与正变换的形式相同。因此，大部分 FFT 算法都具有逆变换形式。

FFT 有多种不同的实现方式，例如从软件层面实现、硬件层面实现，或者将二者结合，不同的实现方式各有优劣。软件层面实现 FFT 算法，具有通用性强、成本低的优点，同时有着运算效率方面的限制；硬件层面实现 FFT 算法，通用性与成本都不占优势，但是相比软件层面却有着更高的效率；而软硬件结合的方式则综合了前两者的优劣，但是却有着研发成本高、实现方法复杂的缺点[5]。

针对大规模 FFT 计算，并行计算相比于串行计算有着一定程度上的优势。实现并行计算的方式主要分为两种，一种是通过中央处理器（CPU）来实现，另一种则是通过图像处理器（GPU）实现。使用 CPU 使用一半分为两种方式，一种是使用 MPI、OpenMP 等协议实现的多进程技术[6]；另一种则是使用在单一进程内使用多线程的方式完成。与多进程相比，多线程在资源共享方面有着明显的优势，即多进程之间以拷贝的方式传递值，而多线程可以使用传递地址的方式避免拷贝。与 CPU 默认使串行计算不同，GPU 默认采用并行计算模式，它提供了更多的并行处理单元和存储控制单元[7]。

1.2 课题的研究意义

快速傅里叶变换（FFT）的出现，以高效完成 DFT 变换的方式给科研、生产领域提供了一条新的实现道路。以前看来难以实现的离散傅里叶变换（DFT），现在可以通过 FFT 得到解决[8]。

傅里叶变换最主要的一个用途是，把时域信号转变为频域信号。但是传统的 DFT 复杂度过高，因此在实际运用中带来了麻烦。直接计算长度为 N 的序列，使用 DFT 的时间复杂度为$O\left(N^2\right)$。Cooley-Tukey 算法将这一复杂度降到了$O\left(N\log N\right)$。因为 DFT 变换卷积的特点，因此 Cooley-Tukey 也可以运用到卷积。

纵使 FFT 已经十分优秀，但任然有一部分问题值得我们关注。这些领域包括：FFT 数据迁移量，系数访问效率、计算精度等方面。本课题将基于不同的应用常见，给出不同的算法选择建议、修改建议。本课题主要目的如下：

1. 基于序列长度，给出 FFT 算法的选择建议。
2. 根据不同的场景，给出数据结构（复数类）的构造建议。

3. 根据不同的平台，给出并行算法的选择指导。

1.3 国内外研究现状

快速傅里叶变换最早基于高斯分治法，但是近现代可用的算法最早被 Cooley 和 Tukey 提出[4]。高斯采用和 Cooley-Tukey 相同的分治方法来计算三角级数，他的这些工作在十九世纪初完成，早于傅里叶开展同态分析的时间。1958 年，Good 提出了一种算法，该算将长度为 $N_1 \times N_2$ 的序列分解为两个长度分别为 N_1 和 N_2 的序列[9]。其中 N_1 和 N_2 互质。Good 通过下标镜像得到这一结果，但这一结果仍要在高效计算小素数长度的 DFT 算法出现之后才变得可用。下标镜像其本身可以看作是中国余数定理的应用，该定理也被运用在互质因子算法（PFA）[10]中用以处理线性同余方程。同年，Winograd 提出了一种使用卷积计算离散傅里叶变换的算法[11]。这种算法和传统的傅里叶变换计算方式不同，采用了计算函数卷积的方式实现，根据卷积定理，函数卷积的傅里叶变换等于函数傅里叶变换的卷积。之后，在 1965 年 Cooley 和 Tukey 提出了针对长度为 $N = 2^m$ 的序列的 DFT 算法，该算法将之前的 DFT 计算复杂度从 $O\left(N^2\right)$ 降低到了 $O\left(N \log N\right)$。值得注意的是，该算法适用于任意长度的序列，即和高斯分治法所针对的序列长度相同。

当一个 DFT 序列被分解为一个或多个不互质的子 DFT 序列时，分治产生了复数乘，该复数乘被 Cooley 和 Tukey 称为旋转因子（twiddle factor）。在 Cooley 和 Tukey 的论文中，他们提出了一个例子，他们首先给定了一个长度为 $N = 2^m$ 的序列，并按照时域分解将其分解为不同频率的子序列。该种操作在后来被称为按时间域分解的基-2FFT 算法。在后来的研究中，许多论文完善了原有算法的性能，其主要表现在如下方面：

1. 因为傅里叶变换的对称性，导致按频率分解（即傅里叶逆变换）的出现。
2. 基-*N* 算法的出现，并且 Bergland 指出[12]，分解基数越高，分解效率越高。

在 1974 年，有两篇文章提出了分裂基快速傅里叶变换（SRFFT）算法[13, 14]。SRFFT 实现的基本思路是，对序列的偶数与奇数部分，用不同的基进行计算。该算法结构简单，易于实现，同时针对实数与对数有着良好的表现。2007 年，一种改进的 SRFFT 算法被提出[15]。

Cooley-Tukey 算法可以以这样一种思路来理解，即将一个一维序列映射到多维序列上，但这个映射与 PFA 算法不同，这种映射是虚拟的。与 Cooley-Tukey 算法不同的是，Good 算法[9]则是真正的一维到多维镜像，并且不产生任何旋转因子。但与 Cooley-Tukey 不同的是，Good 要求目标序列的长度是互素乘的，因此 Good 算法并不适用于长度为

$N = K^m$ 的情况，其中 K 是正整数。1968 年，Rader 的方法[16]展示了如何将一个长度为 N 的 DFT 序列镜像为一个长度为 N-1 的循环卷积，其中 N 是素数。Winograd 在 1978 年提出了一种新的算法（WFTA）[17]，该算法与他在 1975 年提出的算法思路大体相同，都是使用卷积来实现 DFT。该方法首先使用 Good 算法将一个一维的 DFT 映射为多维 DFT 之后，利用卷积来得到循环，该循环以多个乘法构成。如果多维 DFT 使用行列方法执行，而不是循环，则就是之前提到的 PFA 算法[10]。与 WFTA 相比，PFA 有着更多的乘法、更少的加法。虽然 WFTA 有着更好的理论复杂度，但在实际使用中，其未能达到预期效果，因而更多的则是使用 PFA 算法。

许多方法可以用在评估 FFT 算法的效率与复杂度，对于不同的平台有着不同的评估指标与方式。2002 年的一篇文章[18]认为，以前的 FFT 存在寄存器访问次数的问题。在嵌入式系统中，重复访问寄存器而重新装载旋转因子的代价是十分高昂的。因此该文章提出了一种基-2 宽度优先算法。

麻省理工大学 Frigo 和 Johnson 开发的 FFTW 软件库[1, 19]，可以计算任意长度、一维或多维的实数或复数 DFT。FFTW 通过支持多种算法并选择它估计或测量在特定情况下更可取的算法（将转换特定分解为更小的转换）来快速转换数据。它在具有小素数因子大小的数组上效果最佳，2 的幂次方和大素数情况下效果最差，但其复杂度仍然是 Nlog2N。该软件库使用 Cooley-Tukey 算法、Rader 算法或 Bluestein 算法。

NVIDIA 公司提供了一种基于通用图形处理器通用计算（GPGPU）的 FFT 算法，该算法利用该公司的 CUDA 编程语言实现，因此称作 CUFFT。CUFFT 是 CUDA 的函数库，其相对于 CPU 在运算速度上有着明显优势，但仍然未能充分发挥 GPU 在并行计算方面的优势。Govindaraju、Lloyd 和 Dotsenko 提出了一种基于 GPU 的分层混合基 FFT 算法，该算法在 GPU 上的共享内存中利用 Stockham 公式来减少分层算法中的存储器转置开销。相比于 CUFFT 算法，该算法的性能提高了 2-4 倍。Gu、Li 和 Siegel 提出的一种 GPU 算法实现方案，该方案通过多维 Cooley-Tukey 算法实现。该算法减少计算内核的数量，同时调整了对最小全局存储器的访问次数，该算法相比于 CUFFT 算法有着精度上的优势[7]。

1.4 课题目标和研究范围

本项目内容主要为基于对 CPU 与 GPU 上 FFT 的并行算法的研究。同时介绍几种典

型的实现多线程、多进程编程方法。在此基础上分析 FFT 算法的可行性，并给出多线程优化。

本课题主要针对如下算法进行研究：

1. Naïve FFT、Naïve inverse FFT
2. Cooley-Tukey、Cooley-Tukey r
3. R2C、C2R
4. 1d、2d FFT
5. Convolve、Correlate
6. Tiling FFT
7. Radix-N FFT
8. Split-Radix FFT
9. Prime Factor Algorithm
10. Hartely Transform

1.5 文章结构安排

本文将介绍关于快速傅里叶变换在不同平台上的具体应用情况，文章共分为五个章节，具体安排如下：

第一章为绪论，本章首先提出了关于快速傅里叶变换的研究与任务，回顾了国内外研究现状，同时针对研究的重要性做出了分析。针对快速傅里叶变换，介绍了不同算法的主要特点，确定了本文的研究方向。

第二章为相关基础理论，本章节着重介绍一些基础算法与预备知识，同时还介绍了一些并行模型与其评价指标。

第三章为 FFT 算法的设计与分析，在该章节中，我们将通过第二章的模型，结合具体的技术来讨论如何实现并行的 FFT 算法，以及从理论层面验证该算法的可行性。

第四章为 FFT 算法的具体实现，本节将具体对比第三章提出的算法，并检验这些算法在实际的应用中是否如预期一样良好。

第五章为总结，回顾了本次项目成功的地方以及不足之处，同时对未来可能的研究方向做出相应的规划与展望。

第 2 章 相关基础理论

本章将介绍关于 FFT 并行化的基础理论，本章将先介绍傅里叶变换基础理论，之后介绍各种并行架构。

2.1 离散傅里叶变换

快速傅里叶变换（FFT）算法在数字信号、通讯、图像处理、数论、密码学、光学等领域有着广泛的应用。本节将从离散傅里叶变换（DFT）开始介绍，同时以 Cooley-Tukey 算法（基-2/8）为基础介绍有关 FFT 的基础理论知识。

2.1.1 离散傅里叶变换

离散傅里叶变换（DFT）是傅里叶变换在时域和频域上都呈离散的形式，其是一种可逆变换，其时间复杂度为$O\left(N^2\right)$。对于给定的长度为 N 的序列$x\left(n\right)$ ，由公式（式 2-1）给出了 DFT 的数学定义式：

$$X(k)=\sum_{n=0}^{N-1}x(n)W_N^{nk},k=\left(0,1,...,N\text{-}1\right) \tag{2-1}$$

公式（式 2-2）给出了 DFT 的逆变换的数学定义式：

$$x\left(n\right)=\frac{1}{N}\sum_{k=0}^{N-1}X\left(k\right)W_N^{-nk},N=\left(0,1,\ldots,N-1\right) \tag{2-2}$$

其中$W_N^{nk}=e^{-j2\pi nk/N}$，被称为旋转因子，e 是自然数底数，$j=\sqrt{-1}$。

2.1.2 旋转因子

旋转因子最初由 Cooley 和 Tukey 于 1965 年提出[4]。当一个 DFT 序列被分解为，一个或多个不互素的子 DFT 序列时，分治法使得在计算中产生了一个复数乘，该复数乘即旋转因子。对于任意实数 x，根据欧拉公式可以得到：

$$e^{jx}=\cos\left(x\right)+j\sin\left(x\right) \tag{2-3}$$

因此，旋转因子也可以被写作：

$$\begin{aligned} W_N^{nk} &= e^{-j2\pi nk/N} \\ &= \cos\left(2\pi nk/N\right) - j\sin\left(2\pi nk/N\right) \end{aligned} \tag{2-4}$$

根据三角函数的特性，我们可以从公式（式 2-4）中推导出旋转因子的如下特性：

（1）W_N^{nk} 的周期性：

$$W_N^{\ nk} = W_N^{\ (n+m)k} = W_N^{\ n(k+m)} \tag{2-5}$$

（2）W_N^{nk} 的可约性：

$$W_N^{nk} = W_{mN}^{mnk}, W_N^{nk} = W_{N/m}^{nk/m} \tag{2-6}$$

（3）W_N^{nk} 的共轭对称性：

$$\left(W_N^{nk}\right)^* = W_N^{-nk} \tag{2-7}$$

利用这些特性，在计算蝶形变换的时候可以对公式进行简化，从而降低计算量。除此之外，这些特性还提供了一个方向，即，将一个 DFT 分解为多个子 DFT。

2.2 快速傅里叶变换

离散傅里叶变换（DFT）通常使用快速傅里叶变换（FFT 来实现），即 FFT 是 DFT 的快速算法。由公式（式 2-1）和公式（式 2-2），可以发现，$x(n)$、W_N^{nk} 和 $X(k)$ 都是复数。因此，每计算一个 $X(k)$ 的值，都需要进行 N 次复数乘和 $N-1$ 次复数加。同时，$X(k)$ 的长度为 N，所以一共需要计算 N^2 次复数乘和 $N(N-1)$ 次复数加。因此，离散傅里叶变换的复杂度为 $O\left(N^2\right)$。复数运算实际上也是实数运算，表 2-1 列出了一次复数运算所需要的实数运算次数。

表 2-1 复数运算的实数运算次数对照表

	实数乘	实数加
一次复数乘	4 次	2 次
一次复数加	0 次	2 次

通过上表我们可以发现，序列长度为 N 的 DFT 运算一共需要计算 $4N^2$ 次实数乘和 $2N(2N-1)$ 次实数加。通过利用公式（式 2-5）至公式（式 2-7）的特性，可以将长序列

的 DFT 分解为短序列的子 DFT。DFT 的运算量与序列长度 N 成正比，长度越长计算量越大，反之越小。FFT 的出现则是为了解决这一问题，FFT 主要分为两类，一类是按时域分解（Decimation in Time）将输入序列分解为不同频率的子序列；另一类则是按频域分解（Decimation in Frequency），也被称作快速傅里叶逆变换。在本文中，主要介绍其正变换的形式，也就是 DIT。

2.2.1 计算长度为 2 的幂次方的基-2/8FFT

对于一个长度为 $N=2^L$ 的序列，其中 L 为正整数。用 $x(n)$ 表示该序列，首先将序列按照奇偶分组：

$$\left.\begin{array}{c} x(2l)=x_1(l) \\ x(2l+1)=x_2(r) \end{array}\right\}, l=0,1,\ldots,\frac{N}{2}-1 \tag{2-8}$$

据此，可以将 DFT 化简为：

$$\begin{aligned} X_k &= \sum_{N=0}^{N-1} x(n) {W_N}^{nk} \\ &= \sum_{n=0,n=even}^{N-1} x(n) {W_N}^{nk} + \sum_{n=0,n=0dd}^{N-1} x(n) {W_N}^{nk} \\ &= \sum_{l=0}^{\frac{N}{2}-1} x(2l) {W_N}^{2lk} + \sum_{r=0}^{\frac{N}{2}-1} x(2l+1) {W_N}^{(2r+1)k} \\ &= \sum_{l=0}^{\frac{N}{2}-1} x_1(l) ({W_N}^2)^{rk} + \sum_{r=0}^{\frac{N}{2}-1} x_2(r) ({W_N}^2)^{lk} \end{aligned} \tag{2-9}$$

利用旋转因子的可约性，即 ${W_N}^2 = e^{-j2\pi/\frac{N}{2}} = W_{N/2}$ ，公式（式 2-9）可以改写为：

$$\begin{aligned} {W_N}^2 &= \sum_{l=0}^{\frac{N}{2}-1} x_1(l) W_{N/2}^{lk} + {W_N}^k \sum_{l=0}^{\frac{N}{2}-1} x_2(l) W_{N/2}^{lk} \\ &= X_1(k) + {W_N}^k X_2(k) \end{aligned} \tag{2-10}$$

公式（式 2-10）中的 $X_1(k)$ 和 $X_2(k)$ 分别是 $x_1(r)$ 和 $x_1(l)$ 的 *N/2* 点 DFT 运算，即：

$$\begin{aligned} X_1(k) &= \sum_{l=0}^{\frac{N}{2}-1} x_1(l) W_{N/2}^{lk} = \sum_{l=0}^{\frac{N}{2}-1} x(2l) W_{N/2}^{lk} \\ X_2(k) &= \sum_{l=0}^{\frac{N}{2}-1} x_2(l) W_{N/2}^{lk} = \sum_{l=0}^{\frac{N}{2}-1} x(2l+1) W_{N/2}^{lk} \end{aligned} \tag{2-11}$$

由公式（式 2-10）可以发现，我们把一个 DFT 可以被分解为两个的 DFT。但值得注意的是，$x_1(l)$、$x_2(l)$、$X_1(k)$ 和 $X_2(k)$ 都是 $N/2$ 点的序列，即通过公式（式 2-10）只能计算序列前一半部分。但通过公式（式 2-5），利用旋转因子的周期性，我们可以将前一半的旋转因子扩展到后半的序列上，可以得到：

$$\begin{aligned} X_1(\frac{N}{2}+k) &= \sum_{l=0}^{\frac{N}{2}-1} x_1(l) W_{N/2}^{l(\frac{N}{2}+k)} = \sum_{l=0}^{\frac{N}{2}-1} x_1(l) W_{N/2}^{lk} = X_1(k) \\ X_2(\frac{N}{2}+k) &= \sum_{l=0}^{\frac{N}{2}-1} x_2(l) W_{N/2}^{l(\frac{N}{2}+k)} = \sum_{l=0}^{\frac{N}{2}-1} x_2(l) W_{N/2}^{lk} = X_2(k) \end{aligned} \tag{2-12}$$

同时，由公式（式 2-7）和公式（式 2-12）所揭示的旋转因子的共轭对称性，带入公式（式 2-10）可以得到：

1. 对于序列的前半部分，即 $X(k),(k=0,1,\ldots,N/2)$，有：

$$X(k) = X_1(k) + W_N^k X_2(k) \tag{2-13}$$

2. 对于序列的后半部分，即 $X(k),(k=N/2,\ldots N-1)$，有：

$$X(k+\frac{N}{2}) = X_1(k) - W_N^k X_2(k) \tag{2-14}$$

由公式（式 2-13）和公式（式 2-14）可以发现，只需要计算出区间 $[0,(N/2)-1]$ 内的所有 $X_1(k)$ 和 $X_2(k)$ 的值，就可以得到整个序列的结果。如此，相对于原本长度的序列，只需要计算其一半的量即可。例如，输入一个长度为 N 的序列，在其子序列中，我们只需要计算长度为 $N/2$ 的子序列即可，而该子序列也可以一直递归到其长度为 2。因此，FFT 的计算量得以大幅减小。两个子 DFT 之间，可以通过一定的运算组成一个更大的 DFT，该种运算通过蝶形单元实现。下面将介绍蝶形运算。

2.2.2 蝶形运算

2 点 DFT 运算被称为蝶形变换，而整个 FFT 则是由若干级的蝶形变换组合而成的。若将一个 DFT 分解为多个子 DFT，或是将多个子 DFT 合成一个 DFT。使用蝶形变换，能够节省额外的空间开销，做到“原位运算”。将公式（式 2-13）和公式（式 2-14）以图形的方式表达，如图 2-1 所示，图中没有单独标出的系数默认为 1。

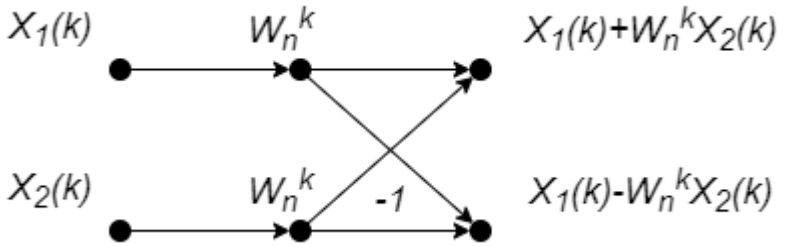

图 2-1 蝶形运算流程图

利用这种方法，以长度为 8 的 DFT 输入为例，可以将其表示为图 2-2 和图 2-3。其中，图 2-2 是一次分解的过程，图 2-3 是两次分解的过程。图 2-3 中没有下标的 $x(n)$ 和 $X(k)$ 表示对于原始输入数据其下标对于的位置。而带有下标的 $x(n)$ 和 $X(k)$ 表示对原始序列进行倒位序后的结果，倒位序分组相同的点下标一样，括号中的数代表点数。在进行第二次蝶形变换时，将第一次变换的结果再次分组后得到，每组有 $N/2=2$ 点数据，其中 $n=0,1$。用之前分解 N/2 点的方法再次分解可以得到公式（式 2-15）：

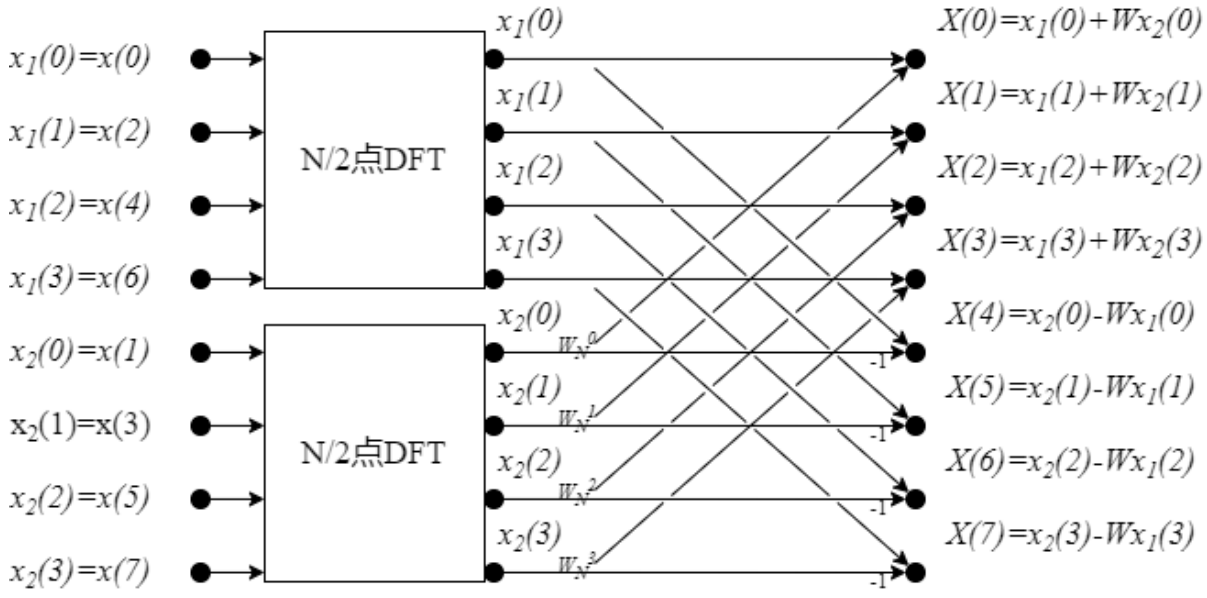

图 2-2 8 点 DFT 一次时域抽取分解流程图

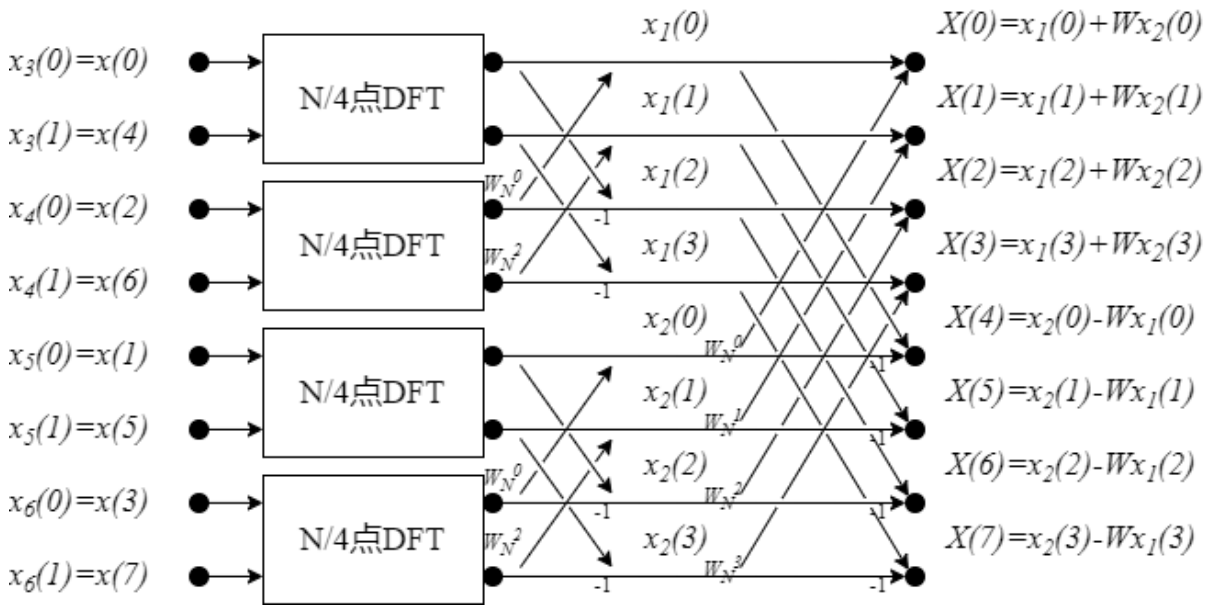

图 2-3 8 点 DFT 二次时域抽取分解流程图

$$X_4(k) \quad X_p(k)=\sum_{l=0}^{\frac{N}{4}-1}x_p(l)W_{N/4}^{lk},(p=3,4,5,6) \tag{2-15}$$

以 $X_4(k)$ 为例，现在为两点的 DFT 变换：

$$\begin{aligned} X_4(0)&=x_4(0)+W_2^0x_4(1)=x(2)+W_0^2x(6)\\ X_4(1)&=x_4(0)+W_2^1x_4(1)=x(2)+W_0^1x(6)\end{aligned} \tag{2-16}$$

由公式（式 2-16）可知，$W_2^1=e^{-j\pi}=-1=-W_2^0$，所以公式中无需进行乘法运算。我们将此种方法扩展到其他的三个分组上。这样就得到了一个完整的 8 点 DIT-DFT 流程图，该流程图包含了倒位序以及三级的蝶形变换。如图 2-4 所示：

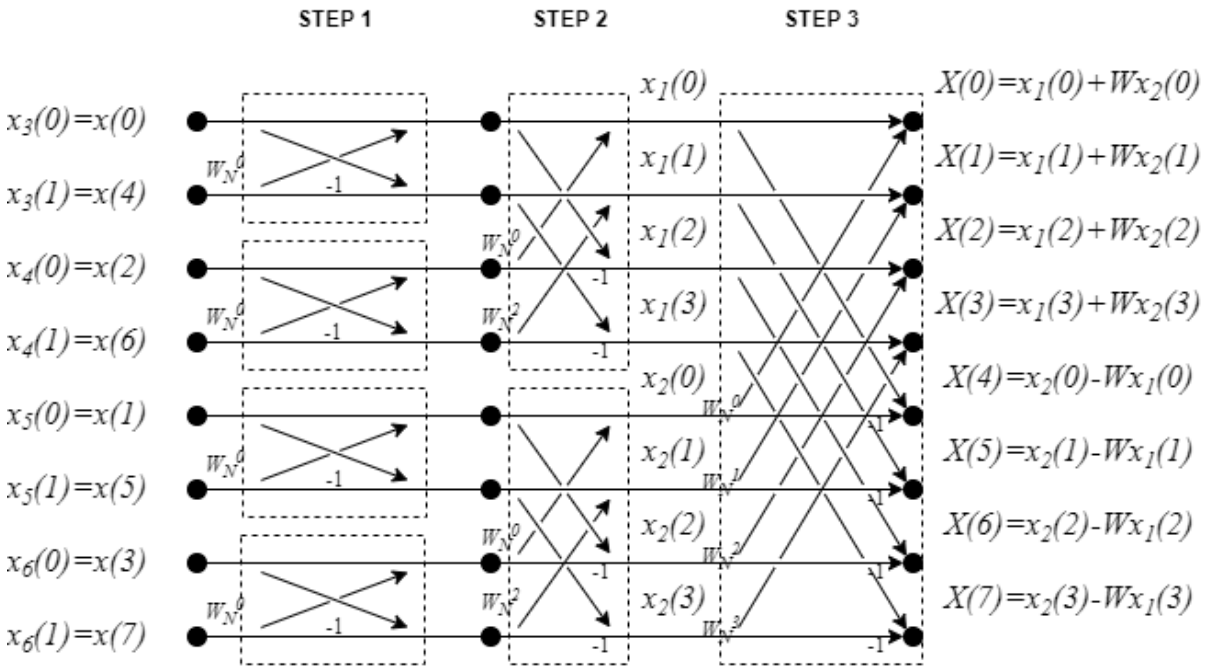

图 2-4 8 点 DFT 流程图

DIT-FFT 的一个重要的特点是原位运算，即不消耗额外的存储空间。从图 2-4 可知，蝶形单元的两个输入进行完运算之后，将同样产生两个输出，并且输出变量只和当前单元的输入变量有关，与其他单元的变量无关。换言之，对于某存储长度为 N 的序列，需要 N 个存储单元，但是在进行运算之后，输出结果任然在这个单元之中，即无需其他额外的存储单元。这种原位运算的特点，使得在实现 FFT 时所需的存储空间大幅减少，因此在空间复杂度上相比于 DFT 更具优势。

DIT-FFT 在计算之前，需要先将数据按照奇偶进行分组，之后作为一个蝶形单元的计算输入，输出按照自然顺序排列。也就是说，我们不能直接使用输入数据，而必须要对数据进行倒位序。如表 2-2 所示，展示了长度为 $N=2^3=8$ 的序列的分组过程：

表 2-2 8 点 DFT 划分表

N	0 1 2 3 4 5 6 7			
第一次划分	偶数		奇数	
	0 2 4 6		1 3 5 7	
第二次划分	偶数	奇数	偶数	奇数
	0 4	2 6	1 5	3 7

倒位序可以由变址运算来完成，这里序列的长度为 2 的幂次方，即 $N=2^m$，因此可以用

m 位二进制数表示序列 $x(n)$。如果我们尝试使用二进制的形式来表示，我们可以发现，蝶形运算所需要的另一个输入正好是当前序号 n 的倒位序，如表 2-3 所示。

表 2-3 8 点二进制倒位序

自然顺序	二进制表示	二进制倒位序	参与运算的单元
0	000	000	0
1	001	100	4
2	010	010	2
3	011	110	6
4	100	001	1
5	101	101	5
6	110	011	3
7	111	111	7

蝶形运算的两点之间的距离是已知的，以 8 点 DFT 为例，由表 2-2 和图 2-4 可以看出，第一级蝶形运算时蝶形单元两点之间的距离为 $D=2^0=1$；第二级蝶形运算时蝶形单元两点之间的距离为 $D=2^1=2$；第三级蝶形运算时蝶形单元两点之间的距离为 $D=2^2=4$。以此类推，对于长度为 $N=2^m$ 的序列来说，第 k 级蝶形运算时，每个蝶形单元输入之间的距离为 $D=2^{k-1}$。

不同级的蝶形运算之间具有相关性，即，后一级的蝶形输入是前两级的蝶形输出，它们执行的方式是串行的；而在同一级中，蝶形运算只与该单元参与运算的两个输入有关，与其他蝶形单元无关，该部分是可以并行的[7]。由图 2-4 可以看出，同一级的不同蝶形单元，它们可能使用同一个旋转因子。表 2-4 列出了针对长度为 $N=2^4=16$ 的 DFT 其旋转因子变化规律：

表 2-4 16 点 DFT 分级旋转因子变化

级数	旋转因子							
4	W_N^0	W_N^1	W_N^2	W_N^3	W_N^4	W_N^5	W_N^6	W_N^6
3	W_N^0		W_N^2		W_N^4		W_N^6	
2	W_N^0				W_N^4			
1	W_N^0							

通过表 2-4 不难发现，前一级调用的旋转因子是下一级的旋转因子的偶序列，W_N^0 是第

一级中唯一的旋转因子。

2.2.3 DIT-FFT 与 DFT 运算量对比

以长度为 $N=8=2^3$ 的序列为例，流程图中共有 3 级蝶形，每级有 4 个蝶形运算。由此可以推出，长度为 $N=2^m$ 的序列共有 L 级蝶形，每级有 2^{m-1} 个蝶形变换。对于该序列来说，每级的运算都是 N 次复数加和 $N/2$ 次复数乘。对于 m 幂次方来说，共需要 $N\log_2 N$ 次复数加和 $N\log_2 N/2$ 次复数乘。这里选择更费时的复数乘作为比较对象，DFT 需要计算 N^2 次，则其计算量之比为：

$$\frac{N^2}{\frac{N}{2}\log_2 N}=\frac{2N}{\log_2 N} \tag{2-17}$$

2.3 并行基础理论与并行架构

在 20 世纪六十年代至 20 世纪末，众多的科学家与工程师研究了种类繁多的计算机并行体系结构。同时，专家们围绕着这么一个问题争论不休，即，是要更少更快的处理器，还是更多更慢的处理器。如今，这个问题所朝向的两个方向分别演化出了拥有几个甚至数十个高性能计算核心的中央计算单元（CPU），和有着成千上万个流处理器的图形计算单元（GPU）。前者相比于后者，有着更强大的控制单元，因而在执行如奇异值分解、特征值分解等复杂计算时效果更佳，而后者拥有更多的计算单元，在计算诸如矩阵迭代等简单运算时更有优势。对于 CPU 而言，程序默认时串行的，需要使用者进行相应的并行设计，而对于 GPU 而言，所有计算都是并行的。

下面，本小节将介绍关于 CPU 的并行模型与架构，在下一小节（2.4 小节）中介绍 GPU 的并行模型与架构。

2.3.1 多处理器互连网络结构

所有的多 CPU 计算机架构都需要为不同的 CPU 之间的通信提供一种手段。这种手段可以是共享内存，即所有 CPU 共享一个存储空间；也可以是利用网络、串口等方式完成 CPU 之间点对点或一对多的通信。本节将简单介绍几种常用的多处理器互联结构。同时采用以下原则评价并行模型：

网络直径：指两个不同的处理器之间的距离，一般情况下来说，距离越近越好，这意味着两者之间的通信延时更短。

等分宽度：指为了将交叉网格分成两半而必须删除的最小边数。等分宽度越大越好，因为在需要大量数据传输的计算中，并行算法复杂度的下界是数据集大小除以等分宽度。

1、总线结构

网络并行计算模型与集群计算模型常常采用的一种结构正是总线结构，如图 2-5 所示。总线结构中的所有处理器共享一条总线，它们均通过这条总线互联。该总线是半双工的，也就是说同一时间只有一个处理器可以发送消息。当有多个处理器同时发送消息时，将由一个特殊的处理器来决定发送消息的顺序，这种操作会增加通信开销。

图 2-5 总线结构

2、 一维阵列结构

一维阵列结构因为其一维线性的特点，又称线性阵列结构。与其他所有的结构相比课程发现，一维阵列连接方式是这几个并行模型中最为简单，同时也是最容易实现的。我们假设每一个处理器是一个节点，则每一个节点与左右相邻的节点连接。两端节点因为其一维特性只与一个节点相连，两端节点其中之一需要承担 IO 责任，如图 2-6 所示。如果我们将其首尾相连，那么我们就得到了一个环形的阵列结构。

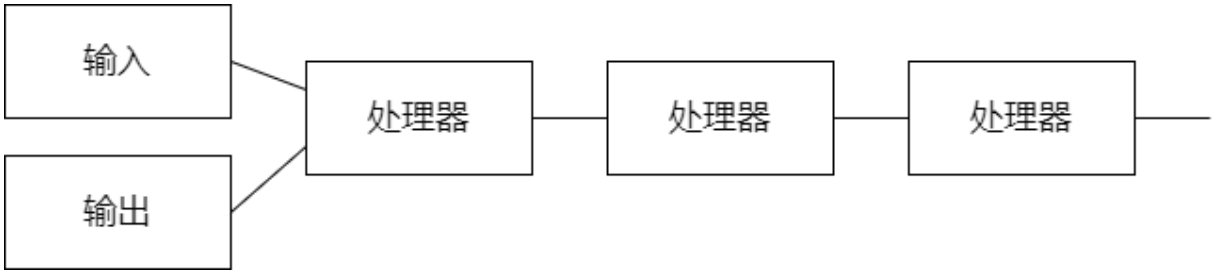

图 2-6 一维阵列结构

该模型实现并行计算的方式是，数据连续不断的发给当前的阵列。每一个节点在处理该数据的时候，同时向后发送数据，达到依次存放的目的。

节点数为 N 的一维阵列结构其网络直径为 $N-1$，等分宽度为 1。

3、二维网格结构

二维网络结构，是一种直接拓扑结构，其处理器分布在一个二维网格中，通信只在与其相邻的处理器中进行。因此，网格中的每个处理器至多能与 4 个周边的处理器通信。网格结构又分为有环和无环结构，无环结构允许边界处理器相连，其每个处理器都与 4 个处理器相邻。如图 2-7 所示。

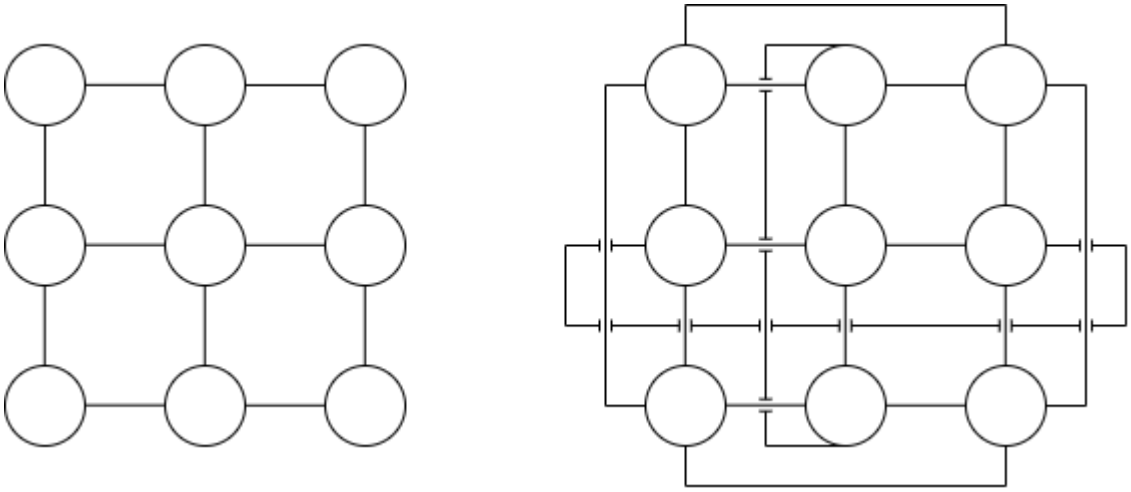

图 2-7 无环与有环的二维网络结构

节点数为 $N\times N$ 的无环网格的网络直径为 $2N-1$，等分宽度为 N。

4、超立方体结构

$N=2^q$ 个节点可以构成 q 个超立方体结构。对节点的编号采用二进制的形式完成。我们比较两个节点的二进制代码，如果代码中只有一位不同，那么这两个节点是相连的。每个节点与 q 个节点相连，即，所有的节点的度等于 q。若两个相连的节点之间的第 k 位二进制数不同，则其连接的边称为第 k 维边。如图 2-8 所示。

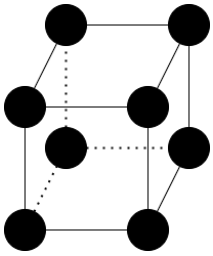

图 2-8 超立方体结构

网络直径为 q，等分宽度为 2^{q-1}。

5、蝶形网络结构

蝶形网络在之前 2.2 小节的 FFT 中已经介绍过，这里介绍其作为并行模型的特点。k 维度蝶形网络共有 $(k+1)2^k$ 个节点，$k2^k$ 条边。所有的节点分成 k+1 行，对于第 i 行来说有 2^i 个节点。假设 $P_{ri}(0 \le r, i \le n)$ 代表第 r 行的第 i 个处理器节点，则每个 P_{ri} 将于 $P_{(r-1)j}$ 直接相连，其中 j 要么等于 i，要么与 i 在二进制表示上只有第 r 位有所不同。

对于节点数位 $(k+1)2^k$ 的蝶形网络来说，其网络直径为 $2k$，等分宽度为 2^k。

6、小结

在这一节中，我们讨论了五种并行网络结构，并且给出了其网络直径与等分宽度。不难看出，并不存在这样一个模型，使得它在各个评价参数上都具有领先地位。如果意图在节点数量得以增加的同时，保留固定的边长，那么二维网格模型将是首要被考虑的模型。通过之前的介绍，我们不难发现，蝶形网络与超立方体结构之间的关系颇为密切，即它们的网络直径与其他模型相比并非对数。通过研究，我们可以发现的是，鲽形网络其等分宽度与层数之间的关系呈现几何关系。而超立方体结构因其特性，无法保持其每一个节点的边是常数。一维和二维阵列有着更好的扩展性，可以轻松地扩展到更大规模。其他的模型则由于体系过于复杂，或是在 IO 方面的影响，不易扩展。

2.3.2 并行计算模型

不同的并行计算机结构产生了不同的并行计算模型，不同的并行计算模型自然也会诞生不同的并行算法。并行模型联通了计算机软件与硬件，它隐藏了计算机硬件的具体实施细节，而将并行计算机的设计思路以数学模型的方式暴露给程序设计者，因而给并行算法的实现带来了更多的可能性。

在如今的并行计算机结构设计环境下，有着这么一些忽略硬件本身的通用性强的并行计算模型。其中最有名的是 PRAM、BSP、LogP 和 C3 模型。这些模型为算法设计者提供了设计的基础，同时给予了分析与评价的指标。同时，这些模型隐藏了算法在不同机器上的实施细节，使得算法设计者可以集中精力在软件层面、算法设计层面，而无需处理繁琐的硬件以及其接口。这些模型均是抽象模型，即无关硬件的模型，因此对于不同的硬件它们仍然具有良好的跨平台性。当算法完成的时候，可以通过这些模型的评价指标来对具体的算法进行评估与讨论。

在本小节，将讨论 PRAM、BSP、LogP 和 C3 模型，同时将对它们的优缺点进行论述。

1、PRAM 模型

并行算计存取机（PRAM）属于单指令数据流（SIMD）并行计算机的一种，这种计算机的特点是拥有一个共享存储空间。PRAM 的典型特征就是一对多。即一个共享存储器，多个逻辑处理器。如图 2-9 所示。

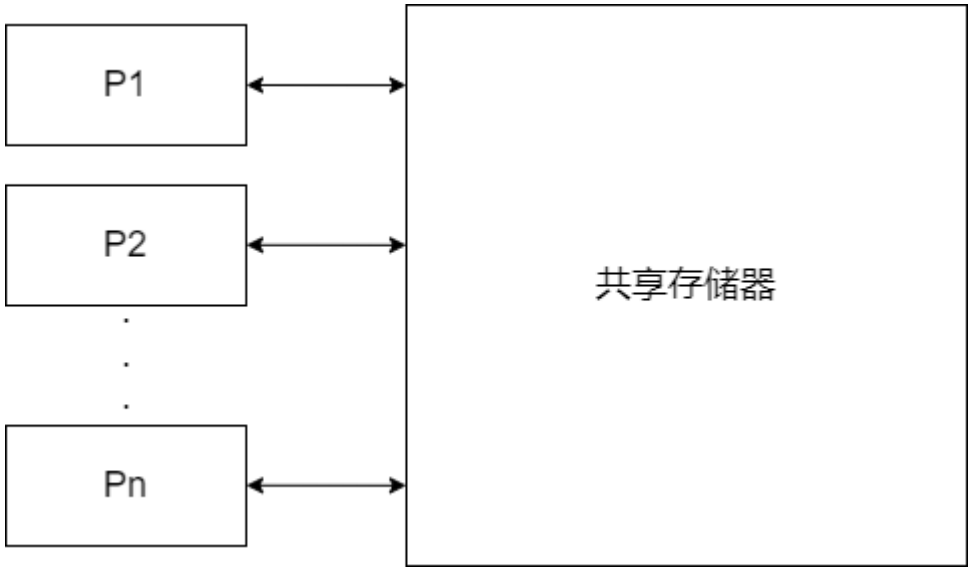

图 2-9 PRAM 模型

PRAM 是一个“理想的”模型，其假设所有的处理器总是同步的处理工作，并且不同的处理器之间的数据交换总是藉由共享存储器完成，而无需点对点的通信。使用共享存储器虽然避免了不同处理器之间的通信开销，但仍然忽略了处理器同步的问题。

2、BSP 模型

整体同步并行计算模型（BSP），是一种分布式内存的并行计算模型，把计算机抽象为三个独立的模块，它们分别是：

（1） 处理器-内存单元

（2） 全局通信网络

（3） 路障同步机制

BSP 模型的特点是采用超步（Superstep）作为并行计算的基本单元。一个 BSP 计算程序是由若干个超步组成，每一个超步又具体分为三个步骤：

（1） 各处理机进行本地计算

（2） 各处理器发出远程内存读写请求

（3） 本次超步的数据通信仅在同步后有效。

不难看出，相比于 PRAM 模型，BSP 模型强调了同步，他介于严格的串行模型和并行模型之间。BSP 模型可以通过三个参数——P、g 和 L ——来评价。其中 P 表示处理器的个数，L 表示同步开销，g 表示网络带宽。假设 P 个处理器同时传输 n 个字节的信息，则 $g \times n$ 就是通信开销。图 2-10 展示了 BSP 模型。

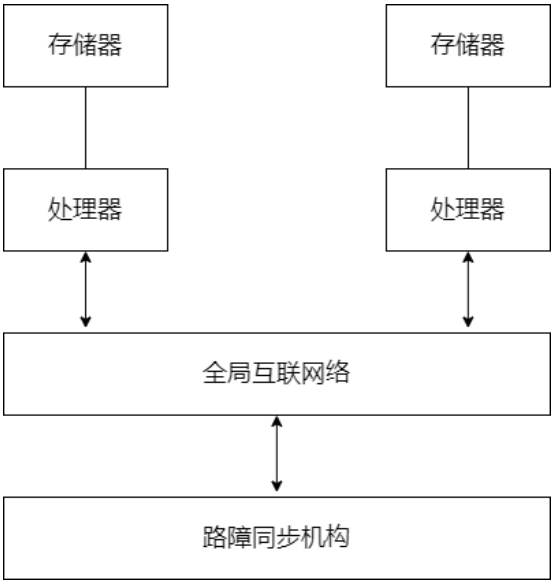

图 2-10 BSP 模型

3、LogP 模型

LogP 模型与 PRAM 模型不同，PRAM 模型采用的是共享内存，无需直接通信的方式实现。而 LogP 的存储空间是离散的，因此必须依赖不同处理器之间通信来传递数据因其特点，LogP 模型面向分布式内存、点对点通信的并行计算机架构。其通信网络由一组参数来描述，但并不涉及具体的网络结构。模型的核心如下：

（1） L（Latency）：两个处理器之间通信等待时间的最大值。

（2） o（overhead）：处理器通信的时间开销，在发送或接受消息的时候，处理器只能等待通信的完成，而不能进行其他的活动。

（3） g（gap）：一个处理器在两次通信之间的最小时间间隔。

（4） P（Processer）：处理器个数。

确定了以上四个参数，就可以将其带入 LogP 模型中求解理论的通信复杂度，同时

还能预估运行的时间开销。根据其前三个网络参数可以推出，存在一个参数 w，同时假设消息长度为 N，当 $N \ll w$ 时，两个处理器之间的通信所需时间为 $2o+L$。

4、C3 模型

C3 模型与 BSP 模型和 LogP 模型相似，主要原因是它们都依赖与同步操作。C3 模型以超级步（Superstep）为运行的基本单位。在每个超步内，可以实施局部计算以发送或接收消息，一个超步的时间由执行的计算数量和收发的通信数量确定。因为强调了通信给模型带来的时间开销，因此 C3 模型适用于粗粒度的算法。

5、小结

与其他三个模型最大不同，PRAM 模型是一个有锁的模型，即其他三个模型只需要关系不同任务的分配、分离与同步。但 PRAM 的同步机制依赖于锁实现，这种几种虽然无需用户亲自实现，但所带来的额外开销仍然是巨大的。不过 PRAM 同时也有着简单的特性，使得基于它设计的算法有着不错的可移植性。

为了屏蔽并行计算机的具体的细节，同时能对当前的并行计算机体系有着良好的适应性，BSP 模型因此而生。与 PRAM 模型最大的不同是，BSP 模型引入了超级步与路障同步机制。经过时间的检验，BSP 模型的这两个机制已经被证明了其良好的可靠性。BSP 模型还通过 P、L 、g 三个参数来对并行模型的性能进行评估。通过这些参数进行评估，可以使得算法的设计与分析更具可靠性。与 PRAM 模型相比，BSP 模型引入了三个额外的参数来评价其在通信方面的时间开销，除此之外与 PRAM 模型几乎一致。

LogP 模型的两个特点是分布式、点对点，它与 BSP 模型的区别在于评价参数更多，存储空间结构不同。它利用四个参数，即 L、o、g、P 来评价其模型的性能。它用少量的参数（尽管是四个模型中最多的）较好地描述了并行计算机系统在通信方面所面临的瓶颈。

PRAM 是上述几个模型中最简单的，因其良好的可移植性得以有着广泛的应用。LogP 则是其中最复杂的，而 BSP 则可以看作是前两者的折中。对比可知，BSP 模型综合了前两者的优点，相比于 PRAM 模型注重了通信开销，相比于 LogP 又简化了部分评价参数，从而使得更易于设计算法。

2.3.3 并行算法模型

在前一节我们介绍了并行计算模型，而在本小节，将提出一些常用的算法模型。

1、数据并行模型

在众多的并行算法模型中，数据并行模型是最简单的模型。在该模型中，一个任务被拆分成为多个子任务，并且子任务被分配给不同的处理器，子任务的数据是不同的，但是其执行的操作确实相似的。这种并行的实现方式是，将一个巨大的任务拆分成多个相似的子任务，由多个处理器同时处理子任务得到。通常情况下，数据并行阶段穿插着进程之间的通信，即在计算数据的同时，需要进行任务的收发以确保能获得新的任务。由于所有处理器面临的任务是相似的，因此通常以数据的规模来将一个大型任务进行等分，以此来保障多个处理器之间的负载均衡。

既可以通过创建一个共享存储空间的方式来实现数据并行，也可以在分布式内存上让处理器之间相互通信来实现。通常来说，共享存储空间在算法的设计上会更加简单；而消息传递则会带来额外的通信开销，对于单个多核机器来说，则是数据的拷贝，还会带来额外的空间复杂度。数据并行模型还有个额外的特点，即数据的并发度随着问题规模的增大而逐渐增大，这样就可以利用更多进程或线程来有效处理。

2、任务图模型

在任何并行算法中，其计算过程都可以看作是一个任务依赖图。任务依赖图可以是微不足道的或非微不足道的。在一些并行算法中，任务依赖图是显式映射的。在任务图模型中，利用任务之间的关系来减少通信开销，提高局部性。在某些问题中，相对于任务的数据量远远大于相对于任务的计算步数，因此任务图模型是合适的。任务被静态映射以减少任务之间的通信开销，有时分布式动态映射方法也是可能的。但即便如此，动态映射仍然使用有关任务依赖图和任务交互模式的信息来减少通信开销。在具有共享内存的可访问空间模式下，任务更容易共享，但在非连续地址或分布式内存中也可以使用一些其他方式。

这些合适的手段包括在基于任务之间的交互模式映射任务时通过增加局部性来减少交互的频率和数量。或者使用异步交互与计算重叠交互。

使用此模型的示例包括快速并行排序、使用分治算法和稀疏矩阵分解。这种由任务依赖图中的独立任务表示的并行度方式为任务并行。

3、工作池模型

工作池模型，也称为任务池模型、线程池模型，其特点是动态映射任务，保证负载均衡。在这种模型下，任何任务都可以由任何进程执行，没有任务必须映射到特定的进程。这种映射关系可以是集中式的，也可以是分布式的。指向任务的指针，它可以存储在优先级队列、列表、哈希表或树中，也可以存储在物理分布的数据结构中。任务可以在程序开始时静态调度，也可以动态生成。进程可以生成工作并将工作添加到全局工作池。如果工作是动态生成的并且使用分布式内存，那么所有进程都需要使用终止检测算法来判断工作池中的工作是否被执行完成。

假设与任务相关的数据量为D，每个任务的计算量为C，如果是$D \ll C$这种情况，则一般可以判断为工作池模型。这种情况下所带来的好处是，任务可以在不同的处理器之间灵活的转移，而无需担心巨大的通信开销。调整通信开销一个常用的手段是调整任务粒度，粗粒度（即每个任务的数据量较大）的任务往往不适合在进程之间转移。

工作池模型的示例是块调度循环并行化及其相关方法。在可以静态获取任务时，工作池模型往往采用集中映射的方式实现，块调度循环并行化正是这种方式。另一个例子是并行树搜索，其中任务是使用集中式或分布式映射方法动态生成的。

4、主-从模型

主-从模型又被称为管理者-工作者模型，在该模型中，一个或多个主进程产生的任务被分派给工作进程。如果程序的设计者可以预估问题的规模大小，或是通过随即映射的方式完成负载均衡的工作，那么任务就可以被预分配。否则的话，子进程将在不同的时间段接受到主进程发来任务。如果产生任务对于主进程来说是一件费时的工作，则可以采用第二种方式，将产生的任务不断发送给子进程，以减少等待的时间。有时候的任务必须要分阶段执行，则需要在一个阶段完成后进行同步。另外的。但级主-从模型可以被推广到多级的主-从模型，即，每个上一级的子进程作为下一级的父进程，以形如树形结构的方式进行任务的分配。这种模型对共享存储空间、分布式存储空间都有着良好的支持。

使用主-从模式时，需要确保主进程在生产任务的过程中不会拖累整个系统，当任务量太小，或者子进程执行速度过快的时候，应该调整每个被分发的任务的工作量。在选择任务粒度时，要保证执行任务的成本相对于传递任务的成本具有优势，即，子进程处理任务的时间应该超过主进程产生并发布任务的时间。异步交互有助于减少，由于主

进程产生与分发任务造成的交互与计算的重叠。

5、生产者-消费者模型

生产者-消费者模型，因为其线性的构造，又被称作流水线模型。以该模型执行并行任务时，并行数据链作为流输入到模型，任务被依次分发并执行。对于同一个数据链，同时执行不同的程序，这种操作叫做流并行。；除了发起流水线进程以外，新的数据的到达还触发了流水线中的一个进程执行新的任务。处理器节点之间的连接方式并不是只有一维线性这一种方式，还能以环、数、图的形式出现。如果我们把一个节点看作是消费者，那么它的前一个节点就是生产者，它的后一个节点就是消费者，故得此名。

在该模型中，负载均衡与任务粒度呈函数关系。任务粒度越大，填充流水线的时间越长，负载均衡实现的时间就更慢。任务粒度过小，则会增加进程之间通信的开销，因为进程在计算完一小块数据之后，必须要通过交互才能再次取得数据。因此，该模型适用于重叠交互计算。

6、混合模型

在大部分时候，但用一个模型往往难以解决问题，因而需要将多个模型混合使用。混合模型既可以是，在任务的同一阶段对不同的数据施加不同的算法模型，也可以是针对任务的不同阶段使用不同的算法模型。在某些情况下，一个算法可能拥有多个模型的特点。例如，数据可以以数据流（流水线）的方式在任务图模型中被分配。在另外的情况下，一个主进程可能可以调用多个子进程来构成主-从模型，而每个子进程又可以使用流水线模型来完成并行设计。并行快排就是一个典型的例子。

2.3.4 并行算法评估

上面我们介绍了几种常见的并行算法模型。对于一个给定的问题，假设我们已经设计好了一个并行算法，那么我们还需要相应的指标对其进行评价。下面我们将要介绍对并行算法进行评价的一些基本概念，例如运行时间、复杂度、加速比等概念。

1、运行时间

并行算法的运行时间是指算法在计算机上具体处理一个问题所花费的时间，即从算法开始执行到算法执行完成的时间。如果多个处理器不能同时启动或结束一个任务，则运行时间定义为：从最早开始执行任务的处理器开始，到最后一个处理器完成执行时结

束。

在具体实施某个算法之前，往往需要先对其进行理论评估，针对运行时间的理论评价指标被叫做时间复杂度，我们一般使用最坏时间复杂度来进行评价。对时间复杂度的分析包括它在最坏情况下的计算时间与通信时间，其中计算时间是指算法所需要执行的基本的操作数或者步数，一般采用 $O(step)$ 来表示；而通信的时间复杂度无法直接计算，往往需要通过之前提到的并行算法模型来计算，例如 BSP 模型、LogP 模型等。本文中，用 T_s 表示串行程序计算时间，用 T_p 表示并行程序的计算时间。

2、总并行开销

并行系统的总开销又称为开销函数，本文中，用 T_0 表示并行总开销。

对于一个问题，假设我们由 p 个处理器，每个处理器所花费的时间为 T_p，那么总的时间花费则是 $p \times T_p$，假如对于该问题的最快串行算法所耗时间为 T_s，那么开销函数则表示为：

$$T_0 = pT_p - T_s \tag{2-18}$$

3、并行度

并行度（DOP）指并行算法中可并执行的操作数。假设处理器的数量无限，则并行度指数和用作并行计算的处理器数量[20]。但很少有并行算法能在执行的过程中保持其并行度不改变，所以更确切的说法还应该加上时间的限制，即在某一时间范围的内并行度。

从直观上来说，并行度反映了并行算法的“并行化程度”，它反映了软件并与硬件并行性的匹配程度。与算法并行度有关的概念是粒度，一般来说，大粒度意味着，处理器独立执行大任务，因此并行度小；而小粒度意味着处理器执行小任务，并行度大。

4、加速比和效率

加速并定义为：

$$S_p = \frac{T_s}{T_p} \tag{2-19}$$

公式（式 2-19）中的 T_s 表示最佳串行算法所执行的时间，T_p 表示该并行算法所执行的时间。并行算法的效率定义如下：

$$E_p = \frac{S_p}{p} \tag{2-20}$$

公式（式 2-20）中的 p 为处理器个数。如果按照一定条件，保持每个处理器的计算规模，并行算法的加速比 S_p 与处理器个数 p 成正比，则称该并行算法在当前条件下、该计算机上具有线性加速比。在某些条件下，如果 $S_p > p$，则在该条件下，该算法具有超线性加速比。

5、成本

在并行算法中，我们假设成本为 C，处理器个数为 p，并行算法的执行时间为 T_p，那么成本可以表示为 $C = pT_p$。成本反应每个处理器解决问题所花费的时间总和。效率也可以表示为，最快串行算法求解问题所用时间与 p 个处理器的成本之比[21]。

2.4 GPU 架构

图形处理单元（GPU）最初用于图形学的图形图像渲染领域。GPU 最初的任务是面对需要大量存储空间的、计算相对简单应用场景。随着半导体技术的发展与成熟，GPU 如今也开始逐渐像 CPU 一样被应用到通用计算领域。图形处理器通用技术（GPGPU），尝试像 CPU 一样运用 GPU 来进行通用计算。但用 GPU 进行通运计算并不是一件简单的事情，因为 CPU 本身是串行结构，而对于 GPU 来说并没有串行程序运行的可能性。GPGPU 语言是专用于设计 GPU 程序的语言，其中包括 NVIDIA 推出的 CUDA 和苹果公司提出的 OpenCL。本文在 GPGPU 方面主要讨论 CUDA。NIVDIA 于 2007 年首次推出了 CUDA，CUDA 既可以被看作是一个编程语言，也可以看成是一个编程框架。一方面，它提供了自己的编程语言，另一方面也为 C 语言和 C++开放了接口。

对于高密度的、单个计算量并不复杂的运算，使用 GPU 来处理可以有效地提高运行速度。相比于多 CPU 结构的服务器，使用 GPU 整列可以在并行计算方面获得更高的性价比。但同时 GPU 的缺点也很明显，相对于 CPU 来说缺少逻辑控制核心，因此无法适应于复杂的运算。GPGPU 使用的条件有两个：

（1） 程序必须可以并行化。

（2） 主机端于设备端的延迟需要尽可能的低。

如果满足了以上的两点要求，还需要进行额外的考量。GPGPU 程序的效率由整个

运行网络的效率决定，也就是短板效应。网络包含了三个部分：1）并行代码在 GPU 上的执行效率；2）串行代码在 CPU 上的执行效率；3）CPU 于 GPU 直接的通信成本。在使用并行算法对串行算法进行提速时，我们应该确保以下这一点：并行程序之间的通信开销应该远小于串行算法，否则并行加速的时间会被消耗在通信上。此外，GPU 虽然具有众多的流处理器，但每个处理器的计算能力却不如 CPU，究其原因时多个流处理器对应一个逻辑控制器，因此在设计 GPGPU 程序时，我们还应该考虑处理器资源是否被合理利用。

2.4.1 CUDA 编程模型

CUDA 于 2007 年被 NIVDIA 公司提出。CUDA 提供了包括编程语言、IDE、CUDA 库在内的一套开发工具。CUDA 语言可以看作是对 C 语言的扩展，在保留对 C 的支持上同时进行了针对 GPU 的适配工作。

值得注意的是，CUDA 虽然本身是面向 GPU 开发的，但通过对 C 和 C++提供接口，是得其成为了同时面向 CPU 于 GPU 的异构计算网络。在这个异构网络中，CPU 被称为主机，GPU 被称为设备。在同一个网络中，可以存在一个主机和多个设备。GPU 与 CPU 协同合作，串行计算以及逻辑控制部分交给 CPU 处理，而 GPU 则负责处理高度线程化的并行计算任务。一个 CUDA 程序被分为两部分装载，一部分装载到 CPU 中，另一部分装载到 GPU 中。GPU 因为有内置的存储空间（即常说的显存），而一般的 CPU 则依赖于内存，显存和内存有着不同的管理方式。CPU 通过调用 C 的内存管理函数来控制其存储空间，而 GPU 则是通过 CUDA 库函数来管理。

在 CUDA 架构中，线程（Thread）是 GPU 执行任务的最小单元，多个线程又组合成一个线程块（Block），如果某些线程块它们所执行的功能是一样的，那么这些线程块又可以组合成一个网格（Grid）。如图 2-11 所示：

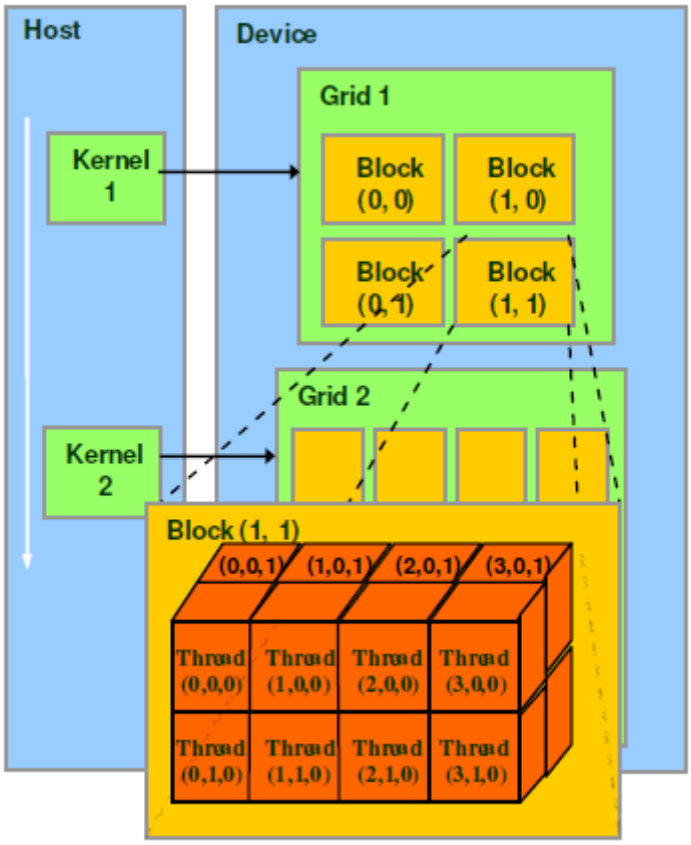

图 2-11 线程、线程块与网格

总的来说，GPU 的线程是以网格的方式来组织的。程序可以通过灵活调用线程来获取更好的性能，因此在 CPU 编程中对线程的调度就显得尤为重要。线程在计算过程中，又被分为若干个线程组，每个线程组与处理器的一个控制单元相互映射。

2.4.2 流处理器与存储模型

一个 GPU 又一个或者多个的流式多处理器（SM），又简称流处理器。一个流处理器中又有多个标量处理器核心（SP）。网格与一个或多个流处理器对应，线程块所包含的线程在一个流处理器上执行。即，程序在运行时，GPU 先对资源进行分配，首先将线程以网格的形式分配在显卡上，网格信息则会在 CUDA 库函数被调用时，从 CPU 传输到 GPU 上，然后 GPU 将网格划为线程块后分配给流处理器。在分配线程块时，GPU 首先会对每个流处理器进行检测，判断该流处理器是否满足执行新任务的要求，如果满足要求则为其分配，否则继续检查下一个流处理器，指导满足要求。线程块中的线程并发执行，并且同一个块中的线程可以同步与通信，但是不同块之间的线程只能通过全局内存进行资源共享。一个流处理器最多可以被分配给 8 个线程块，流处理器中的线程调度

单元也将被分配到线程块进行细分，将其组合为更小的线程束（warp）。线程束的大小与硬件有关，目前来说，每个线程束由 32 个线程构成。由于流处理器的 SP 在执行指令的时候需要执行 4 次流水线结构，因此一个线程束是 CUDA 程序的最小执行单元，并且一个线程束中的线程可以并行执行[7]。

CUDA 的存储器模型也是层次结构。图 2-12 展示了这种结构：

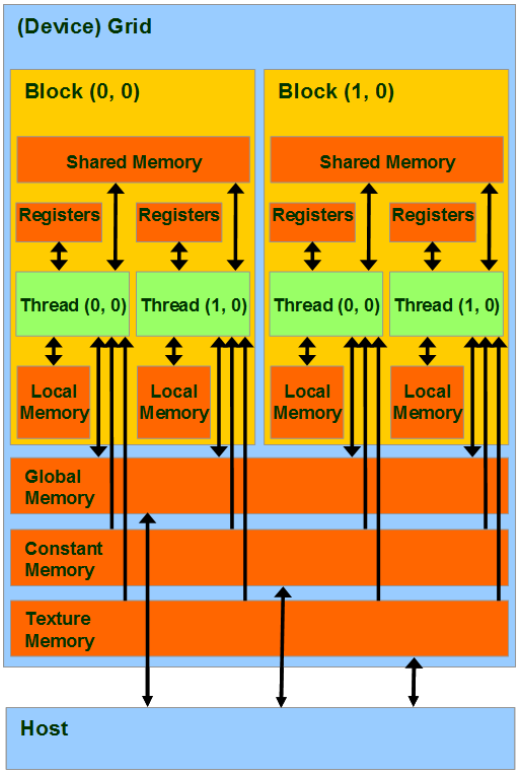

图 2-12 CUDA 存储器模型

GPU 上的各种资源均是通过这种模型得以控制。可以看出，CUDA 一共由 6 中存储模型，即寄存器、本地存储空间、共享存储空间、材质存储空间、常量存储空间和全局存储空间。这里简单介绍其中几种，同一个线程块之间的线程通过共享存储空间进行数据交互，同一个网格中的线程通过全局存储空间进行数据交互。而全局存储空间也可以被 CPU 直接访问，因此也可以用作和内存之间的交互。

第 3 章 FFT 算法的设计与分析

3.1 快速傅里叶变换串行算法

从对第 2.2 章中的 FFT 算法介绍可知，FFT 算法大致分为两个部分，其中一个是倒位序，另一个则是蝶形运算。以 Cooley-Tukey 为例的串行算法如算法 3.1 所示：

算法 3.1 FFT 串行算法

Input:

被计算的序列，*x(n)*;

数据的规模，*N*;

Output:

计算完成后的数据，*x(n)*;

```
1: j = 0;
2: for i = 0; i < N - 1; i++ do
3:     if i < j then
4:         swap(x(i), x(j));
5:     end if
6:     k = N / 2;
7:     while k <= j do
8:         j = j – k;
9:         k = k / 2;
10:    end while
11:    j = j + k;
12: end for
13: 计算 N 对应的蝶形运算级数 M;
14: for m = 1; m <= M; m++ do
15:    计算第 m 级中所含节点数 numA;
16:    计算第 m 级中每个分组中蝶形单元的数量 numB;
17:    for l = 1; l <= numB; l++ do
18:        for n = l – 1; n < N – 1; n = n + numA do
19:            根据公式（式 2-13）和公式（式 2-14）进行蝶形运算;
```

20: **end for**

21: **end for**

22: **end for**

值得一提的是，上面的算法 3-1 是 Cooley-Tukey 算法针对长度为 $N=2^M$ 的序列设计的，其他的串行算法这里不再赘述，关于串行算法的效果，我们将在下一章进行比较。

3.2 快速傅里叶变换并行设计

3.2.1 FFT 并行可行性分析

我们通过并行技术来试图对 FFT 算法进行优化，但不是所有的算法都可以并行化实现，也不是所有的可算法都适合 GPGPU 技术。下面，我们将从两个方面来分析 FFT 算法并行化的可行性与优化设计。

（1） FFT 是否能并行化：

通过在第 2 章中对 FFT 算法的介绍，我们可以确定出 FFT 算法主要分为两个部分，一是倒位序，而是蝶形运算。对于基-2FFT 来说，其蝶形单元的输入和输出均有两个，对于基-3FFT 来说，则是三个。这里我们以基-2FFT 为例，我们可以确定，在同一个蝶形单元中，其输入输出与同级的其他蝶形单元无关，只与该单元的两个输入与输出有关。也就是说，对于同级的蝶形单元的计算是可以并行的部分。而设计数据交换的不同级之间的数据则不适合并行处理，因为其计算量少，主要是逻辑控制的特点，使用并行技术所带来的通信开销可能远大于收益。因此这部分数据交换的内容我们选择使用串行技术处理。

（2） FFT 是否适合并行编程：

通过上面的介绍，我们知道，FFT 算法主要分为两个部分。我们假设串行部分的执行时间为 T_s，可并行部分的执行时间为 T_p，整个程序的总执行时间为 T_s+T_p。通过使用并行技术，我们将可并行部分的速度提高了 n 倍，即执行时间减少到原来的 $1/n$，那么，现在的总执行时间为 $T_s+(1/n)T_p$。不难看出，一个算法中并行部分所占比重越高，并行化后的效率提升越大。对于 FFT 来说，其蝶形运算是可并行的，并且是费时的运算，因此 FFT 的并行化对其效率提高很有帮助。

3.2.2 旋转因子并行设计

通过第 2 章的介绍，我们可以发现在快速傅里叶变换中，没进行一次蝶形变换，旋

转因子就会被调用一次，其属于频繁被调用的变量。但同时我们发现，如果我们知道了问题的规模，那么我们就可以确定旋转因子的值与个数。在一次 FFT 变换中，问题规模不发生变化，因此这些旋转因子也是不发生变化的。通过前一小节对 FFT 可并行化的分析，我们可以知道，在不同级之间的蝶形变换时串行的，而同一级中的蝶形变换是可以并行的。从图 2-4 中可以看出，同一级的同一线程的蝶形变换所使用的旋转因子是一样的，表 2-2 中列出了不同级所调用的旋转因子。

根据旋转因子的这些特点，我们可以预处理一部分旋转因子，在多次进行 FFT 变换的时候，直接调用旋转因子，而不必在每次调用时进行计算。

3.2.3 任务划分

通过前面的分析我们可以知道，在并行 FFT 的设计中主要由两个部分，一个是串行的、不同级之间的数据交换；一个是可并行的、同级的蝶形变换。因此，这里我们可以考虑之前提到过的主-从模型，即，在串行时由主线程来进行简单的、不同级之间的数据交换，而对于复杂的蝶形变换则交给多个子线程来处理。使用主-从模型的关键在于任务的划分与分配。首先，需要比较在 FFT 算法的每个步骤的效率和复杂度来确定是否需要对其进行并行处理，因为对于细粒度的任务来说，并行的通信开销往往大于串行的时间花费。在问题规模较小，即细粒度的任务，我们考虑使用串行处理；而对于问题规模较大，即粗粒度的任务，我们考虑并行化。FFT 作为一个大规模的、具有分治特点的算法，我们在 3.2.1 节中对其分析了哪部分适合串行处理，哪一部分适合并行处理。

需要额外注意的是，因为存在多级的蝶形变换，所以我们的程序需要不断的从主线程分发任务到子线程，同时在完成一级的变换后回收数据到主线程。因此，在我们的程序中，会不断的使用开启线程、等待线程两个步骤，即不断的分发与同步。因此这里需要额外注意同步所带来的时间开销。这里我们简单的将上面的步骤理解为“并行-串行-并行”这一过程，如图 3-1 所示，如果两次并行计算之间的串行所需要的同步开销大于使用多个线程来完成的同步开销，那么我们应该让子线程们依次完成数据的传递，而避免使用主线程完成所有的数据交互工作。此外，我们还可以考虑，在子线程计算蝶形变换的时候，主线程进行某些计算来提高下次数据交互的速度。

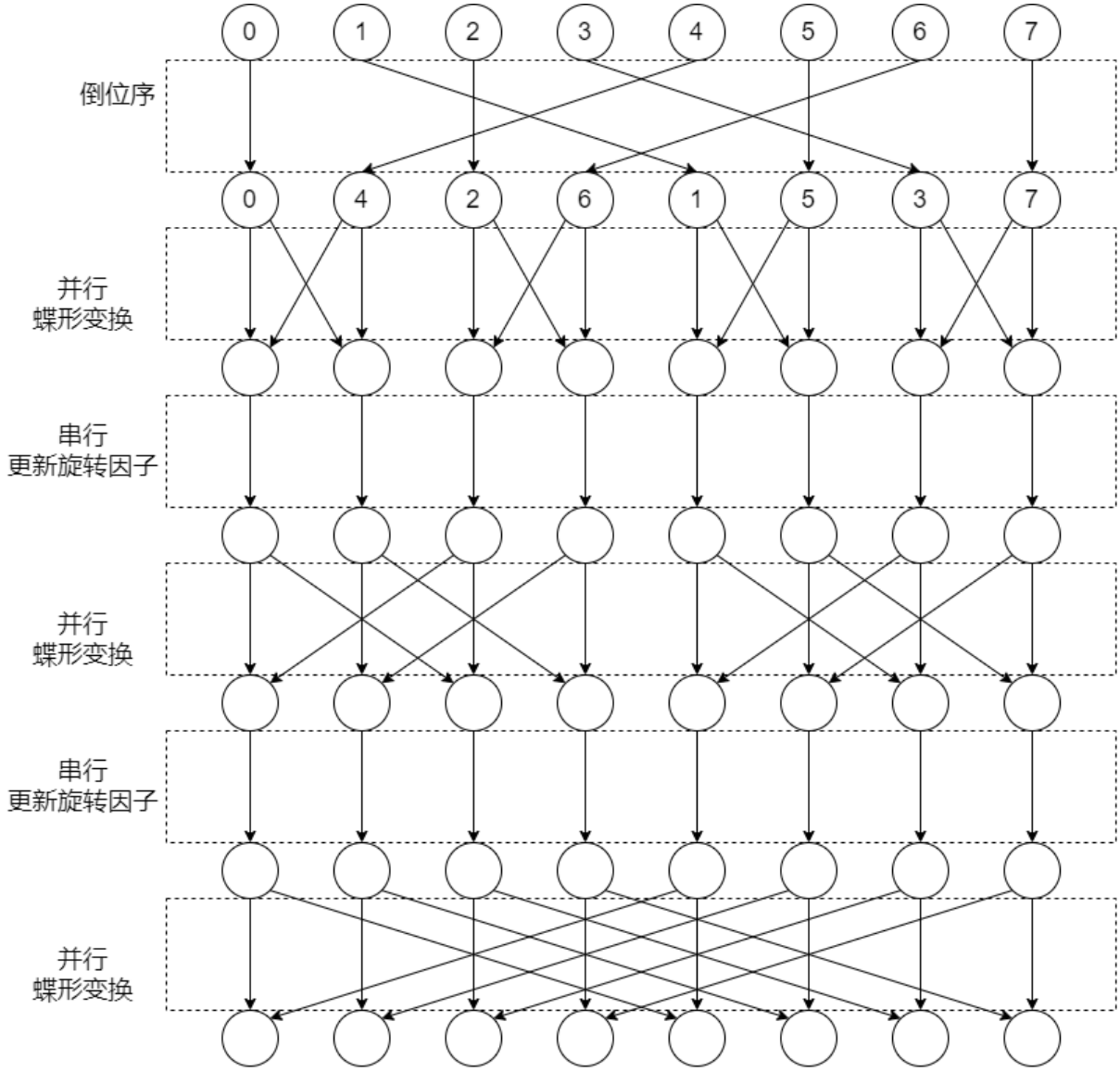

图 3-1 8 点 FFT 并行与串行流程划分

下面我们对图 3-1 做一些补充说明。首先，我们这里所讨论的并行算法均要求序列长度为 $N=k^M$，其中 $k\geq 2$。对于 Cooley-Tukey 算法来说，虽然其适用于任意长度的 FFT 计算，但是为了并行化，我们任然要求它的输入长度为 $N=2^M$。同时我们规定，任务规模远大于并行时使用的处理器数量。

对于长度为 $N=2^M$ 的序列，假设我们有 p+1 个处理器。对于第一阶段的倒位序来说，我们采用串行的方式来完成。对于该阶段来说，时间复杂度时线性的，一共需要进行 $N-1$次交换。在完成倒位序之后，我们开始第一次蝶形变换，这里使用 p 个处理器来并行处理，每个处理器负责 $N/p(2^{M-1})$组蝶形变换。完成一次蝶形变换后，主线程同步（等待）子线程，并计算在下一层所需要的旋转因子。在该阶段，我们忽略主线程计

算旋转因子的时间，因为与计算蝶形变换相比，主线程的工作量并不高。考虑子线程的蝶形变换，我们知道，串行的 FFT 算法复杂度为$O(N\log_2 N)$，这里假设我们使用了 p 个子线程来做蝶形变换，那么此时的复杂度则为$O((N/p)\log_2 N)$。以 Cooley-Tukey 算法为例的并行 FFT 算法如（算法 3-2）所示：

算法 3-2：并行 FFT 算法

Input:

需要计算的数据，*x(n);*

数据的规模 *N*;

Output:

计算完成后的数据，*x(n)*;

1: 倒位序;
2: 当前线程（主线程）创建 *p* 个线程;
3: 计算 *N* 所对应的蝶形变换层级规模 *M*;
4: 创建旋转因子存储数组 *W[N]*;
5: 计算第 1 层所用的旋转因子W_N^0; W[0] = W_N^0;
6: **for** *m* = 1; *m* <= *M*; *m*++ **do**
7: 主线程为子线程分配蝶形单元与旋转因子;
8: 主线程计算下一层的旋转因子，并按照表 2-4 的顺序放入 *W* 中;
9: 每个子线程计算当前的蝶形变换;
10: 主线程等待所有子线程完成;
11: **end for**

下面我们对并行 FFT 算法进行简单的分析，前面提到了倒位序与蝶形变换所用的时间复杂度分别为$O(N-1)$和$O((N/p)\log N)$。在我们假设，主线程计算旋转因子的速度快于子线程进行蝶形变换，这一前提条件下，同时忽略主线程等待子线程的开销。我们可以得到并行 FFT 的理论时间复杂度为：

$$O((N-1)+(N/p)\log_2 N)=O(\log_2(2^{N-1}N^{N/p})) \tag{3-1}$$

同时，我们考虑空间复杂度，虽然 FFT 是原位变换，但是在这里我们使用了额外的 *N* 的大小的数组用来存放蝶形变换所需的旋转因子。因此，并行 FFT 的空间复杂度为：

$$O(N) \tag{3-2}$$

3.3 基于 CUDA 的 FFT 设计

3.3.1 GPU 程序执行的一般步骤

结合在 3.2 节中对 FFT 并行化的介绍，与 2.4 节中对 GPGPU 技术的介绍。本节将介绍使用 CUDA 技术实现 FFT 的并行化。在具体的设计算法之前，我们需要明确的是，GPU 主要通过大量线程时间高密度、大规模的数据任务，同时 CPU 用于复杂的逻辑控制。因此在整个算法中，我们使用 GPU 完成并行的蝶形变换部分，使用 CPU 完成串行部分与逻辑控制。

CUDA 中的程序设计分为两部分，即主机端与设备端，也就是 CPU 代码与 GPU 代码。CPU 部分代码的风格类似与 C 语言或 C++；而 GPU 端则使用 CUDA 提供的 API 调用其库函数，并且将设备端的二进制码传递给 GPU。设备端代码包含了线程中网格的维度、网格内线程块的索引、线程块的维度、线程块中线程的索引以及其他指令。GPU 在隐藏延时与提高吞吐量方面，通过零代价、细粒度的线程切换操作实现。和在 3.2 节中提到的 CPU 编程技术不同，在多核 CPU 上设计程序，更多关注线程本身。而使用 GPGPU 技术则需要重点关注存储空间的设置。下面是 GPU 程序执行的一般步骤：

（1） 添加外部依赖、初始化 GPU 设备；

（2） 为 CPU 和 GPU 分配内存空间，并初始化；

（3） GPU 进行相应计算，并将结构写入显存；

（4） 将数据从现存读入内存；

（5） CPU 处理内存中的数据；

（6） 释放空间、推出程序。

3.3.2 GPU 上的并行 FFT 设计

通过之前 3.1 节的串行 FFT 设计与 3.2 节中 CPU 并行 FFT 的设计，我们已经知道，FFT 算法的蝶形变换部分计算密度高，并且同级的蝶形变换可以并行化，因此蝶形变换适用于 GPU 程序的移植，以提高效率，增大加速比。而在 3.2 节中我们介绍了，在蝶形变换之前需要进行一步倒位序的操作，该部分涉及到复杂的逻辑控制，对此我们在 3.2 节中使用一个串行的 CPU 线程来处理。在本节中，我们同样使用 CPU 来完成这部分的运算。

在将 FFT 移植为 CUDA 程序时，主要涉及的参数有如下几个：

（1） 主机端的输入数据 *dic*（Data Input CPU）；

（2） 输出数据 *doc*（Data Output CPU）；

（3） 设备端的输入数据 *dig*（Data Input GPU）；

（4） 设备端的输出数据 *dog*（Data Output GPU）；

（5） 数据规模 *N*；

（6） 最优解网格 G；

（7） 线程块 B。

下面算法 3-3 展示了基于 CUDA 设计的并行 FFT 算法，其中使用到的 CUDA_FFT()函数在算法 3-4 中被展示。

算法 3-3：基于 CUDA 的并行 FFT 算法

Input:

dic, *N*, *G*, *B*;

Output:

dog;

1: 初始 GPU;
2: 分配内存并初始化;
3: 倒位序;
4: 将主机端输入数据 dic 拷贝到设备端 *dig*;
5: **for** *Ns* = 1, *i* = 0; *Ns* < *N*; *Ns* = *Ns* * 2, *i*++ **do**
6: 调用 CUDA_FFT<<<*G*, *B*>>>(*dataIn*, *dataOut*, *N*, *i*)函数;
7: **end for**
8: 将 dog 拷贝到 doc;
9: 释放内存;

在算法 3-3 中，我们对同一级的 $N/(2^{M-1})$ 个蝶形变换使用 *p* 个线程来计算，对于一般规模的 FFT 输入来说，可以做到线程与蝶形单元的一一对应。在完成计算的同时，需要将数据拷贝到共享存储器中，以便不同线程块之中的线程能够共享数据。在计算完成后，所有数据写入全局存储器，以便能将其拷贝到内存中，最终由 CPU 处理并且输出结果。

基于 CUDA 的 FFT 的复杂度，与之前我们讨论的针对 CPU 设计的并行 FFT 的复杂度相同。在蝶形变换阶段没有引入其他的计算，对于 *M* 级 FFT 来说，需要 $(N/2)\log N$ 次复数乘，以及 $N\log N$ 次复数加，因此其时间复杂度仍然为 $O(N\log N)$。在不考虑通

信的情况下，对于长度为 $N=2^M$ 的 FFT 来说，使用 CUDA 计算的复杂度与公式（式 3-1）相同。但是存储空间则需要额外的规模为 N 显存。

算法 3-4 CUDA_FFT

Input:

dataIn, dataOut, N, i;

Output:

dataOut;

```
1: 获取当前线程，并存入变量 x;
2: if x < N then
3:     将数据从全局内存拷贝到共享内存;
4:     if x < (N / 2) then
5:         for 第 m 级所有蝶形单元 do
6:             读取预先计算好的旋转因子;
7:             蝶形变换;
8:         end for
9:         将数据从显存拷贝到内存;
10:    end if
11: end if
```

3.4 并行优化

3.4.1 CPU 并行优化

与在 3.2 中提到的 CPU 并行技术不同，CPU 本身为串行程序，我们将其扩展到了并行程序上。对于 CPU 的并行，我们一般使用分离与同步、消息传输与发送来确保程序的良好运转。对于 MPI 技术来说，其实现是调用了多个处理器来创建多个进程，我们可以在程序中预定义每个进程要处理的任务。而进程之间的通信以消息传递（按地址拷贝数据）的方式进行，因此在 MPI 技术中，多线程的同步与通信往往采用如下方式完成：

（1） 对于第 i 号进程，处理第 t 号任务；

（2） 第 i 号进程的消息传递给第 0 号进程；

（3） 进程等待，完成同步。

不难发现，对于 MPI 程序来说，多个进程总是执行相同的任务，并且需要将任务交给一个进程来完成一级蝶形变换到下一级蝶形变换之间的数据传输。这种传输方式需要耗费额外的时间在通信上。与 CUDA 的内存模型相似，MPI 也提供了“共享内存窗口”来为同一节点中的进程提供访问其他进程存储空间的方式，如图 3-2 所示。但 MPI 本身对于不同节点之间的进程，仍然只能通过通信的方式来解决。

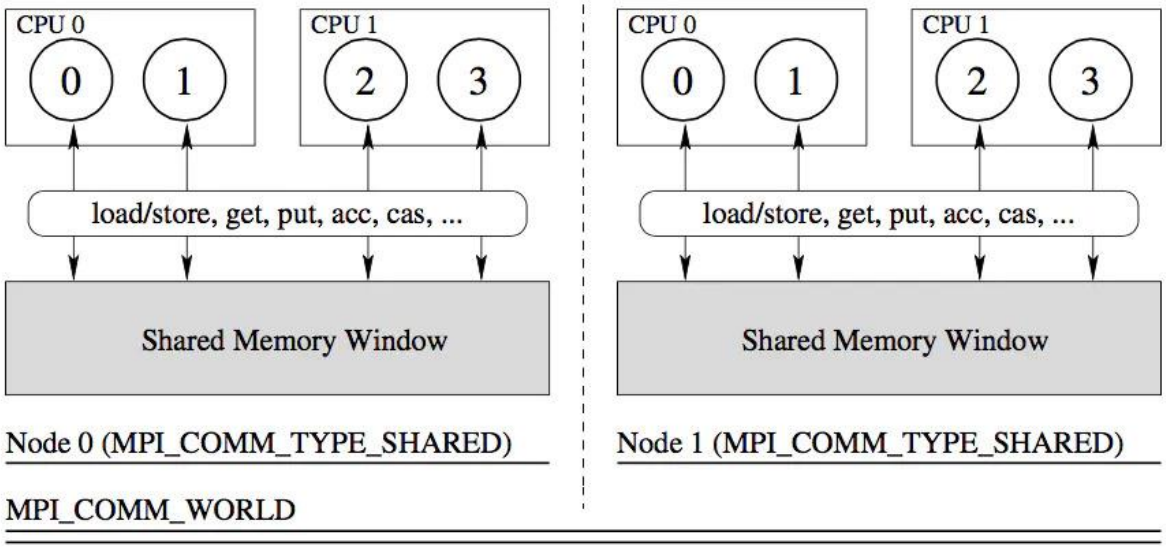

图 3-2 MPI 共享内存

除了使用 MPI（或 OpenMP）技术，C++本身也提供了线程库<thread>，这使得我们可以通过多线程的方式来完成并行 FFT 算法，而非使用多进程。与多进程相比，使用线程库最大的好处是，可以让所有线程在一块连续的地址空间中进行运算。与 MPI 或者 CUDA 相比，多线程的方法可以使得并行 FFT 能够像串行 FFT 那样做到原位运算，即不需要额外的存储空间，也不需要额外的通信开销。但与 MPI 类似的是，多线程仍然需要进行同步。这一点通过标准库提供的 std::thread::detach()和 std::thread::join()可以良好的完成。额外值得一提的是，对于幂次方长度的 FFT，我们使用 3-2 的并行 FFT 算法，无论是使用 MPI 技术或者 C++自身的标准库，都无需依赖锁与互斥量。

3.4.2 GPU 并行优化

在前一小节中，我们讨论了 CPU 并行的两种不同技术它们程序的基本单元，对 MPI 来说是进程，而对于<thread>库来说是线程。GPU 的基本单元与后者相同，以线程的方式完成并行程序。GPU 任务主要由两种执行模式，一种是基于互斥量和锁的思想，该种模式同 MPI 程序，即多个流处理器中的处理核心执行数据不同的相同操作；另一种则是

通过巨大的寄存器文件实现切换与负载。

与前一小节相同，我们依然关注蝶形变换部分不同层级之间的数据传输。对于 CPU 来说，每个 CPU 核心具有单独的指令流，而对于 GPU 来说则是一个指令被同时传输到 N 个流处理器中的标量处理核心，如图 3-3 所示。

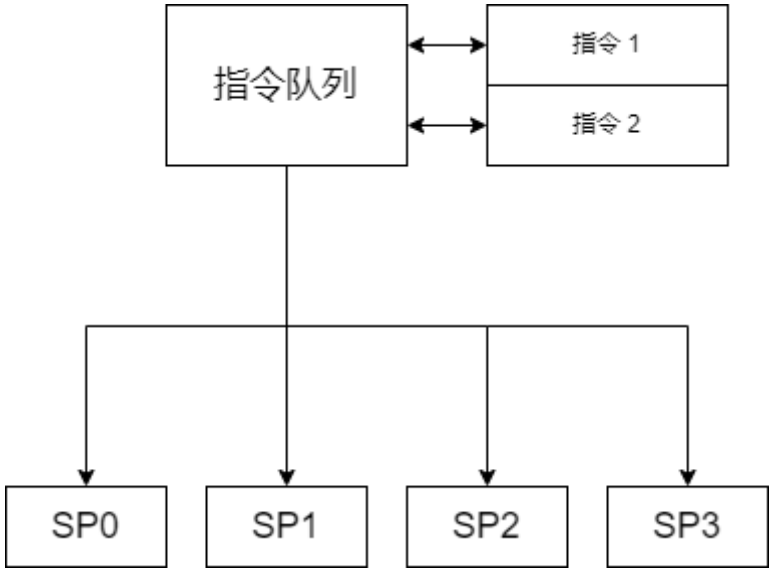

图 3-3 GPU 指令分配

在 3.2 节中，我们讨论了程序在什么什么时候并行合适，但是没有讨论要使用多少线程进行并行合适。图 3-1 可以看出，对于长度为 $N=2^3=8$ 的 FFT 来说，其第 1 层拥有 $2^{3-1}=4$ 个蝶形单元。我们之前已经讨论了，在同一级中的蝶形单元之间的计算相对独立，因此对于 N 点的 FFT 运算，我们至多可以使用 $N/2$ 个线程来处理。之前我们已经说过，一个流处理器可以被分配一个线程，对于 GTX1070 移动端显卡来说，有 2048 个流处理器，即该台电脑能处理的最大 2 的幂次方为 10，当然，也可以让一个流处理器处理多个线程。

第 4 章 FFT 算法的实现

4.1 实验平台

本次实验主要选取三个平台，分别是：

（1） ARM-Linux

（2） Linux（Ubuntu）

（3） Windows

ARM 平台因为内存和处理器核心问题，只进行串行 FFT 算法的验证。对于 Linux 和 Windows 则进行了串行与并行 FFT 的验证。

4.1.1 Arm-Linux 平台参数

本次的 Arm 开发板选用 Licheepi Nano，主要参数如表 4-1 所示。

表 4-1 Arm-Linux 平台主要参数

部件	型号或类型	参数 1	参数 2
CPU	Allwinner F1C100S	核心数量 1 核 1 线程	主频 533MHz
Memory	SPI-DDR1	32MB	

4.1.2 Linux（Ubuntu）平台参数

Linux（Ubuntu）平台为塔式工作站，具体参数如表 4-2 所示。

表 4-2 Linux（Ubuntu）平台主要参数

部件	型号或类型	参数 1	参数 2	参数 3
CPU	Xeno Gold 5218R	核心数量 20 核 40 线程	基频 2.10GHz 睿频 4.00GHz	
GPU	GeForce RTX 3090	CUDA 核心 10496	核心频率 1.7GHz	显存大小 24GB
Memory	DDR4	128GB		

4.1.3 Windows 平台参数

Windows 平台共有两个，其中一个平台拥有独立显卡，另一个平台因为不含独立显卡，所以无 CUDA 程序测试。两个平台的主要参数如表 4-3 所示。

表 4-3 Windows 平台主要参数

部件	型号或类型	参数 1	参数 2	参数 3
平台 1 CPU	Core I7-8750H	核心数量 6 核 12 线程	基频 2.2GHz 睿频 3.9GHz	
平台 1 GPU	GeForce GTX 1070	CUDA 核心 2048	核心频率 1.5GHz	显存大小 8GB
平台 1 Memory	DDR4	16GB		
平台 2 CPU	Core I5-1135G7	核心数量 4 核 8 线程	基频 2.4GHz 睿频 4.2GHz	
平台 2 Memory	DDR4	16GB		

4.2 数据结构和辅助函数

在具体介绍 FFT 的实现之前，我们需要介绍一些数据结构的定义与重要的算法。这些数据结构主要包括，复数类、旋转因子的构造和 C++的一些函数对象的处理；算法包括倒位序、线性同余方程的求解等。本文的项目使用 C++和 CUDA 编写，对于 PFA 算法使用 Python3 编写，但不进行效率评估。

4.2.1 数据结构

1、复数类的定义

C++标准库提供了复数类，但是我们仍然需要自行构造一个复数类，原因由两点：

（1） 我们期望将复数类模板化，以适配单精度浮点数和双精度浮点数；

（2） 我们需要重载部分运算符，以保证复数与旋转因子之相乘可以正确完成。

在复数类的内部，我们存放两个保护成员，即复数的实部和虚部，对于该成员变量，我们使用返回引用的方式为其添加“get()”方法。复数类的大概定义如下所示。

```
template<typename T>
class complex_t {
```

```
private:
    // 虚部，image
    T im_data;
    // 实部，real
    T re_data;
public:
    // 其他函数
}
```

在其他函数中，主要定义了对复数类的运算符重载，以及一些其他的函数，例如计算共轭、正切、平方和指数等。

2、旋转因子的定义

在具体的工程文件中，根据算法的不同，旋转因子可能会有所不同。这里以一维的 DFT 和 Cooley-Tukey 算法所用的旋转因子为例。对于旋转因子W_N^k其计算方式如算法 4-1 所示。

算法 4-1 旋转因子计算函数（omega_func）

Input:

输入序列的长度 N，旋转因子的上标 k;

Output:

旋转因子W_N^k;

1: $r = 1$;

2: *theta* = -1 * 2 * pi * k / N;

3: **return** W_N^k = (r * cos(*theta*), r * sin(*theta*));

在具体的 FFT 实现中，我们采取先构造旋转因子，然后将旋转因子放入预定义的存储空间。在进行蝶形变换的时候直接调用数组中预先计算好的旋转因子。

4.2.2 辅助函数

1、倒位序

倒位序在算法 3-1 的第一部分已经出现，这里单独列出以便更直观的了解。

算法 4-2 针对 2 的幂次方的倒位序

Input:

原始序列 *x*, 序列长度 *n*;

Output:

倒位序之后的序列 *x*;

```
 1: n1 = N – 1;
 2: n2 = N / 2;
 3: i = 0;
 4: j = 0;
 5: k = 0;
 6: for i = 0; i <= n1; i++ do
 7:         if i < j then
 8:                 swap(x(i), x(j));
 9:         end if
10:         k = n2;
11:         while k <= j do
12:                 j = j – k;
13:                 k = k / 2;
14:         end while
15: end for
16: return x;
```

2、幂次方判断

对于 radix-n 的 FFT 来说，必须要确保其输入的数据长度为 *n* 的幂次方。下面介绍本文使用的 radix-2/3/5 的算法如何判断数据长度是否为 2/3/5 的幂次方。对于 2 的幂次方来说，在二进制表示上总是表示为 1 开头后跟上若干个 0 的情况。如果一个二进制数

表示为 1000，那么该二进制数减去 1 则为 0111，我们可以发现它们两者相与为 0。利用这一特点，我们可以给出如下的实现代码：

```
bool is_pow_of_2(size_t value)
{
    return (value & (value - 1)) == 0;
}
```

但对于 3 或者 5 这样的特殊情况，则不能使用这种方法实现。这里的实现办法是，在一定范围内，找到最大的 *n* 的幂次方 *E*，如果一个数 *p* 能够被 *E* 整除，那么 *p* 是 *n* 的幂次方。以 3 为例的算法实现如下：

```
/*
    2^64 = 18,446,744,073,709,551,616
    3^40 = 12,157,665,459,056,928,801
*/
bool is_pow_of_3(size_t value)
{
    return value > 0 && (12157665459056928801u) % value == 0;
}
```

即，找到 64 位（C++中最大的、预定义的数据类型大小）中最大的 3 的幂次方，再与输入的值进行比较即可。

3、线性同余方程求解

线性同余方程是计算互质因子算法（PFA）重要的一环，但因为其并非与之前介绍的 DIT-FFT，因此本文不会将其纳入到性能对比中，但会在工程文件中实现。下面简单介绍一下线性同余方程。在数论中，形如式（公式 4-1）的同余方程被称作线性同余方程。

$$ax \equiv b(\mathrm{mod}\, n) \tag{4-1}$$

此方程，当且仅当 *b* 能被 *a* 与 *n* 的最大公约数整除时有解。如果此时有解，那么所有的解可以表示为：

$$\{x_0 + k\frac{n}{d} \mid k \in \mathbb{Z}\} \tag{4-2}$$

其中 d 时 a 与 n 的最大公约数，在模 n 的完全剩余系中 $\{0,1,\ldots n-1\}$，恰有 n 个解。线性同余方程的求解算法如算法 4-3 所示。

算法 4-3 线性同余方程求解

Input:

a, b, n;

Output:

X;

```
1: if b == 0 then
2:         return 无解;
3: end if
4: d = gcd(a, n);
5: if b % d != 0 then
6:         return  无解;
7: end if
8: x = 0;
9: for ; x < n; x++ do
10:        if (a * x - b) % n == 0 then
11:                break;
12:        end if
13: end for
14: for i = 0; i < d; i++ do
15:        X[i] = (x + (m / d) * i);
16: end for
17: return X;
```

4.3 串行 FFT 的实现

本小节将对比不同串行 FFT 的运行时间，并在后面的小节中进行分析。这里选取进行对比的算法有：

（1） DFT

（2） Cooley-Tukey

（3） R2C

（4） Radix-2/3/5

（5） SRFFT-Naïve

（6） SRFFT-Radix2

4.3.1 Linux 平台下的运行时间

在本次对比中，同时比对了 Arm-Linux 和 Ubuntu 下的运行时间差异，对比图如图 4-1 所示，其中运行时间取对数处理，Radix-3/5 的规模为小于 2 的幂次方的最大值。

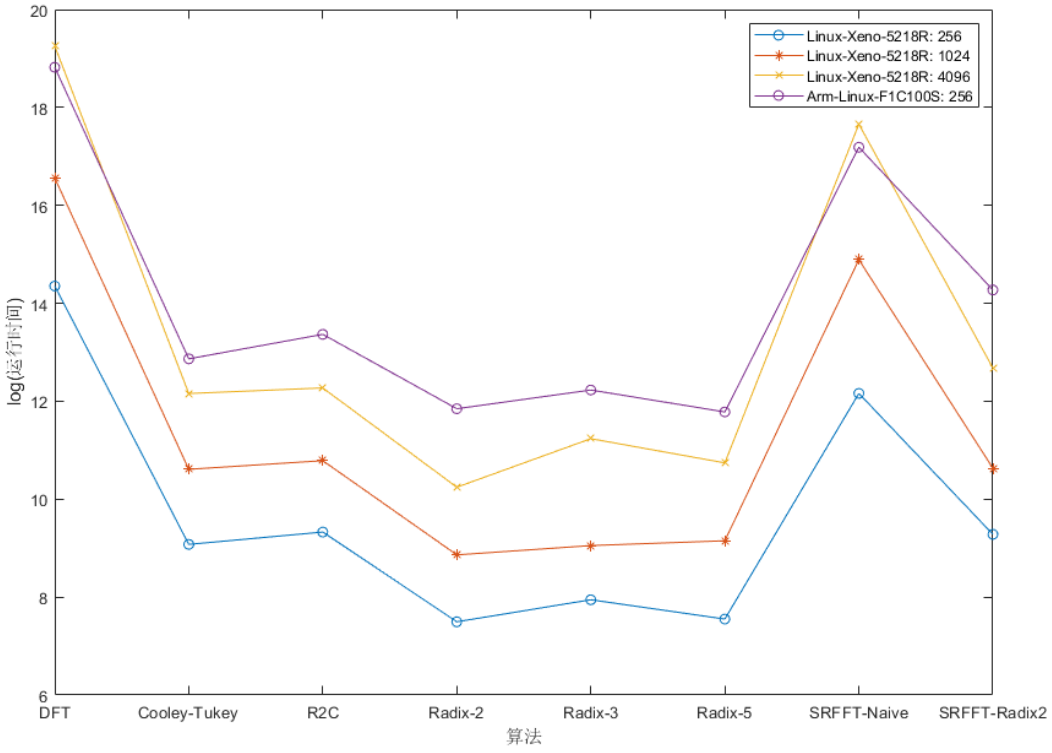

图 4-1 Linux 平台下运行时间对比

4.3.2 Windows 平台下的运行时间

串行对比选用平台 1 完成，运行时间如图 4-2 所示，同样对运行时间取对数处理。

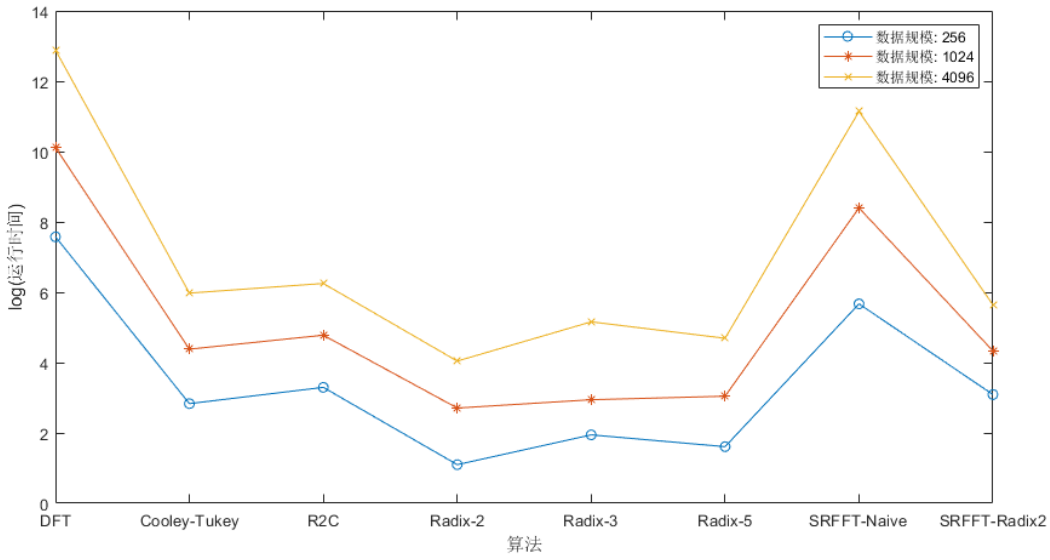

图 4-2 Windows 平台下运行时间对比

这里需要注意的是，Windows 平台与 Linux 的计时器底层逻辑不同，因此无法对两个平台进行直接的比较。

4.4 并行 FFT 的实现

并行 FFT 这里使用三种技术完成，C++的 thread 库、MPI 和 CUDA 完成。

4.4.1 以 thread 库的方式实现

在 Windows 平台的测试数据如表 4-4 所示，折线图如图 4-3 所示。

表 4-4 Windows 平台下使用 thread 库运行时间

线程数	1024	2048	4096	8192	16384	32678	65536	131072
1	4	7	15	35	110	498	614	16493
8	5	7	15	37	122	515	613	16962
10	6	9	14	36	110	510	590	16316
12	7	8	14	37	109	504	608	16254

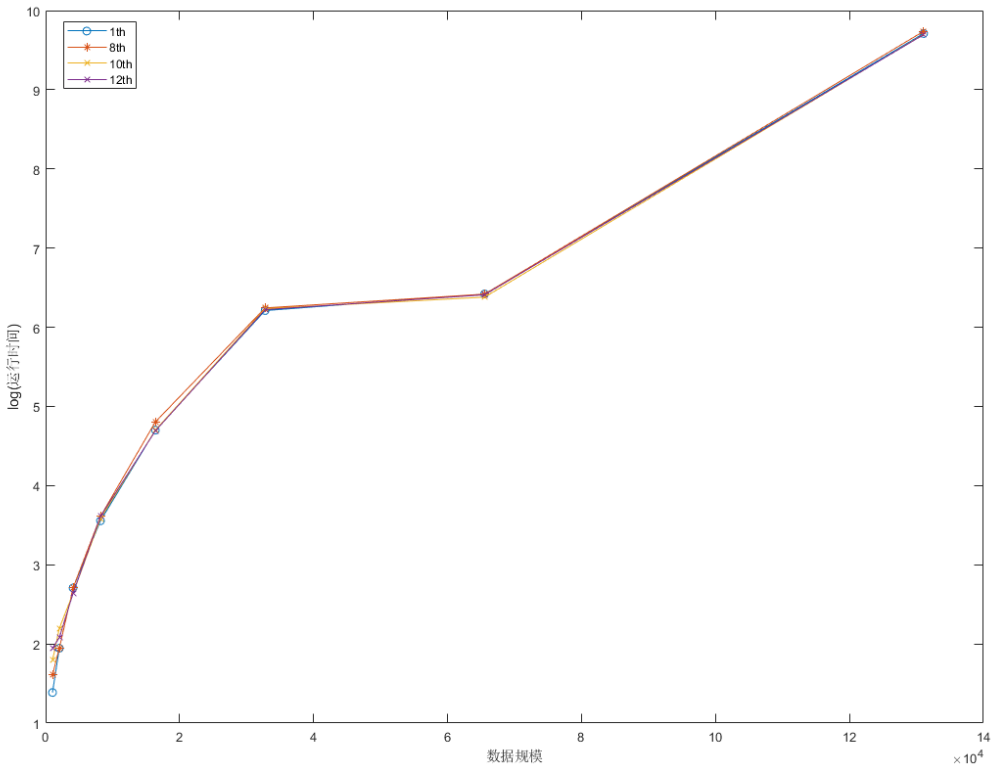

图 4-3 Windows 平台下 thread 库运行时间图

这里我们惊讶的发现，无论是 1 个线程还是 12 个线程，它们的运行时间几乎是一样的。我们为进程分配最高的优先级，然后查看任务管理器，如图 4-4 所示。

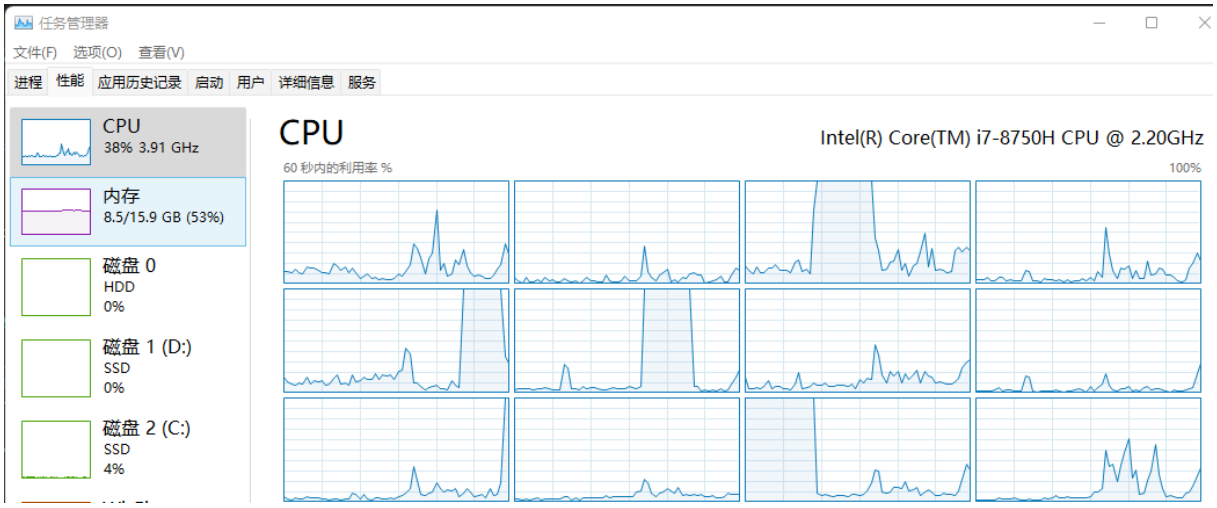

图 4-4 查看任务管理器

我们可以发现，虽然我们启用了多个线程，但单个的线程仍然只对应了 CPU 的一个逻辑处理单元。即，无论是 1 个还是 12 个线程，均在一个逻辑 CPU 上运行。这里我考虑使用 Windows SDK 提供的线程创建工具<process.h>来创建 HANDLE，试图将线程

直接绑定到处理器上。但仍然发现，在同一时间，只有一个线程被处理，无法做到多核并行计算，只能并发的计算。

4.4.2 以 MPI 的方式实现

这里在 Linux 环境下使用 OpenMPI 实现，运行时间如图 4-5 所示。

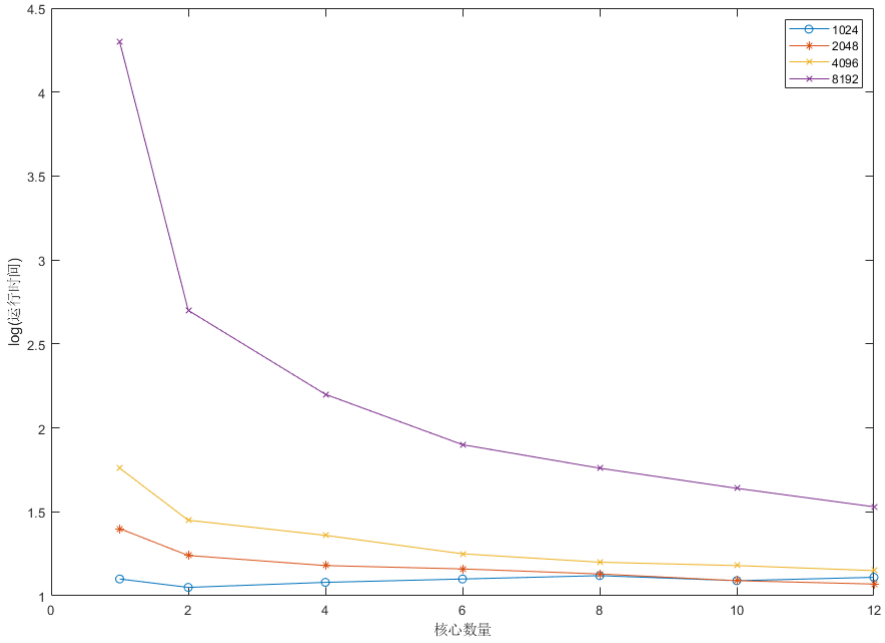

图 4-5 Linux 下 OpenMPI 运行时间图

我们可以利用公式（式 2-19）算出加速比，如图 4-6 所示。

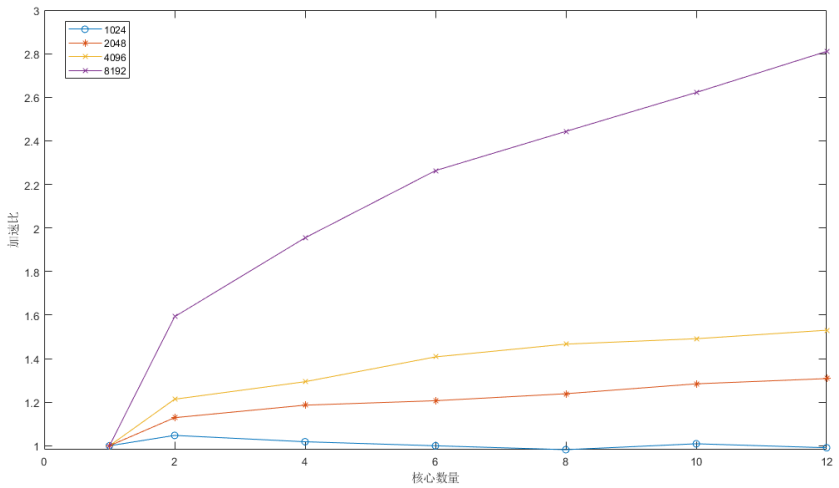

图 4-6 加速比

同时我们还可以根据公式（式 2-20）计算出效率，如图 4-7 所示。

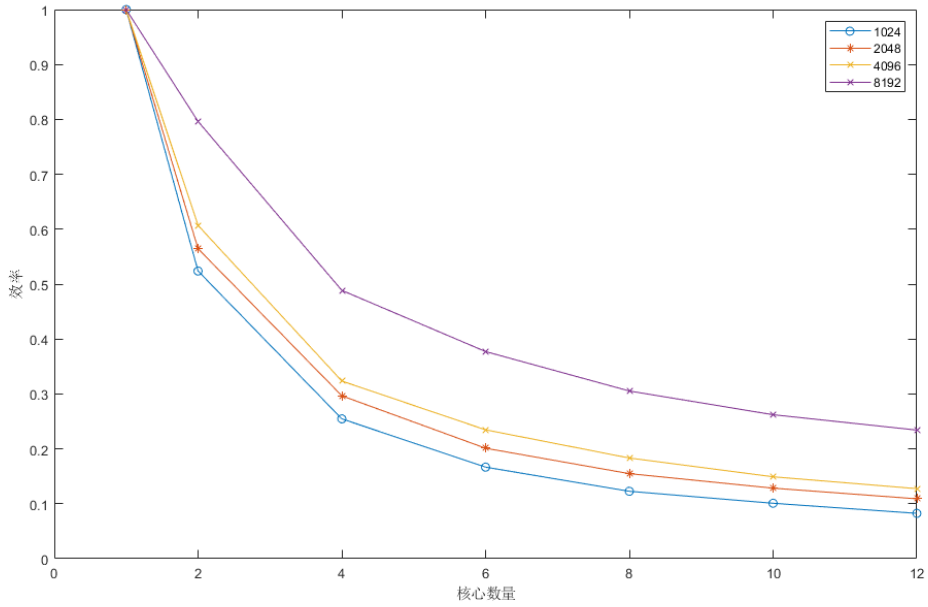

图 4-7 效率图

4.4.3 以 CUDA 的方式实现

本节中将主要对比两个算法，一个是本项目实现的基于 CUDA 的 Cooley-Tukey 算法，另一个则是 CUDA 库提供的 CUFFT 函数，运行结构如图 4-8 所示。

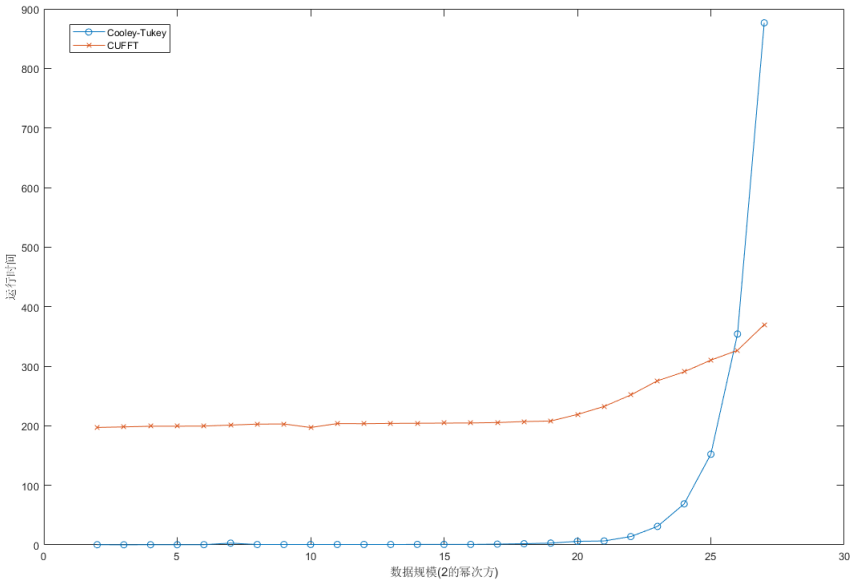

图 4-8 基于 CUDA 的 FFT 运行结果

结论

FFT 在数字信号处理、微分方程求解、图像处理等诸多方面有着广泛的应用。随着计算机技术的不断发展、微电子技术的日益革新，FFT 技术被越来越多地应用在实际的生产生活中。同时，科学与工业界对 FFT 算法的效率的要求也越来越高。受到处理器与存储空间的限制，FFT 在工业领域面对巨大数据的输入时耗时变得十分长。近年来，CPU 的核心数量不断增加，GPU 的流处理器数量也大幅提升。因此，本文针对这种新的趋势，针对并行技术与 FFT 的结合展开了尝试。并针对几种不同的技术、算法和平台做出了相应的实验。本文的主要工作如下：

（1） 提出与分析并行 FFT

本文在第二章介绍了基础 FFT 理论、并行理论，并且在第三章讨论了 FFT 并行化的可能性，以及如何并行的问题。分析了哪些步骤适合并行，哪些步骤适合串行。同时计算了并行的理论复杂度。在第三章，我们通过时间复杂度、空间复杂度，以及额外的评价指标，如加速比、效率，来确定了，将同一级的蝶形变换部分作为并行部分、将不同级的蝶形变换直接的数据传输作为串行部分，这方案的可行性。

（2） 并行 FFT 的相关优化

随着研究的深入，我们发现，并不是所有的并行设计都能够满足提高原本串行算法的效率。对于 MPI 技术来说，通信与同步是其运行时间的主要额外开销；而对于 CUDA 技术来说，主要的额外开销则来自消息在 CPU 与 GPU 之间的通信。于是我们提出了相应的优化方案。1）对于使用<thread>线程库的程序，之间开辟一个全局共享的内存空间即可，多线程技术使用主-从模型，主线程为子线程安排任务，同时在子线程被分配任务时进行下一轮的旋转因子计算。2）对于使用 MPI 技术的程序来说，我们采用预先计算旋转因子，并让每一个进程重复不同数据的相同计算。因为是在一台电脑上完成的计算，因此可以通过共享内存窗口来实现本地所有进程之间的数据共享，而不需要再依赖进程之间的点对点传递。3）对于 CUDA 技术来说，我们探讨了不同的显存配置与内主机之间交互的可能性，提出了相应的优化方案。

（3） FFT 具体的部署与实现

我们在不同的平台上完成了不同技术、算法的相应的 FFT 实现。基本验证了我们之前关于 FFT 并行化做出的一些假设，同时也暴露了一些不足与问题，这些问题有：

（1） 多线程技术无法做到真正的并行。

（2） MPI 技术不同消息传递方式没有被考虑。

（3） 未能利用 CUDA 的存储空间进行更进一步的优化。

虽然对 FFT 的研究已经成果颇丰，但随着新技术、新架构的不断出现。如何提高 FFT 在不同平台，针对不同任务目标的处理能力依然十分值得研究。对于 Arm 平台来说，CPU 的速度基本上是几百兆赫兹，并且核心数量也远不如桌面处理器。本文所提出的方法，均是在软件层面上实施，并没有做到软硬件相结合。因此，在面对低功耗、低速率的 Arm 平台，如何有效地实现 FFT 仍然有待研究。

同时，本文通过第三章的理论推导、第四章的实验，发现了在数据规模达到一定规模后，大部分 FFT 的运行时间仍然是巨大的，尽管它们有着对数级别的复杂度。因此，如何高效处理大规模的 FFT 运算同样是一个重要的研究方向。

致谢

首先感谢苏波老师对我的关系、培养以及照顾。从最开始的并行计算课程，到最终的毕业设计都多亏了苏波老师的帮助才能顺利的完成。感谢龙吟老师在机器学习方面的培养于帮助，给予了参加项目历练的机会，并且还在平日工作中帮我解答疑惑。感谢三名辅导员在平时生活上的帮助。

此外，还要感谢一些任课教师对我的帮助，他们是，计算机网络教师王昆，数据库原理教师张世铃、设计模式教师周巧临。

感谢德阳红外科技创新中心有限公司的同事们，袁永刚老师对我在嵌入式、电子线路和整体项目设计方面的指导于帮助，肖敬前辈对我在图形图像方面给予的专业帮助，以及陈静老师、王珺前辈、张娟前辈对我平时工作生活中的帮助。

感谢原专业（信息对抗技术）的老师同学的帮助，感谢计算机学院各位老师、同学的帮助于关心。同时感谢原西一 306 室友，吴彪、欧阳龙、郭艺超；感谢东一 B622 的室友，裴珂、宋海源、张帮辉、韩建明。

同时，还要感谢我的家人于朋友们。

此外，感谢两家校外饭店，四川名小吃面馆和向味之恋，好吃不贵。

参考文献

[1] FRIGO M, JOHNSON S G. The design and implementation of FFTW3 [J]. Proceedings of the IEEE, 2005, 93(2): 216-231.

[2] KATOH K, KUMA K-I, TOH H, et al. MAFFT version 5: improvement in accuracy of multiple sequence alignment [J]. Nucleic acids research, 2005, 33(2): 511-518.

[3] HSIAO C-F, CHEN Y, LEE C-Y. A generalized mixed-radix algorithm for memory-based FFT processors [J]. IEEE Transactions on Circuits and Systems II: Express Briefs, 2010, 57(1): 26-30.

[4] COOLEY J W, TUKEY J W. An algorithm for the machine calculation of complex Fourier series [J]. Mathematics of computation, 1965, 19(90): 297-301.

[5] 周益民. 图像处理并行算法的研究 [D].成都:电子科技大学, 2006.

[6] 徐金棒. 基于多核多线程的 FFT 算法和堆排序算法的并行优化和实现 [D].郑州:郑州大学, 2011.

[7] 任山. 基于查找表的 FFT CUDA 并行算法研究 [D].长沙:湖南大学, 2017.

[8] 郑伟华. 快速傅立叶变换-算法及应用 [D].长沙:湖南大学, 2015.

[9] GOOD I J. The interaction algorithm and practical Fourier analysis [J]. Journal of the Royal Statistical Society: Series B (Methodological), 1958, 20(2): 361-372.

[10] KOLBA D, PARKS T. A prime factor FFT algorithm using high-speed convolution [J]. IEEE Transactions on Acoustics, Speech, and Signal Processing, 1977, 25(4): 281-294.

[11] WINOGRAD S. The effect of the field of constants on the number of multiplications; proceedings of the 16th Annual Symposium on Foundations of Computer Science (sfcs 1975), F, 1975 [C]. IEEE Computer Society.

[12] BERGLAND G D. A fast Fourier transform algorithm using base 8 iterations [J]. Mathematics of Computation, 1968, 22(102): 275-279.

[13] DUHAMEL P, HOLLMANN H. Split radix'FFT algorithm [J]. Electronics letters, 1984, 20(1): 14-16.

[14] MARTENS J-B. Recursive cyclotomic factorization--A new algorithm for calculating the discrete Fourier transform [J]. IEEE transactions on acoustics, speech, and signal processing, 1984, 32(4): 750-761.

[15] BOUGUEZEL S, AHMAD M O, SWAMY M. A general class of split-radix FFT algorithms for the computation of the DFT of length-$2^{m} $ [J]. IEEE Transactions on signal processing, 2007, 55(8): 4127-4138.

[16] RADER C M. Discrete Fourier transforms when the number of data samples is prime [J]. Proceedings of the IEEE, 1968, 56(6): 1107-1108.

[17] WINOGRAD S. On computing the discrete Fourier transform [J]. Mathematics of computation, 1978, 32(141): 175-199.

[18] JIANG Y, ZHOU T, TANG Y, et al. Twiddle-factor-based FFT algorithm with reduced memory access; proceedings of the Proceedings 16th International Parallel and Distributed Processing Symposium, F, 2002 [C]. IEEE.

[19] FRIGO M, JOHNSON S G. FFTW: An adaptive software architecture for the FFT; proceedings of the Proceedings of the 1998 IEEE International Conference on Acoustics, Speech and Signal Processing, ICASSP'98 (Cat No 98CH36181), F, 1998 [C]. IEEE.

[20] 危疆树. 图像处理算法分析及其并行模式研究 [D].成都:电子科技大学, 2006.

[21] 蒋海峰. 多核平台下基于 CnC 的车辆识别算法的并行优化 [D].长春:东北大学, 2010.