江南大学

JIANGNAN UNIVERSITY

神威·太湖之光并行计算开发入门

尹达恒

江南大学超级计算俱乐部

2019 年 7 月

An Introduction to Parallel Programming on Sunway TaihuLight Supercomputer

Tutorial and tips for parallel programming beginners on Sunway

Based on porting and optimizing of mlpack

By

Howard Yin

Supercomputer Club, Jiangnan University

July 2019

前　言

- 本书实际上是一个教程形式的学习笔记；
- 本书主要为不熟悉并行编程的同学编写，主要讲解具体的工具使用，不会过多涉及抽象的优化算法。玩过并行编程的同学请直接看超算中心发的两个手册，清华的大佬写的比我好，搞过 CUDA 写过 cu 玩过 SIMD 的一看就能懂；
- 本书主要的行文线索是 2018 年 12 月左右启动的机器学习算法库在太湖之光系统中的移植和并行优化项目；
- 本书是超算中心下发的官方教程《“神威・太湖之光”计算机编译系统用户手册 Version 0.9》（本书简称《编译手册》）和《神威・太湖之光并行程序设计与优化（第一版）Ver1.2》（本书简称《优化手册》）中重要知识点的实践总结；
- 比起 Fortran 作者更擅长也更常用 C 语言，C 语言也能胜任超算系统中的大部分工作，因此本书所有代码均使用 C 语言编写，关于编译器也只涉及 Sunway 的 C 语言编译器；
- 本书不区分“‘神威・太湖之光’计算机系统”、“‘神威・太湖之光’系统”、“Sunway 系统”、“神威系统”和“太湖之光系统”，以及处理器名称“SW26010”、“Sunway26010”和“申威 26010”，请读者注意；
- 本书将作为每周的俱乐部工作报告持续更新。

学习本教程前需要掌握的知识

- C 语言，不仅是 C 语言的使用，编译器的运行流程要了解。本书所有代码都是 C 语言，神威编译器也只介绍 C 语言编译器；
- 用命令行操作 Linux 系统（Sunway 的操作系统没有图形界面）；
- 看过某些并行计算教程开头的介绍章节（比如 CUDA 教程），至少知道并行计算是在干嘛。

国家超级计算中心简介

神威・太湖之光超级计算机是由国家并行计算机工程技术研究中心研制、部署于国家超级计算无锡中心的超级计算机。类似的超级计算中心在国内共有 7

个，它们分别是：

- 天津中心：2009 年 6 月，滨海新区与国防科技大学签署合作协议共建国家超级计算天津中心。天津中心由国家科技部、滨海新区、开发区、国防科技大学共同投资 6 亿元建设，于 2009 年底开始投入运营。定位为国家重大科技服务平台、产业技术创新平台、人才聚集培养平台，主要承接国家“863”重大科技专项，是重要的大规模集成电路设计中心和基础软件工程中心及产业化基地；
- 深圳中心：国家超级计算深圳中心又称深圳云计算中心，总投资 12.3 亿元。主机系统由中国科学院计算技术研究所研制、曙光信息产业（北京）有限公司制造，2010 年 5 月经世界超级计算机组织实测确认运算速度达每秒 1271 万亿次。深圳中心立足深圳、主要面向全国、服务华南、港、澳、台及东南亚地区，承担各种大规模科学计算和工程计算任务，同时以其强大的数据处理和存储能力为社会提供云计算服务；
- 长沙中心：国家超级计算长沙中心位于湖南大学校区内，其依托的计算设备是国防科技大学研制的“天河一号”超级计算机。长沙中心总投资 7.2 亿元，运算能力达每秒 300 万亿次，由湖南大学负责运营，国防科技大学提供计算设备和技术支持；
- 济南中心：国家超级计算济南中心主要依托的计算设备是“神威·蓝光”超级计算机，其持续性能可达 0.796PFlops。济南中心秉持立足山东、辐射周边、服务全国的工作思路，积极服务于山东省” 两区一圈一带” 发展战略需求，支持国家重大科技创新和战略性新兴产业，服务地方经济发展。2018 年 8 月，神威 E 级超算原型机在国家超级计算济南中心完成部署。
- 广州中心：国家超级计算广州中心由广东省人民政府、广州市人民政府、国防科技大学、中山大学共同建设，主要依托设备是“天河二号”超级计算机；
- 无锡中心：国家超级计算无锡中心是无锡市政府直属事业单位，由清华大学与无锡市政府共同建设，并委托清华大学管理运营。国家超级计算无锡中心依托的“神威·太湖之光”计算机系统是我国“十二五”期间“863 计划”的重大科研成果，由国家并行计算机工程技术研究中心研制，运算系统全面采用了由国家高性能集成电路设计中心通过自主核心技术研制的国产“申威 26010”众核处理器，是我国第一台全部采用国产处理器构建的世界排名第一的超级计算机；
- 郑州中心：2019 年 5 月，国家超级计算郑州中心获得科技部批复筹建，成

为全国第 7 家批复建设的国家超级计算中心，也是科技部出台认定管理办法后批复筹建的首家国家超级计算中心。郑州中心拟依托郑州大学建设运营，计划于 2020 年上半年建设完成，峰值计算能力预期将达到 100Pflops。

神威・太湖之光简介

“神威・太湖之光”计算机系统 (Sunway TaihuLight) 采用全机水冷、直流供电、高密度组装，配有精确的资源调度管理算法、丰富的并行编程语言和开发环境。2016 年 6 月 20 日国际 TOP500 公布的排名数据中，“神威・太湖之光”计算机系统以每秒 12.54 亿亿次的峰值运算速度和 9.3 亿亿次的持续运算速度高居榜首。“神威・太湖之光”是中国第一台全部采用国内自主研发技术构建的超级计算机。详细的架构描述可见《优化手册》第一章。

申威 26010 简介

“神威・太湖之光”计算机系统采用的处理器是申威 26010(Sunway 26010)，该处理器由上海高性能集成电路设计中心自主研制，其指令集为自主研发的 64 位申威指令集，工作频率 1.5Ghz。该处理器不同于现有的纯 CPU、CPU-MIC、CPU-GPU 架构，而是采用了主-从核架构，每个申威 26010 中有 4 个处理器，每个处理器包含 1 个主核（运算控制核心）和 64 个从核（8x8 核心阵列），主从核的关系类似于 CPU-GPU 架构。详细的架构描述可见《优化手册》1.2 节。

同构计算和异构计算 [1]

同构计算是使用相同类型指令集和体系架构的计算单元组成系统的计算方式。同构超算只单纯使用一种处理器，例如“神威・蓝光”只采用了 8704 片申威 1600。而异构计算主要是指使用不同类型指令集和体系架构的计算单元组成系统的计算方式。常见的计算单元类别包括 CPU、GPU 等协处理器、DSP、ASIC、FPGA 等。例如天河 2 号有 16000 个计算节点，每个节点由 2 片 Intel 的 E5 2692 和 3 片 Xeon PHI 组成，共使用了 32000 片 Intel 的 E5 2692 和 48000 片 Xeon PHI；天河 1A 使用了 14336 片 Intel Xeon X5670 处理器和 7168 片 NVIDIA Tesla M2050 高性能计算卡。相同功耗的情况下，异构超算能获得非常高的理论双精浮点性能和更高的性能功耗比，这也使异构处理器其更受超级计算机系统偏爱。

“神威 • 太湖之光”计算机系统所采用的申威 26010 属于异构众核处理器。不同于“神威 • 蓝光”的同构申威 1600 处理器和天河 2 号的 CPU-GPU 架构，申威 26010 的异构模式相当于将 CPU 和 GPU 集成在一个芯片上，从“神威 • 太湖之光”整体上看，这相当于将 GPU 分散放置到了 CPU 的周围，使得单个处理器同时具有 CPU 核 GPU 的功能。和 CPU-GPU 体系相比，这种架构的优势在于减少了计算过程中的数据传输时间，从而提高整体的运算速度。

目 录

第 1 章　Sunway 系统的基本操作

1.1　登录

未来 Sunway 官网和计算系统的登录方式可能变化，此处的登录方法仅供参考。超算中心下发了两套用户名和密码，一套用于登录 VPN，一套用于登录 SSH。

1.1.1　登录 Sunway 系统 SSL VPN

1. 使用 IE 浏览器[1]进入神威・太湖之光官网 www.nsccwx.cn；

2. 在右上角“登录”处选择一个运营商进行登录（图1.1a）；

3. 在弹出的警告页面[2]中选择“详细信息”→“转到此页面”（图1.1b）；

4. 在登录界面输入俱乐部下发的 VPN 用户名密码进行登录（图1.1c）；

5. 第一次登录完成后浏览器会自动下载 EasyConnect 工具，此工具为后续登录使用（图1.1d和图1.1e）。

再次登录时，打开第一次登录完成后自动下载的 EasyConnect 工具输入俱乐部下发的 VPN 用户名密码即可。

1.1.2　SSH 登录超算系统

1. 登录 VPN；

2. 在弹出的页面找到可用资源的 IP 地址（图1.1f）；

3. 在命令行窗口输入“ssh [俱乐部下发的用户名]@[可用资源的 IP 地址]”命令进行登录[3]，登录时会提示输入密码[4]，此处密码为 SSH 登录密码（图1.1g）；

4. 出现 bash 界面说明登录成功（图1.1h）。

[1]亲测 Edge 和 Chrome 在第一次登录时无法下载 EasyConnect 工具，卡在加载界面

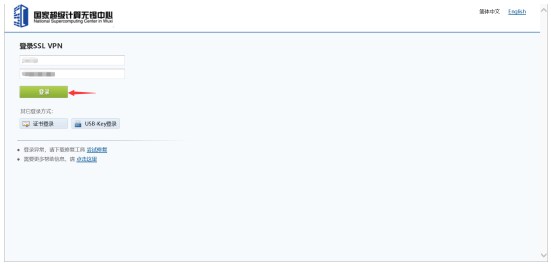
[2]截至本章作成日 2018 年 12 月 1 日，太湖之光的官方网站服务器在使用自主生成的 SSL 证书进行 https 连接，这类非信任机构生成的证书会被现在的浏览器识别为不安全

[3]这里的用户名是 SSH 用户名

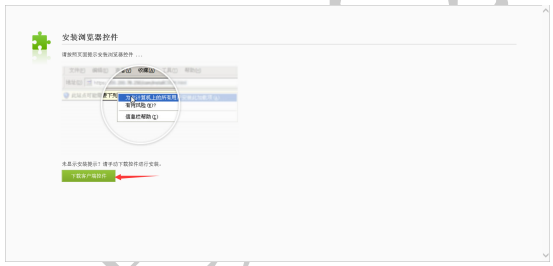
[4]多次登录有时会出现 WARNING 和登录失败，大部分情况下这都是神威系统为了安全对 SSH 登录验证的公钥进行定时修改所致（猜测）。这时只要删除用户目录下的 C:/用户/用户名/.ssh/known_hosts 文件重新登录 SSH 即可（这个文件记录了所有登录过的 SSH 服务器的公钥，再次登录时会进行公钥比对，如果不符则登录失败）

(a) 官网主页

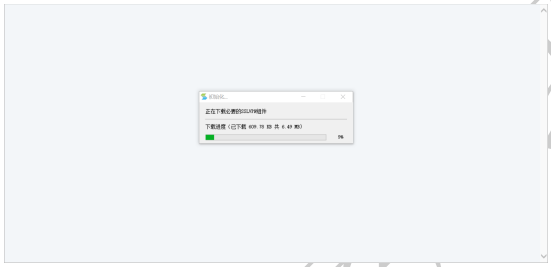
(b) 进入登录页面



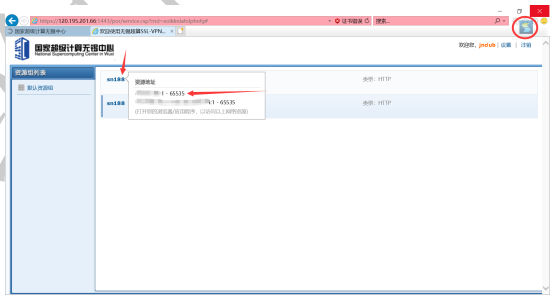
(c) 登录 VPN



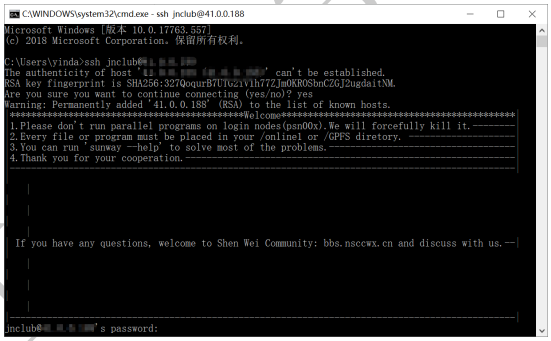
(d) 下载 EasyConnect



(e) 安装 EasyConnect



(f) 找可用资源



(g) 用可用资源 IP 登录 SSH

(h) 登录成功

图 1.1 超算系统登录流程

1.2 第一个程序：矩阵相加

矩阵计算是所有并行计算的基础，第一个并行程序以矩阵相加程序为例，介绍在单个 SW26010 处理器上编程时 Sunway 编译器和 Athread 的一些基本操作。矩阵相加程序的要求非常简单：将两个 64x2048 的矩阵相加。

1.2.1 单个 SW26010 处理器并行计算流程

正如前言中所说，每个 SW26010 芯片中都包含 4 个处理器，每个处理器有 1 个主核和 64 个从核，第一个程序的编写将着眼于在一个处理器上的编程，程序只使用 1 个主核和 64 个从核。

1. 运行于主核上的主程序启动 N 个并行执行的线程 (N≤64)，每个线程都在一个从核上执行；

2. 在并行执行的线程中，每个线程都从主存储中**不同的位置**取出数据放入从核各自的局部存储中[56]；

3. 从核在自己的局部存储上进行计算；

4. 从核计算完成后，都分别将计算结果放回到主存储中后结束自身的线程；

5. 在从核运行过程中主程序进行其他其他操作并等待所有从核线程结束；

6. 所有的从核线程结束（从核函数退出）之后，主程序进行下一步并行操作或结束。

1.2.2 相关 Athread 函数

下面是矩阵相加程序中使用到的并行函数相关必要知识的介绍，更加详细的描述见《编译手册》第五章“加速线程库”。

1.2.2.1 在主核中调用的 Athread 函数

1. athread_init() 核组初始化，在所有 athread 操作之前都要进行调用；

2. athread_set_num_threads([要设置的线程总数]) 设置下一次并行操作启动的线程总数，如果不设置此项值则下一次并行操作 (athread_spawn) 启动 64 个线

[5]在一般的并行编程中都有在并行线程中获取线程编号的方法，并行线程即通过线程编号计算出自己要从哪个位置取数

[6]从主存储中取出数据放入从核的局部存储使用称作 DMA(Direct Memory Access，直接内存存取) 的方法，DMA 运行独立于主核和从核，可以在主核和从核进行计算的同时传输数据，在基于传输的并行优化方面尤其有用

程；

3. athread_spawn([函数指针],[传入参数]) 在从核上启动并行线程，[函数指针] 为指向从核函数入口的指针 (从核函数名)，且该指针需要在文件开头使用一个 Athread 库中的宏“extern SLAVE_FUN(从核函数名)()”进行声明；[传入参数] 为向每个从核函数传递的实参该参数会在运行时传递到每个并行从核函数线程中；

4. athread_join() 等待所有从核线程结束，程序执行到此函数时会阻塞直到所有从核线程结束才会执行下一步；

5. athread_halt() 关闭线程组流水线，在所有并行操作完成之后才能调用此函数，在程序结束时调用，调用后运算核心将无法在本进程中再次使用；

6. gettimeofday([变量指针],NULL)：在本例程中用此函数获取程序运行时间统计并行算法性能。

1.2.2.2 在从核中调用的 Athread 函数

1. athread_get_id() 获取线程逻辑标号，一般在从核程序中通过此函数的值判断应该从主存储的哪个位置取数据；

2. athread_get([传输模式],[源地址],[目的地址],[数据量],[回答字地址],[],[],[]) 通过 DMA(见注6) 方法从主存中读取数据写入从核局部存储。各形参含义为：

- 传输模式：DMA 传输命令模式，本例程中只涉及使用 PE_MODE；
- 源地址：要传输的数据在主核中的地址 (在主核程序中的变量名)。对于数组形式的数据，此处填数组变量的首地址，就像下面这样[7]。多维数组的取址方法以此类推。

```
&源数组名[取数位置]//一维数组首地址位置
&源数组名[取数位置1][取数位置2]//二维数组首地址位置

```

上面写出的源数组名数组名需要在从核程序开头进行“extern”声明，如下所示。

```
extern 数据类型 源数组名;//文件开头声明主核中的某个数组

```

[7]这里的取数位置即是从核函数要从何处取数的标志 (见注5)，该值一般由 athread_get_id() 获取到的线程逻辑标号计算得出

- 目的地址：从主存中取得的数据放入局部存储的哪个地址 (变量) 中。对于数组形式的数据其写法和源地址相同，且数组名需要在从核函数文件开头进行“__thread_local”声明，如下所示。

```
extern 数据类型 源数组名;//文件开头声明主核中的某个数组
```

- 数据量：从源地址开始读取多少**字节**数据到局部存储 (目的地址) 中。对于数组形式的数据，此参数的值为 [要读多少个数]*[此数组的数据类型占多少字节][8]。

- 回答字地址：当 athread_get 数据传输完成时，回答字地址中的值加 1。多个 athread_get 同时运行可以共用一个回答字，每个 athread_get 运行结束时都会使回答字地址中的值加 1。一般在等待 athread_get 传输完成的地方会有“while([回答字]!=[共用此回答字的 athread_get 函数个数]);”。回答字变量在从核函数文件开头以“__thread_local volatile”声明。

- 最后三个参数和 DMA 主存跨步有关，本例程不涉及。

1.2.3 并行编程思路

并行编程程序分主核和从核两个程序流程，请理解上文所述的各函数的作用并理解下列流程后自行编写程序。参考程序见附录A。

主核程序：主核程序中先用传统方法求一下矩阵和统计运行时间，和下面的并行优化后的时间对比，感受一下并行加速的效果。

1. 生成两个 64x2048 矩阵；
2. athread_init() 初始化；
3. athread_spawn() 启动 64 个从核线程，每个线程处理每个 64X2048 矩阵中的 2048 个数；
4. athread_join() 等待线程全部结束；
5. athread_halt() 关闭线程流水线；
6. 输出结果，退出程序。

在上述操作中的开头和结尾调用 gettimeofday() 获取当前时间相减得到并行程序的运行时间。

[8]实测在太湖之光的编译系统中，int 型和 float 型占 4 个字节，long 型和 double 型占 8 个字节

从核程序：

1. athread_get_id() 获取线程逻辑标号；

2. athread_get() 根据线程逻辑标号从主核程序变量中取得要用的数据，每个线程取在两个 64x2048 矩阵中各取 2048 个数；

3. 计算相加结果；

4. athread_put() 将结果写回主核程序变量中；

5. 退出程序。

1.3 程序的编译

“神威 • 太湖之光”系统的编译环境有两种，一种是服务于高速计算系统的编译环境，即 SW26010 搭建的神威主系统；另一种服务于辅助计算系统，该系统是常见的 Intel X86 CPU + GPU 结构服务器。这里只介绍高速计算系统环境下的程序编译。本节主要内容来自《优化手册》2.5 节“编译环境”。

1.3.1 Sunway 程序编译简介

目前的“神威 • 太湖之光”系统支持的编程语言主要包括 C 语言 (C99)、C++ 语言 (C++03 和 C++11) 和 Fortran(Fortran2003)，其中 C++ 目前还不能编译从核程序。

神威系统的 C 语言编译器指令为“sw5cc”，编译模式分为主核和从核两种[9]，编译主核程序的指令为：

```
sw5cc -host [选项] 文件名.c
```

编译从核的指令为：

```
sw5cc -slave [选项] 文件名.c
```

主核程序和从核程序编译完成后，则使用下面这个命令进行混合链接生成可执行程序：

[9]正如前言中所说，异构计算使用不同类型指令集和体系架构的计算单元组成系统，因此在编译程序时主核和从核使用的编译器并不相同

```
sw5cc -hybrid [选项] 主核文件名.o 从核文件名.o
```

本章只介绍编译器的基本使用方法，不过多涉及编译器的编译选项，详细了解编译选项可见《优化手册》表格 2-2。

1.3.2 用命令行编译程序

假设在1.2.3节编写的主核程序源文件名“master_arrAdd.c”、从核程序源文件名“slave_arrAdd.c”，则可以输入下面这三条指令对源文件进行编译和链接：

1. 编译主核，-c 选项表示为每个源文件生成一个.o 文件但不进行链接

```
sw5cc -host -c master_arrAdd.c
```

2. 编译从核

```
sw5cc -slave -c slave_arrAdd.c
```

3. 链接，-o arrAdd 选项表示输出的可执行文件名为“arrAdd”

```
sw5cc -hybrid master_arrAdd.o slave_arrAdd.o -o arrAdd
```

编译完成后即可在当前目录下看到一个可执行文件“arrAdd”。接下来介绍运行这个可执行文件的方法。

1.4 程序的运行

和上一节介绍的编译环境一样，神威系统的运行也分高速计算系统运行和辅助计算系统运行两种，这里也只介绍程序在高速计算系统中的运行，主要内容来自《优化手册》2.6 节“作业提交”。

1.4.1 Sunway 程序运行简介

“神威 • 太湖之光”的高速计算系统计算资源由一个作业管理系统进行管理，其本质是一个队列。在高速计算系统上运行的程序称为“作业”，要运行某个作

业时，将这个作业提交到作业管理系统的队列中，作业管理系统按照先进先出方式取出队列中的作业放到计算节点上运行。

俱乐部目前持有的账号可以向免费的开发调试计算队列提交作业：高速计算系统的 q_sw_expr 和辅助计算系统的 q_x86_expr。这两个队列中每个作业的最长运行时间为 60 分钟，最大并行规模为 64。

1.4.2 用命令行运行程序

神威系统提交作业的指令为“bsub”，其指令选项内容丰富，此处不一一展开，详细的选项可见《优化手册》2.6.2 节，这里只介绍几个常见的选项。

使用“bsub”指令运行1.3.2中编译完成的可执行程序“arrAdd”，可键入如下指令：

```
bsub -I -b -q q_sw_expr -n 1 -cgsp 64 ./arrAdd
```

指令选项解释：

- -I：使作业输出在当前窗口；
- -q q_sw_expr：将作业提交到 q_sw_expr 计算队列；
- -b：指定从核栈位于局部存储，该选项使得从核程序先全部读入从核局部再运行，可以减少主从核数据传输的次数，对于并行计算的速度提升非常重要；
- -n 1：指定程序要用的核组个数为 1；
- -cgsp 64：指定每个核组内要用的从核个数为 64，由于一个 SW26010 一个核组只有 64 个从核，因此该值不能大于 64。该选项即影响程序中 athread_spawn() 生成的从核线程的个数；
- ./arrAdd：要执行的可执行文件位置。

若使用附录A中的程序，在程序执行完成后可得到并行和非并行计算的结果与时间对比。

1.5 使用 Linux make 进行编译和运行

1.5.1 Linux make 和 Makefile 介绍其一

在装有 GNU Make 的 Linux 系统上，make 是一条可执行指令，该指令读入一个名为 Makefile 的文件，通过执行这个文件中指定的指令完成程序的编译、安

装和运行。该指令有如下功能：

1. 使最终用户可以在不知道构建细节的情况下构建和安装软件，构建细节都记录在所提供的 Makefile 中，用户只需要在 Makefile 文件所在目录下键入 make 指令即可；

2. make 可以通过读取和记录文件修改时间自动知道那些文件需要更新（重新编译），并且它也会通过 Makefile 中所记录的文件依赖关系自动决定文件更新的适当顺序，使得相关的文件一并更新；

3. make 本质上只是一个系统指令的执行器，只要是能通过键入指令完成构建和运行的程序都可以通过 make 完成构建和运行，因此 make 的使用不限于任何一种特定的语言。对于程序中任何一种非源文件，makefile 文件中可以指定 shell 指令去生成它。shell 指令可以执行编译器生成目标文件，执行链接器生成可执行文件，执行 ar 更新库文件，执行 TeX（一个文本排版软件）或 Makeinfo 去格式化文档等。

1.5.2 Makefile 的格式

Makefile 文件的语法可以看作是一系列的任务定义，定义任务的文本格式非常简单：

```
target:[dependencies \dots]
  command
  command
  \dots
```

其各部分的用处如下：

- target：定义任务名称。每个 Malefile 文件都包含有多个任务，每个任务都有一个唯一的 target 指定这个任务的名称。任务的名称只是这个任务的一个代号，但是为了可读性，任务的编写和命名规则一般有如下约定：
  - 将生成中间文件、编译、清理生成文件、运行等任务分开编写；
  - 如果这个任务的作用是生成一个文件，则任务名是要生成的文件名；
  - 如果这个任务的作用是编译整个工程，则任务名是“build”或是最终的可执行文件名；
  - 如果这个任务是清理生成文件，则任务名为“clean”；

– 如果这个任务是运行，则任务名为“run”。

当在 Malefile 中定义了一个任务后，可以使用“make target”指令 Malefile 中定义的某个特定任务；而若直接键入“make”，则默认运行 Makefile 中定义的第一个任务。

dependencies … ：依赖项。该项指定当前任务与哪些文件有关或是在哪些任务完成之后才能运行，dependencies 是文件名称或其他任务的任务名称（target），多个名称以空格分隔。在 make 指令调用此任务时，make 程序对 dependencies 中的每个名称的处理如下：

1. 如果有该名称对应的文件，则检查文件的修改日期，将其与上一次执行时存储的修改日期对比，若不同则说明文件被修改。如果在 dependencies 对应的文件中检测到任何一个文件被修改，则先执行该名称对应的任务（如果有的话，否则只执行本任务[10]）再执行本任务；

2. 如果有该名称对应的任务，则检查该任务的 dependencies，规则同上。

递归式地运行以上两步，直到检查的任务中没有名称对应任务，从而能使得一个文件被修改时执行且只执行相关的任务进行中间文件和输出文件的编译更新；

• command：完成任务要执行的指令。该项指定当前任务要完成哪些指令，每个指令前必须要以 tab 缩进，指令间以换行分隔。当调用一个任务时，该任务下定义的指令将被按顺序执行。

1.5.3 编写 Makefile

以1.2.3中编写的程序和1.3.2中的编译过程为例编写一个简单的 Makefile：

```
arrAdd:master_arrAdd.o slave_arrAdd.o
  sw5cc -hybrid master_arrAdd.o slave_arrAdd.o -o arrAdd
master_arrAdd.o:master_arrAdd.c
  sw5cc -host -c master_arrAdd.c
slave_arrAdd.o:slave_arrAdd.c
  sw5cc -slave -c slave_arrAdd.c
clean:
  -rm master_arrAdd.o slave_arrAdd.o arrAdd
run:arrAdd
  bsub -I -b -q q_sw_expr -n 1 -cgsp 64 ./arrAdd
```

[10]想想看这就是以要生成的文件名作为任务名的好处

按照前面所讲的文件格式和命名规则，该 Makefile 定义了 5 个任务，其中默认任务为“arrAdd”，该文件生成可执行文件“arrAdd”；此外还有一个运行任务“run”和清理任务“clean”。该文件放在和源文件同一目录下，则可在该目录下运行“make”编译程序或运行“make run”执行程序等操作。如果一切正常，上述并行优化方法的实际运行时间应该是用双重 for 循环计算的 1/200。

1.6 本章总结

1.6.1 知识点概括

本章主要介绍了在神威系统编写程序的流程和必须要知道的一些基本操作，内容可大致概括为图1.2。

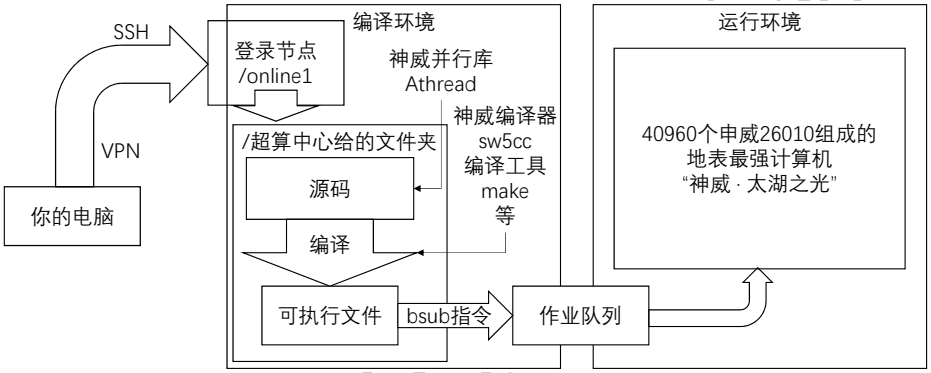


图 1.2 第 1 章的知识点概括

1.6.2 练习

- 用 Athread 实现并行的矩阵减法，并与无并行的矩阵减法比较运行时间；
- 用 Athread 实现并行的矩阵乘法，并与无并行的矩阵乘法比较运行时间。

第 2 章　SIMD 优化

前面介绍的 Athread 是专为神威系统设计的原生并行计算库，属于神威系统中支持最好、功能最齐全的并行库。但是除 Athread 之外，神威系统中还有几个按照国际标准设计的标准并行计算库，比如 OpenACC 和 MPI 等，这些库存在的意义主要是适应各种不同的编程和优化需求，并且方便神威系统移植一些用标准库编写的软件，从而扩大神威系统的适用范围。SIMD 就是这些库之一，比起 Athread，SIMD 技术发挥作用的系统层次更加偏向底层，其功能也更加简单。本章将着重于介绍神威系统中的 SIMD 技术，主要内容对应于《编译手册》第八章和《优化手册》5.4 节。

2.1　SIMD 介绍

2.1.1　SIMD 是什么

SIMD 全称单指令流多数据流 (SingleInstruction Multiple Data)，是一种同时对一组数据（又称“数据向量”）分别执行相同的操作的并行技术，即利用数据级并行来提高计算效率。SIMD 和上一章所讲的多线程从核并行类似[1]，都是让大量相同的操作在同一时间段内并发进行。但 SIMD 与多线程从核并行的区别主要来自以下两点：

- SIMD 将要并行处理的数据存放于**寄存器**中，从核并行将数据读入到从核内存；
- SIMD 的并行是**指令**的并行（即每执行一条指令时都同时复制给多个数据执行），多线程从核并行是线程（把多个指令组成的一段程序整个复制到多个从核上执行）的并行。

可以看出 SIMD 虽然和前面介绍的从核并行有很多相似点，但是其实它们的实现是完全不同的。SIMD 目前使用广泛的领域主要是多媒体文件处理。Intel 家的 SIMD 技术有 MMX 和 SSE[2]；AMD 家的有 3D Now! 技术[3]等。

[1]回忆一下 athread_spawn() 函数

[2]这些技术起初是用 SIMD 加速多媒体文件的处理过程，现在经常可以在一些深度学习框架的编译选项中发现这些东西

[3]也是个多媒体指令集。顾名思义，和三维图像处理有关的

2.1.2 神威系统中的 SIMD 介绍

2.1.2.1 支持的运算

申威 26010 处理器的主核和从核都支持 SIMD，且支持的 SIMD 宽度都是 256 位，支持的操作如表2.1。

表 2.1 神威系统 SIMD 支持的运算

支持运算 / 运算类型	逻辑	移位	加减	乘除
8 个 32 位定点运算	√	√	√	×
1 个 256 位定点运算	√	√	×	×
4 个 64 位定点运算	√	√	√	√

2.1.2.2 使用形式

神威编译器在 C 语言上扩展了一些 SIMD 数据类型和函数，扩展的函数基本都是直接映射到 SIMD 指令，在编译时由编译器自动进行 inline，因此编程时不需要了解具体的 SIMD 指令，只需要调用函数即可进行 SIMD 操作。神威编译器的 C 语言 SIMD 扩展可以在 C 语言一级获得与汇编程序一样的性能。

2.2 神威 SIMD 使用入门

2.2.1 SIMD 数据类型

神威编译器 SIMD 扩展的数据类型如表2.2。注意 SIMD 扩展数据类型本质只是多个基本数据类型的拼接，但是和 C 语言中的 `int` 、`int` 等类型一样是值类型而不是像 `int[8]` 、`float[4]` 一样的指针类型。

2.2.2 SIMD 的等号赋值和数据转换

2.2.2.1 标准类型到 SIMD 扩展类型等号赋值

标准类型到 SIMD 扩展类型的赋值是以值扩展方式进行的，具体规则如下：

- 标准类型到 `int256` 型：先转化为 `long` 型然后把值复制到低 64 位，符号位复制到高 192 位；
- 标准类型到非 `int256` 型：先转化为扩展类型对应的标准类型再**复制**指定份数放入扩展类型中。例如变量 `floatv4 fv; int iv;` 的赋值 `fv=iv;` 或

表 2.2 神威编译器 SIMD 扩展数据类型

类型	含义
intv8	8 个 32 位有符号整型
uintv8	8 个 32 位无符号整型
int256	1 个 256 位有符号长整型
uint256	1 个 256 位无符号长整型
floatv4	4 个 64 位单精度浮点
doublev4	4 个 64 位双精度浮点

`fv=1;` 是先将 `iv` 或整型常数 1 转化为浮点型再复制 4 份放入 `fv` 中；

- 数组赋值：例如数据类型 `floatv4 fv;` 可以有 `fv = {1.0,2,3.0};` 这样的赋值，数组赋值从低位到高位，缺省补 0，其赋值结果为：

192~255 位	128~191 位	64~127 位	0~63 位
0	3.0	2.0	1.0

又比如数组 `float fs[4]` 可以有 `floatv4* fv = (floatv4*)(&fs[0]);` [4]。

2.2.2.2 SIMD 扩展类型之间等号赋值

SIMD 数据互相之间赋值是以传值模式进行的。相同数据类型的 SIMD 数据互相赋值和正常的 C 语言赋值一样，不同类型的数据赋值语句的数据转换规则如下所示：

- floatv4 和 doublev4 赋值：单精度和双精度浮点数扩展类型互相赋值时会自动进行数据类型的转换，源扩展类型值中的每个数分别转换后放入目的扩展类型中；
- int256 和 intv8/uintv8：256 位整型和 8 个 32 位整型互相赋值时没有数据类型的转换，数据不发生任何变化。

2.2.2.3 SIMD 扩展类型到标准类型等号赋值

只取扩展类型最低位的部分。

[4]这条语句如何理解？指针很有趣，请结合前文所讲的扩展类型的本质仔细揣摩。有疑问请看《编译手册》8.3.6 节

2.2.3 SIMD 宏赋值

宏赋值就比较好理解了，就是向一个宏定义中“输入”几个数得到存入这些数的扩展类型。

2.2.3.1 输入多个数的赋值宏

这类宏接受几个独立的数值，返回存入这些数的扩展类型，具体的宏如表2.3。

表 2.3 神威编译器中输入多个数的 SIMD 赋值宏

宏定义	输入参数	输出类型
simd_set_intv8(...)	8 个 int	intv8
simd_set_uintv8(...)	8 个 unsigned int	uintv8
simd_set_int256(...)	4 个 long	int256
simd_set_uint256(...)	4 个 unsigned long	uint256
simd_set_floatv4(...)	4 个 float	floatv4
simd_set_doublev4(...)	4 个 double	doublev4

2.2.3.2 输入数组首地址的赋值宏

比起用上面介绍的输入多个数的赋值宏，在并行优化中一般更倾向于使用速度更快[5]的输入数组首地址的赋值宏对扩展数据类型进行赋值。具体的宏如表2.4。其中 `simd_load(.,.)` 和 `simd_store(.,.)` 要求**对界**，即输入的数组起始位置必须是数组起点的第 $256 \times n (n \in \mathbf{N})$ 位[6]；而 `simd_loadu(.,.)` 和 `simd_storeu(.,.)` 不要求对界存储。这两个赋值宏一般搭配使用。

输入数组首地址的赋值宏一次必须读取 256 位。在实际的优化中，如果要输入的数组在计算到最后末尾不足 256 位，一般来说直接将剩下的数据直接串行计算或从末尾开始读最后的数据即可。

[5]显而易见，输入单个数赋值需要对内存空间的离散访问，而输入数组的意味着访问连续存储，速度势必会更快

[6]不对界的访问会引起异常，然后由操作系统模拟，产生很大的性能损失

表 2.4 神威编译器中输入数组首地址的 SIMD 赋值宏

宏定义	对应操作	输入参数
`simd_load(.,.)` `simd_loadu(.,.)`	从输入的数组首地址开始读取 256 位数据装入输入的扩展类型变量中	扩展类型变量[7], 数组首地址[8]
`simd_loade(.,.)`	如果输入的扩展类型变量是浮点型，则从输入的数组首地址开始读取 64 位数据复制 4 份装入输入的扩展类型变量中，否则若是整型则读取 32 位数据复制 8 份装入	
`simd_store(.,.)` `simd_storeu(.,.)`	将扩展类型变量中的值写入到从输入的数组首地址开始的 256 位中	

2.2.4 SIMD 数据输出宏

- 控制台输出，效果同 `printf("[ %d, %d, %d, %d, %d, %d, %d, %d ]", v[7], v[6], v[5], v[4], v[3], v[2], v[1], v[0])` 或 `printf("[ %f, %f, %f, %f ]", v[3], v[2], v[1], v[0])`，不同的函数对应以不同的数据类型输出。输出字符串为以方括号包围的多个数，数字之间以逗号分隔；
- 文件输出，效果同 `fprintf(file, "[ %d, %d, %d, %d, %d, %d, %d, %d ]", v[7], v[6], v[5], v[4], v[3], v[2], v[1], v[0])` 或 `fprintf(file, "[ %f, %f, %f, %f ]", v[3], v[2], v[1], v[0])`，输出字符串格式同上。

其宏定义如表2.5所示。

表 2.5 神威编译器 SIMD 输出操作宏定义

控制台输出宏	文件输出宏	对应的类型
`simd_print_intv8(...)`	`simd_fprint_intv8(FILE*,...)`	`intv8`
`simd_print_uintv8(...)`	`simd_fprint_uintv8(FILE*,...)`	`uintv8`
`simd_print_int256(...)`	`simd_fprint_int256(FILE*,...)`	`int256`
`simd_print_uint256(...)`	`simd_fprint_uint256(FILE*,...)`	`uint256`
`simd_print_floatv4(...)`	`simd_fprint_floatv4(FILE*,...)`	`floatv4`
`simd_print_doublev4(...)`	`simd_fprint_doublev4(FILE*,...)`	`doublev4`

2.2.5 SIMD 运算

SIMD 运算有运算符和运算宏两种实现方式，这两种实现方式是等价的，但是运算符只能执行一部分运算，运算宏能执行所有支持的运算。基本的算术运算、逻辑运算、位运算的运算符和宏如表2.6。支持的全部运算宏定义请见《编译手册》8.5.3 和 8.5.4 节。

表 2.6 神威编译器 SIMD 基本算术运算符宏定义

运算	宏 / 类型 / 符	intv8	floatv4	doublev4
加	+	simd_vaddw	simd_vadds	simd_vaddd
减	-	simd_vsubw	simd_vsubs	simd_vsubd
乘	*	无	simd_vmuls	simd_vmuld
除	无	无	simd_vdivs	simd_vdivd
按位与	&	simd_vandw	无	无
按位或	|	simd_vbisw	无	无
异或	^	simd_vxorw	无	无
左移	«	simd_vsllw	无	无
右移	»	simd_vsrlw	无	无

2.3 神威 SIMD 优化示例

SIMD 基础学会了就可以拿来进行实际的优化了。本节将使用 SIMD 对1.2.3节的参考程序（附录A）进行进一步的优化。

1.2.3节的程序中，只有从核在进行并行计算，因此 SIMD 优化只需要在从核代码上进行。其基本思路如下：

1. 循环分裂：将从核代码中的循环分裂为子循环，每个子循环由 8 个 `int` 加法组成；

2. 变量提取：将子循环中的 8 个加法操作的加数变量和结果变量提取出来，用 simd_load 分别装载到 SIMD 扩展变量中；

3. 计算：执行 SIMD 加法运算；

4. 结果写回：用 simd_store 将计算结果从 SIMD 扩展变量写回结果数组中。

以此编写的程序如附录A.3所示。此外，神威编译器编译带有 SIMD 库的程序时需要加上`-msimd` 编译选项，将该选项加在 Makefile（见1.5.3节）从核编译指令后即可。

依然是命令行打 `make run` 运行程序，如果一切正常，可以看到 SIMD 优化后的程序和1.2.3节原本的并行程序相比性能提升并不明显，这是由于在矩阵相加运算中，并行程序的主要时间消耗在 `athread_get` 和 `athread_put` 读写数据的 DMA 操作上，而 SIMD 是对计算过程的优化，因此优化效果不明显。

2.4 本章总结

2.4.1 知识点概括

本章主要讲解的神威系统中 SIMD 的使用，主要知识点如图2.1所示。

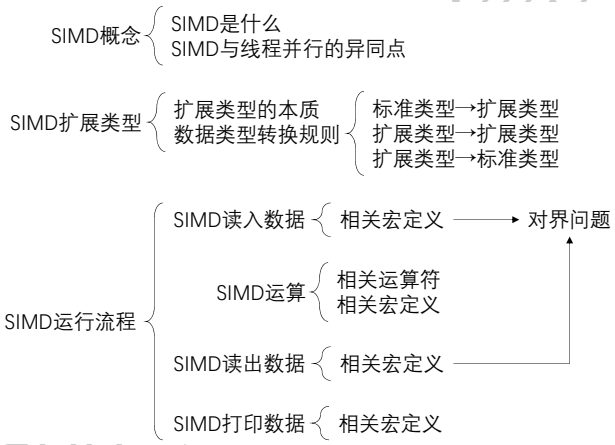


图 2.1 第 2 章的知识点概括

2.4.2 练习

- 用 Athread 实现并行的矩阵减法，并用 SIMD 优化之；
- 用 Athread 实现并行的矩阵乘法，并用 SIMD 优化之。

第 3 章 Darknet 移植

3.1 Darknet 介绍

Darknet 是一个开源的 C 语言神经网络框架，由著名的机器视觉专家 Joseph Redmon 开发 [2]。Darknet 性能优异，安装简单，在目标识别领域有广泛应用的 YOLO 模型 [3] 的最初版本就是在这个框架下开发的。作为作为“神威 • 太湖之光”并行编程的入门练习，Darknet 有如下优点 [4]：

- 易于安装：在 makefile 里面选择自己需要的附加项（cuda，cudnn，opencv 等）直接 make 即可；
- 没有任何依赖项：整个框架都用 C 语言进行编写，可以不依赖任何库，连 opencv 作者都编写了可以对其进行替代的函数；
- 结构清晰：其框架的基础文件都在 src 文件夹，而定义的一些检测、分类函数则在 example 文件夹，可根据需要直接对源代码进行查看和修改；
- 友好 python 接口：虽然 darknet 使用 c 语言进行编写，但是也提供了 python 的接口，通过 python 函数，能够使用 python 直接对训练好的.weight 格式的模型进行调用；
- 易于移植：该框架部署十分简单，且可以根据机器情况，使用 cpu 和 gpu，特别是检测识别任务的本地端部署，darknet 会显得异常方便。

此外，由于 Darknet 是一个神经网络框架，其中的主要运算任务都可以以矩阵计算的形式所表述，这使得 Darknet 的移植和优化有助于读者对矩阵和并行更加深入的理解。最后，Darknet 框架整体结构比较成熟完整，读者能在移植和优化过程中了解到优秀的机器学习框架的架构与编写方式，对软件架构能力的提升有一定帮助。

3.2 Darknet 源码结构

从 Joseph Redmon 的个人网站或 Github 上下载得到的 Darknet 源码包解压后可得到如图所示的文件和文件夹，各文件和文件夹的作用如下 [4]：

- cfg 文件夹内是一些模型的架构，每个 cfg 文件类似于 caffe 的 prototxt 文件，通过该文件定义的整个模型的架构；

- data 文件夹内放置了一些 label 文件，如 coco9k 的类别名等，和一些样例图，主要在训练和测试时使用；
- src 文件夹内是最底层的框架定义文件，所有层的定义等最基本的函数全部在该文件夹内，是 Darknet 框架的源码所在；
- examples 文件夹是更为高层的一些函数，如检测函数，识别函数等，这些函数直接调用了底层的函数，是对底层函数的封装；
- include 文件夹，顾名思义，存放头文件的地方；
- python 文件夹里是使用 python 对模型的调用方法，基本都在 darknet.py 中。要实现 python 的调用，还需要用到 darknet 的动态库 libdarknet.so；
- scripts 文件夹中是一些脚本，如下载 coco 数据集，将 voc 格式的数据集转换为训练所需格式的脚本等；
- 一系列和功能无关的 license 文件；
- 本章的重点 Makefile 文件，用于框架的编译和运行。

3.3 Darknet 的移植

3.3.1 Linux make 和 Makefile 介绍其二

在1.5.1节中已经介绍过 Makefile 的文件结构和 Linux make 的基本使用方法，在本节中将进一步介绍 Makefile 文件中的变量定义和计算，为理解 Darknet 的编译过程打下基础。

在 Makefile 文件中，如果有某些指令和名称比较长或是需要在文件不同位置多次使用，为了保证可读性和修改的方便则需要用类似 C 语言宏定义的方式，用一些“宏”（更准确地说应该是变量）代表这些指令或名称。Linux make 和 Makefile 即提供了这样的功能，这种功能的存在也使得 Makefile 被称为“make 脚本”[1]。这一功能可以概况为如下几种语法：

- 变量定义：

```
变量名=值；
```

[1]为啥叫“脚本”？“脚本”的英文是“script”，可以联想两种解释型编程语言：js 和 python。js 全称是“JavaScript”，python 的源代码文件也叫“python script”，它们都是以解释器解释脚本文件中源代码的方式运行的。make 也可以看作是一个解释型编程语言的解释器，Makefile 就是它读入并解释的脚本文件，1.5.1节和本节就是在介绍这个编程语言的语法

- 变量调用：

```
$(变量名)
```

- 变量修改：

```
变量名+=$(变量名)或值
```

```
变量名=$(变量名)或值+$(变量名)或值+...
```

- 变量比较和 if 语句：

```
ifeq ($(变量名), $(变量名)或值)
[一些操作]
endif
```

- 内嵌函数：

```
$(函数名 变量1,变量2,...)
```

- 静态模式：可以将多个任务合并于一个任务定义中，如下任务运行前 `$@` 将被文件路径`%.o` 替代、`$<` 将被文件路径`%.cpp` 替代，生成每个.o 和每个.cpp 一一对应的多个指令：

```
[文件路径]%.o: [文件路径]%.cpp [其他dependences]
    [一部分指令] $< [一部分指令] $@

```

而下面这种的 $@ 将被 target 替代、$^ 被 dependences 替代，但没有上面那种多重替代的效果：

```
[target]: [dependences]
    [一部分指令] $^ [一部分指令] $@
```

Makefile 中的所有变量均是字符串，变量修改中的“+”表示的就是字符串的连接, 变量比较就是字符串的比较；字符串的调用可以在文件的任何地方，Linux make 在运行时会自动将 Makefile 中的变量调用替换为变量对应的字符串[2]。例如1.5.3中的 Makefile：

```
arrAdd:master_arrAdd.o slave_arrAdd.o
    sw5cc -hybrid master_arrAdd.o slave_arrAdd.o -o arrAdd
master_arrAdd.o:master_arrAdd.c
    sw5cc -host -c master_arrAdd.c
slave_arrAdd.o:slave_arrAdd.c
    sw5cc -slave -c slave_arrAdd.c
clean:
    -rm master_arrAdd.o slave_arrAdd.o arrAdd
run:arrAdd
    bsub -I -b -q q_sw_expr -n 1 -cgsp 64 ./arrAdd
```

把其中的输出文件名“arrAdd”、两个.o 文件和 bsub 的运行选项用变量表示，可将 Makefile 改写如下：

```
EXEC=arrAdd
OBJS=master_arrAdd.o slave_arrAdd.o

OPT=-I -b
OPT+=-q q_sw_expr
OPT+=-n 1
OPT+=-cgsp 64

$(EXEC):$(OBJS)
    sw5cc -hybrid $(OBJS) -o $(EXEC)
master_arrAdd.o:master_arrAdd.c
    sw5cc -host -c master_arrAdd.c
slave_arrAdd.o:slave_arrAdd.c
    sw5cc -slave -c slave_arrAdd.c
clean:
```

[2]就像 C 语言的宏一样

```
16      -rm $(OBJS) $(EXEC)
run:$(EXEC)     bsub $(OPT) ./$(EXEC)
```

静态模式语法可以以一个任务定义同时定义多个任务，例如当上面的 Makefile 要编译多个主核和从核源文件时，可以用内嵌函数和静态模式语法进一步缩写：

```
MASTER=master/
2 SLAVE=slave/
OBJDIR=obj/
4
EXEC=arrAdd
6 OBJ=master_arrAdd1.o master_arrAdd2.o slave_arrAdd1.o slave_arrAdd2.o
OBJS=$(addprefix $(OBJDIR), $(OBJ))
8
OPT=-I -b
10 OPT+=-q q_sw_expr
OPT+=-n 1
12 OPT+=-cgsp 64

14 $(EXEC):$(OBJS)
    sw5cc -hybrid $^ -o $@
16 $(OUTDIR)%.o:$(MASTER)%.c
    sw5cc -host -c $< -o $@
18 $(OUTDIR)%.o:$(SLAVE)%.c
    sw5cc -slave -c $< -o $@
20 clean:
    -rm $(OBJS) $(OUTPUT)
22 run:$(EXEC)
    bsub $(OPT) ./$@
```

在此 Makefile 目录下运行 Linux make 时，行 16~19 的两个静态模式语法会使得 make 从 MASTER 变量和 SLAVE 变量所指目录下找到所有.c 文件编译后放入 OUTDIR 所指目录中，即将多个任务用一个任务完成定义。在大型项目中善用此方法可以大大减少 Makefile 文件的长度，使项目更易于维护。

3.3.2 Darknet 的 Makefile

官网下载的 Darknet 源码中的 Makefile 文件如附录B所示，本节将对这个 Makefile 文件及其所定义的编译过程进行解析。

3.3.2.1 变量定义

Makefile 文件 1~30 行是变量的定义，其各部分作用如下：

- 行 1~5:整体的编译设定。指定编译过程中是否包含 GPU、CUDNN、OpenCV、OpenMP 和 DEBUG 模式。这些选项会在第 32~59 行被一系列 ifeq 调用，根据其值改变编译选项；
- 行 7~12：NVCC 的编译选项。这个变量在第 92 行编译.cu 文件时调用。NVCC 是 Nvidia C Compiler，编译 CUDA 的.cu 文件的编译器，这里的这些编译选项表示编译的目标设备的算力[3]；
- 行 16~20：SLIB 和 ALIB 是生成的链接库的.so 和.a 文件名称、EXEC 是生成的可执行文件的名称、OBJDIR 是中间文件的存储路径，它们都在后面的编译过程中被调用。VPATH 是 Makefile 的一个特殊变量，VPATH 中指代的目录下的文件在后面的过程中不需要再输完整路径[4]，多个目录以冒号分隔。VPATH 实际发挥作用的地方是行 85~92 三个静态模式语法中没有路径的.c、.cpp、.cu 源文件；
- 行 22~30：CC、CPP、NVCC、AR 都是 GCC 里的编译器，ARFLAGS、OPTS、LDFLAGS、COMMON、CFLAGS 都是这些编译器的编译选项，这些变量也都在后面的编译过程中被调用。

3.3.2.2 编译选项的设置

Makefile 文件 32~73 行是编译选项的设置，其主要作用有二：

- 用 ifeq 根据行 1~5 定义的整体编译设定修改行 22~30 定义的编译选项；
- 确定输出文件的文件位置，以供 make 检查文件更新使用（行 68 和 69 的内嵌函数 addprefix 是加前缀，行 70 的 $(wildcard src/*.h) 是返回 src 目录下的所有.h 文件路径）。这些文件位置都会在后面的编译过程被调用。

3.3.2.3 编译任务

从行 76 开始的所有部分均是和编译相关的任务定义。其各部分作用如下：

- 行 72：默认 make 任务，生成 obj、backup、results 三个目录（行 94~99 的任务），生成静态链接库和动态链接库（行 79~83 ALIB 和 SLIB 变量值对应的任

[3]对于不同算力的 Nvidia 卡 NVCC 有不同的优化策略

[4]就像 Windows 和 Linux 里面的环境变量

务），生成可执行文件（EXEC 变量值对应的任务）；

- 行 76~77：生成可执行文件。该任务依赖于 obj 目录下的一系列.o 文件（变量 EXECOBJ）和静态链接库（变量 ALIB）；
- 行 79~83：静态模式语法，用 obj 目录下的一系列.o 文件（变量 OBJS）生成静态链接库和动态链接库；
- 行 85~92：静态模式语法，用源代码生成 obj 目录下一系列.o 文件；
- 行 94~104：编译相关的其他任务。.PHONY: clean 使 clean 任务必被执行。

3.3.3 开始移植 Darknet

做了这么多前期学习，终于可以正式开始移植 Darknet 了。但是有了前面那些分析感觉移植 Darknet 的这一节没啥好讲的了。移植主要是改 Makefile 文件，这里简单记录一下我改了哪吧。读者请自己下载 Darknet 在神威上编译，按照编译错误信息一步步进行修改，下面这个修改过程也是对着错误信息一步步出来的，仅供参考。

3.3.3.1 Makefile 文件修改

1. 编译器首先要修改的肯定是编译器。要把原来的 C 语言编译器（变量 CC）和 C++ 编译器（变量 CPP）的值改成神威里的编译器 `sw5cc` 和 `sw5CC` ；

2. 编译选项其次是要改编译选项（变量 CFLAGS）。原来的一些编译选项在 `sw5cc` 里面是不支持的，然后因为目前只是移植所以编译选项设置为`-host` 全部在主核编译运行；

3. 连接选项在连接任务中，sw5cc 连接时要加上`-hybrid` 选项；

4. 运行方式原有的 Makefile 文件里面没有运行任务，要加一个运行任务（如1.4节）；

5. 其他修改

- 生成动态链接库有点问题，目前无法解决，故先删去生成动态链接库的任务；
- 为了之后方便优化和维护继续往 Makefile 里加东西时不会和原来不支持的编译模式弄混，最好把所有神威系统不支持的编译模式删掉，包括 CUDA、OpenMP 和 OpenCV 编译，只保留没有任何依赖的普通模式和 debug 编译模式。

移植后 Darknet 的 Makefile 文件如附录C所示。

3.3.3.2 源代码文件修改

1. include/darknet.h 这个文件在编译时报类型重定义错误，原因是里面的 network 和 layer 类型嵌套方式的问题，改成能正常编译的 struct 嵌套就行了。

经过以上修改,即可在 Darknet 根目录下运行“make”生成可执行文件、“make run”提交运行了。运行“make run”若任务队列返回了“usage: ./darknet <function>”即说明移植成功。

3.4 本章总结

本章的主要脉络如下：

1. 解析 Darknet 源码文件的结构；
2. 进一步讲解了 Makefile 文件的语法，为解析 Darknet 的 Makefile 做铺垫；
3. 解析 Darknet 的 Makefile，明确移植 Darknet 需要修改的地方；
4. 移植 Darknet，解决一些细节问题。

第 4 章　Darknet 的并行优化

Darknet 的移植完成后就可以开始进行并行优化了。Darknet 的并行优化主要是将 Darknet 的核心代码（主要是神经网络中的矩阵运算）使用并行方式重新书写并完成编译。

待续

附录 A　第一个程序

A.1　主核

```
#include <stdlib.h>
#include <stdio.h>
#include <athread.h>
#include <sys/time.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>

extern SLAVE_FUN(func)();

#define X 64
#define Y 2048

int A[X][Y], B[X][Y], C[X][Y],CC[X][Y];

void init()
{
    int i, j;
    for (i = 0; i < X; i++)
        for (j = 0; j < Y; j++)
        {
            A[i][j] = i + j;
            B[i][j] = i + j + 1;
            C[i][j] = 0;
            CC[i][j]=0;
        }
    printf("Init finished\n");
}

int main(void)
{
    int i, j;
    struct timeval start, end;
    init();
    double time_use;
    printf("No boost proceed:\n");
    gettimeofday(&start, NULL);
    for (i = 0; i < X; i++)
        for (j = 0; j < Y; j++)
            C[i][j] = A[i][j] + B[i][j];
    gettimeofday(&end, NULL);
    time_use = 1000000 * (end.tv_sec - start.tv_sec) + end.tv_usec - start.tv_usec;
```

```
    printf("Time usage is %lf us\n", time_use);
    printf("(C[32][0], C[63][999]) = (%d, %d)\n", C[32][0], C[63][999]);    init();
    athread_init();
    printf("Boosted proceed:\n");
    gettimeofday(&start, NULL);
    athread_spawn(func, 0);
    athread_join();
    gettimeofday(&end, NULL);
    time_use = 1000000 * (end.tv_sec - start.tv_sec) + end.tv_usec - start.tv_usec;
    printf("Time usage is %lf us\n", time_use);
    printf("(C[32][0], C[63][999]) = (%d, %d)\n", C[32][0], C[63][999]);
    athread_halt();
    return 0;
}
```

A.2 从核

```
#include "slave.h"
#define X 64
#define Y 2048

__thread_local int my_id;
__thread_local volatile unsigned long get_reply, put_reply;
__thread_local int A_slave[Y], B_slave[Y], C_slave[Y];
extern int A[X][Y], B[X][Y], C[X][Y];

void func()
{
    int i;
    my_id = athread_get_id(-1);
    get_reply = 0;
    put_reply = 0;
    athread_get(PE_MODE, &A[my_id][0], A_slave, Y * 4, &get_reply, 0, 0, 0);
    athread_get(PE_MODE, &B[my_id][0], B_slave, Y * 4, &get_reply, 0, 0, 0);
    while (get_reply != 2)
        ;
    for (i = 0; i < Y; i++)
    {
        C_slave[i] = A_slave[i] + B_slave[i];
    }
    athread_put(PE_MODE, C_slave, &C[my_id][0], Y * 4, &put_reply, 0, 0);
    while (put_reply != 1)
        ;
}
```

A.3 经过 SIMD 优化后的从核

```
#include "simd.h"
#include "slave.h"
#define X 64
#define Y 2048

__thread_local int my_id;
__thread_local volatile unsigned long get_reply, put_reply;
__thread_local int A_slave[Y], B_slave[Y], C_slave[Y];
extern int A[X][Y], B[X][Y], C[X][Y];

void func()
{
    int i;
    intv8 v,va;
    my_id = athread_get_id(-1);
    get_reply = 0;
    put_reply = 0;
    athread_get(PE_MODE, &A[my_id][0], A_slave, Y * 4, &get_reply, 0, 0, 0);
    athread_get(PE_MODE, &B[my_id][0], B_slave, Y * 4, &get_reply, 0, 0, 0);
    while (get_reply != 2)
        ;
    for (i = 0; i < Y; i+=8)
    {
        simd_load(va,&(A_slave[i]));
        simd_load(v,&(B_slave[i]));
        v=v+va;
        simd_store(v,&(C_slave[i]));
    }
    athread_put(PE_MODE, C_slave, &C[my_id][0], Y * 4, &put_reply, 0, 0);
    while (put_reply != 1)
        ;
}
```

附录 B Darknet 的 Makefile

```
GPU=0
CUDNN=0
OPENCV=0
OPENMP=0
DEBUG=0

ARCH= -gencode arch=compute_30,code=sm_30 \
      -gencode arch=compute_35,code=sm_35 \
      -gencode arch=compute_50,code=[sm_50,compute_50] \
      -gencode arch=compute_52,code=[sm_52,compute_52]
#     -gencode arch=compute_20,code=[sm_20,sm_21] \ This one is deprecated?

# This is what I use, uncomment if you know your arch and want to specify
# ARCH= -gencode arch=compute_52,code=compute_52

VPATH=./src/:./examples
SLIB=libdarknet.so
ALIB=libdarknet.a
EXEC=darknet
OBJDIR=./obj/

CC=gcc
CPP=g++
NVCC=nvcc
AR=ar
ARFLAGS=rcs
OPTS=-Ofast
LDFLAGS= -lm -pthread
COMMON= -Iinclude/ -Isrc/
CFLAGS=-Wall -Wno-unused-result -Wno-unknown-pragmas -Wfatal-errors -fPIC

ifeq ($(OPENMP), 1)
CFLAGS+= -fopenmp
endif

ifeq ($(DEBUG), 1)
OPTS=-O0 -g
endif

CFLAGS+=$(OPTS)

ifeq ($(OPENCV), 1)
COMMON+= -DOPENCV
CFLAGS+= -DOPENCV
```

```
LDFLAGS+= `pkg-config --libs opencv` -lstdc++
COMMON+= `pkg-config --cflags opencv`endif

ifeq ($(GPU), 1)
COMMON+= -DGPU -I/usr/local/cuda/include/
CFLAGS+= -DGPU
LDFLAGS+= -L/usr/local/cuda/lib64 -lcuda -lcudart -lcublas -lcurand
endif

ifeq ($(CUDNN), 1)
COMMON+= -DCUDNN
CFLAGS+= -DCUDNN
LDFLAGS+= -lcudnn
endif

OBJ=gemm.o utils.o cuda.o deconvolutional_layer.o convolutional_layer.o list.o image.o
    activations.o im2col.o col2im.o blas.o crop_layer.o dropout_layer.o maxpool_layer.o
    softmax_layer.o data.o matrix.o network.o connected_layer.o cost_layer.o parser.o
    option_list.o detection_layer.o route_layer.o upsample_layer.o box.o
    normalization_layer.o avgpool_layer.o layer.o local_layer.o shortcut_layer.o
    logistic_layer.o activation_layer.o rnn_layer.o gru_layer.o crnn_layer.o demo.o
    batchnorm_layer.o region_layer.o reorg_layer.o tree.o  lstm_layer.o l2norm_layer.o
    yolo_layer.o iseg_layer.o image_opencv.o
EXECOBJA=captcha.o lsd.o super.o art.o tag.o cifar.o go.o rnn.o segmenter.o regressor.o
    classifier.o coco.o yolo.o detector.o nightmare.o instance-segmenter.o darknet.o
ifeq ($(GPU), 1)
LDFLAGS+= -lstdc++
OBJ+=convolutional_kernels.o deconvolutional_kernels.o activation_kernels.o im2col_kernels.o
     col2im_kernels.o blas_kernels.o crop_layer_kernels.o dropout_layer_kernels.o
    maxpool_layer_kernels.o avgpool_layer_kernels.o
endif

EXECOBJ = $(addprefix $(OBJDIR), $(EXECOBJA))
OBJS = $(addprefix $(OBJDIR), $(OBJ))
DEPS = $(wildcard src/*.h) Makefile include/darknet.h

all: obj backup results $(SLIB) $(ALIB) $(EXEC)
#all: obj  results $(SLIB) $(ALIB) $(EXEC)


$(EXEC): $(EXECOBJ) $(ALIB)
    $(CC) $(COMMON) $(CFLAGS) $^ -o $@ $(LDFLAGS) $(ALIB)

$(ALIB): $(OBJS)
    $(AR) $(ARFLAGS) $@ $^

$(SLIB): $(OBJS)
    $(CC) $(CFLAGS) -shared $^ -o $@ $(LDFLAGS)
```

```
84 $(OBJDIR)%.o: %.cpp $(DEPS)    $(CPP) $(COMMON) $(CFLAGS) -c $< -o $@

86 $(OBJDIR)%.o: %.c $(DEPS)
       $(CC) $(COMMON) $(CFLAGS) -c $< -o $@
88
   $(OBJDIR)%.o: %.cu $(DEPS)
90     $(NVCC) $(ARCH) $(COMMON) --compiler-options "$(CFLAGS)" -c $< -o $@

92 obj:
       mkdir -p obj
94 backup:
       mkdir -p backup
96 results:
       mkdir -p results
98
   .PHONY: clean
100
   clean:
102    rm -rf $(OBJS) $(SLIB) $(ALIB) $(EXEC) $(EXECOBJ) $(OBJDIR)/*
```

附录 C　移植后 Darknet 的 Makefile 文件

```
DEBUG=0

VPATH=./src/:./examples
ALIB=libdarknet.a
EXEC=darknet
OBJDIR=./obj/

CC=sw5cc
CPP=sw5CC
AR=ar
ARFLAGS=rcs
OPTS= #-Ofast
LDFLAGS= -lm -lpthread
COMMON= -Iinclude/ -Isrc/ -fPIC -Wall
CFLAGS= -host
HFLAGS= -hybrid

ifeq ($(DEBUG), 1)
OPTS=-O0 -g
endif

CC+=$(COMMON)
CPP+=$(COMMON)
CFLAGS+=$(OPTS)
HFLAGS+=$(OPTS)

OBJ=gemm.o utils.o cuda.o deconvolutional_layer.o convolutional_layer.o list.o image.o
    activations.o im2col.o col2im.o blas.o crop_layer.o dropout_layer.o maxpool_layer.o
    softmax_layer.o data.o matrix.o network.o connected_layer.o cost_layer.o parser.o
    option_list.o detection_layer.o route_layer.o upsample_layer.o box.o
    normalization_layer.o avgpool_layer.o layer.o local_layer.o shortcut_layer.o
    logistic_layer.o activation_layer.o rnn_layer.o gru_layer.o crnn_layer.o demo.o
    batchnorm_layer.o region_layer.o reorg_layer.o tree.o  lstm_layer.o l2norm_layer.o
    yolo_layer.o iseg_layer.o image_opencv.o
EXECOBJA=captcha.o lsd.o super.o art.o tag.o cifar.o go.o rnn.o segmenter.o regressor.o
    classifier.o coco.o yolo.o detector.o nightmare.o instance-segmenter.o darknet.o

EXECOBJ = $(addprefix $(OBJDIR), $(EXECOBJA))
OBJS = $(addprefix $(OBJDIR), $(OBJ))
DEPS = $(wildcard src/*.h) Makefile include/darknet.h

all: obj backup results $(ALIB) $(EXEC)
#all: obj  results $(ALIB) $(EXEC)

```

```
$(EXEC): $(EXECOBJ) $(ALIB)  $(CC) $(HFLAGS) $^ -o $@ $(LDFLAGS) $(ALIB)

$(ALIB): $(OBJS)
  $(AR) $(ARFLAGS) $@ $^

$(OBJDIR)%.o: %.cpp $(DEPS)
  $(CPP) $(CFLAGS) -c $< -o $@ $(LDFLAGS)

$(OBJDIR)%.o: %.c $(DEPS)
  $(CC) $(CFLAGS) -c $< -o $@ $(LDFLAGS)

obj:
  mkdir -p obj
backup:
  mkdir -p backup
results:
  mkdir -p results
run: $(EXEC)
  bsub -I -b -q q_sw_expr -n 1 -cgsp 64 ./darknet go

.PHONY: clean

clean:
  rm -rf $(OBJS) $(ALIB) $(EXEC) $(EXECOBJ) $(OBJDIR)/*
```

参考文献

[1] 为何中国超算偏爱异构计算[J/OL]. 知乎专栏[2019-07-09]. https://zhuanlan.zhihu.com/p/20908218.

[2] REDMON J. Darknet: Open source neural networks in c[EB/OL]. 2013–2016. http://pjreddie.com/darknet/.

[3] REDMON J, FARHADI A. Yolov3: An incremental improvement[J]. arXiv, 2018.

[4] Darknet 概述 - 搬砖笔记 - CSDN 博客[EB/OL]. [2019-07-14]. https://blog.csdn.net/u010122972/article/details/83541978.

图形列表

表格列表