CS205 · C/C++ Programming

Project3 Report: SGEMM Optimization

赖海斌
12211612

摘要：在本次 Project 里我们要优化 SGEMM。我们先进行了一些理论探索，然后进行了基准测试。我们对 OpenBLAS 和 Intel MKL 进行了测试和底层分析，对其代码结构、指令原理、通信模型进行了研究。这里边无论是不同硬件上的针对优化，还是软件上的分治操作，都是一个复杂的系统工程。而这工程中最重要的一环，便是在“计算-访存”的钢丝绳上保持平衡。我们在这短短的 3 周内，简单了解工程师们对此的各种思想与工作。

无论是硬件上算力的堆叠，还是软件上循环的优化，GEMM 加速带有着计算机独特的暴力美学。让我们跟着这方面的专家，我最喜欢的导演之一，Quentin Tarantino，来导演属于自己的计算程序。

关键词：并行计算；BLAS；向量化；异构计算加速；平衡点理论；SIMD

目录 Content

摘要 ..... 错误！未定义书签。

Part 1: 需求分析 — **Chapter 1: RESERVOIR DOGS** 3

Part 2: 实验平台简介 — **Chapter 2: FOUR ROOMS** 3

- 2.1 CS205 Server — **The Misbehavers** 3
- 2.2 启明超算集群 38 队列 — **The Man from Hollywood** 5
- 2.3 平台软件 — **The Missing Ingredient** 6

Part 3: 实验标准与实验原理 — **Chapter 3: PULP FICTION** 6

- 3.1 时间测量 — **Butch's Golden Watch** 6
- 3.2 GEMM 乘法统一 api 与 矩阵 Structure — **Vincent And Marsellus' Wife** 7
- 3.3 计算-通信平衡公式 — **Pumpkin And Honey Bunny** 10

Part 4: SGEMM 基准测试 — **Chapter 4: THE HATEFUL EIGHT** 11

Part 5: 优化方案 — **Chapter 5: KILL BILL** 24

- 5.1 击败 16 x 16 — **Vernita Green** 24
- 5.2 击败 128 x 128 — **O-ren Ishi** 25
- 5.3 击败 1K x 1K — **Budd** 25
- 5.4 击败 8K x 8K — **Elle** 26
- 5.5 击败 64K x 64K — **Bill** 28

Part 6: 性能测试 — **Chapter 6: DJANGO UNCHAINED** 30

Part 7: 总结 — **Chapter 7: INGLOURIOUS BASTERDS** 34

Reference 37

Part 1: 需求、目标 **Chapter 1: RESERVOIR DOGS**

油气湍流模拟、地震波传播演算、分子动力学模拟[1]等科学模拟计算中，通用矩阵乘法 GEMM 占据了主要的计算过程[6]。图形学、深度学习、大语言模型训练也和矩阵乘法高度相关。本次 Project 的目标就是优化单精度浮点数通用矩阵乘法 SGEMM。

Project3 开始的那一周正好去上海打 ASC 超算比赛了，在比赛里和不同学校的同学交流优化的方案。结果听完他们的优化，我心里只冒出一句话：哎！要是上学期学了 C/C++，甚至要是早 3 个周做这次 Project，可能这次比赛，就可以有更多的优化了吧！但是比赛已经过去，明年在场上的可能就是学弟学妹了。所以看到题目的时候，我心里想，一定要**做一个比 Intel MKL、OpenBLAS 还要优秀的 GEMM 出来**，以后学弟学妹们就能用上这个 api，让南科大走得更远。

带着这个想法，我针对 Project 订立了下面几个目标：

- 1. 熟悉我们的计算硬件；
- 2. 学习一部分理论知识，应对可能的优化；
- 2. 测量 Intel MKL 和 OpenBLAS 在 SGEMM 上的性能；
- 3. 针对不同的矩阵规模，采取不同的优化策略；
- 4. 测试我们的乘法性能，并试图把它用在科学计算软件中，查看我们程序的加速时间。

Part 2: 平台、决斗场 **Chapter 2: FOUR ROOMS**

本节介绍我们跟 OpenBLAS，Intel MKL 比武的场地，CS205 Server 和启明 38 队列。

2.1 CS205 Server **The Misbehavers**

我们的实验与优化将主要在 CS205 Server 上进行。Server 有两个 CPU 共 48 核，使用 VMware 开启虚拟化，使得 NUMA node 数量为 1。配置详见表 2.1。同时，该 Server 也支持 sse4,avx2,avx512 拓展指令集。

Model name	Intel(R) Xeon(R) Gold 6240 CPU @ 2.60GHz
CPU Architecture	x86_64
CPU family	6
Thread(s) per core	1
CPU(s)	48
Core(s) per socket	24
Memory	128GiB
Caches	100.5MiB
Hypervisor vendor	VMware
NUMA node	1

表 2.1 硬件 CPU 环境

Server 装备了 4 张 RTX 2080Ti Turing 架构 GPU。详见表 2.2[12]。

[1] OpenCAEPoro, ParaSeis, Gromacs

GPU name	NVIDIA GeForce RTX 2080 Ti
GPU amount	4
CUDA Version	12.2
Driver Version	535.171.04
Memory	11264MiB
GPU Base Clock MHz	1350
SM amount per GPU	68

表 2.2 硬件 GPU 环境

Turing 架构是一款比较经典的 GPU 架构。出于学习的目的，我在这里简单介绍一下 Turing 架构，比如下面这个 72 个 SM 的 GPU[2]:

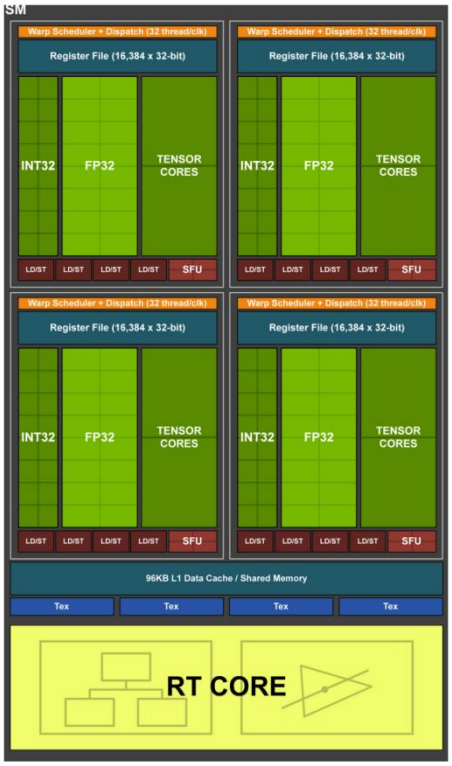
在这里，有这么几个重要的设计理念：

- PCI Express 3.0 Host Interface
- GigaThread Engine
- GPC (Graphics Processing Cluster)
- Raster Engine
- SM (Streaming Multiprocessor)
- L2 Cache

[2] 图片来源：文献[12]，NVIDIA Turing Architecture White Paper

PCIe3 是 GPU 连接 CPU 的沟通线，它会通过链路点对点地与 CPU 进行全双工通信[3]。GigaThread Engine 类似于 GPU 的 OS，但在软件层面再做一个 OS，对于游戏渲染还是速度太慢了，因此这个 GTE 是直接用硬件写出的调度系统。如果 GPU 是个课题组，你可以认为 GTE 是课题组的**大老板**，它会解析 PCIe3 发过来的各种基金、文章，然后调度**小老板**图形处理簇 GPC 开始发文章，创建线程。小老板底下有一个**延毕老博士**光栅化引擎 Raster Engine，这个课题难而且老板关注的不多，所以延毕了。小老板带着一群**硕士生** TPC 纹理处理簇，每个硕士生带着两个做创新实践的**本科生** SM（Stream Multiprocessor，流多处理器）[4]。

本科生 SM 就是 GPU 的帕鲁，他们负责执行具体的运算，SM 里有很多个 CUDA Core，比如整数运算 Core，浮点数运算 Core 张量 tensor Core，Turing 架构特有的光线追踪 RT Core（本次 Project 可能很难用到它），特别函数 SFU Core[5]。他们每个都有自己的小 ALU，SM 会通过 4 个 Warp Scheduler 来分别调度使用他们，Dispatch 会给 Core 们分发具体的指令与数据。



Warp 是一个很可怕的驱动者，它就像 GPA，会尽可能地压榨学生的 Core。比如它收到 1 个命令要开启 32 个线程计算浮点运算，那它就会要求 16 个 FP Core 一起计算 2 次。这样的计算模型被叫做 SIMT，单指令多线程。如果在计算的过程中出现了分支问题，Warp 会继续从 Instruction Cache 中取新的指令，让这群 Core 永远都不能躺在宿舍里睡觉。

1 个 Warp Scheduler 内的核心计算出的结果会存储在 register File 中，而 SM 内要想 Core 执行的线程实现相互通信，就要借助 SM 的 L1 Data Cache，这样，一个 SM 内通信就建立起来了。而如果本科生想跟组内其他人交流，就要通过组内大群/飞书：L2 Cache 来进行通信。

而如果一个 GPU 课题组想跟另一个课题组合作，传统的方法是 GPU 通过 PCIe3 将数据发回到 CPU 的 DRAM Main Memory 上，另一个 GPU 再去拿取数据。更先进的方法是 **NVLink**，它通过高速通道直接与别的课题组建立合作通讯。

2.2 启明超算集群 38 队列 **The Man from Hollywood**

在本次 Project 中我将自己的 GEMM 算法应用到实际的软件中，并使用南科大科学与工程计算中心启明超算集群 38 队列进行标准测量。

截至 2024 年 2 月，南科大“启明”超算集群拥有 299 个计算节点，10072 个计算核心，每年可向用户提供超过 8823 万核时计算资源，支撑着 13 个院系 116 个课题组的科研计算需求。而其中属于“启明 2.0”的 38 队列是启明集群内使用率最高的队列，课题组用户用量大。针对该队列的算法优化可以帮助减少用户作业的计算时间，提高队列计算效率[25]。

表 10. 2023 年，“启明”集群各 CPU 队列使用情况统计（时间单位：小时）

队列名	完成率	使用率	作业平均等待时间	千核时等待时间	用户数	课题组数	院系数
v3-64	99.57%	13.81%	0.03	0.06	41	25	11
v3-128	88.34%	0.15%	0.01	0.42	9	9	4
v3-6t	93.95%	8.56%	0.01	0.01	15	12	9
38	98.33%	16.75%	0.43	0.88	39	25	10
63	96.84%	10.29%	0.02	0.72	11	9	5

[3] 可以学习这两篇文章介绍 PCIe3：https://zhuanlan.zhihu.com/p/639722704 https://zhuanlan.zhihu.com/p/454282470
[4] GPU 的小科普 https://zhuanlan.zhihu.com/p/406572255
[5] GPU 架构介绍 https://zhuanlan.zhihu.com/p/423129242

38 队列共有 36 个刀片节点。每个节点 CPU 型号为 Intel Xeon(R) Gold 6338 CPU，单节点内有 2 个 NUMA node，64 核，50MB Cache 及 512GB Memory，CPU 支持 sse4,avx2,avx512 拓展指令集。节点间使用 200GB HDR Infiniband Switch 进行通信，作业采用 IBM Spectrum LSF 调度系统进行管理[24]。

38	CPU: 2 * Intel Xeon Gold 6338 (2.0GHz, 32 核)； 内存：512 GB；	CPU：2 * Intel 第三代志强处理器 (单颗 CPU：2.0GHz，32 核)； 内存：512 GB；	36 节点	64	2304

表 2.3 38 队列 CPU

2.3 平台软件 **The Missing Ingredient**

在 CS205 Server 上老师提供了 gcc, nvcc 编译器，make 和 cmake 工具以及 nsight 性能检测器。我在此基础上又编译了 OpenBLAS 数学库, OpenMPI 以及下载了 Intel® oneAPI Base Toolkit，内含 Intel MKL 数学库和 Intel MPI，Intel VTune Profiler。

在启明上集群使用 module 管理工具对软件进行管理。上面提供了 gcc，make 和 cmake，在此基础上我也安装了 Intel® oneAPI Base Toolkit。下表为平台数学库、编译器与操作系统具体版本。

Server/Version	Intel® oneAPI Base Toolkit	OpenBLAS
CS205 Server	2024.1.0.596	0.3.26
Qiming 2.0	2024.1.0.596	0.3.26

Server/Version	nvcc	OS	GCC	make
CS205 Server	cuda12.4 compiler.34097967	Ubuntu 22.04.4	11.4.0	4.3
Qiming 2.0	-	Rocky 4.18.0	12.2.0[6]	4.2.1

Part 3: 标准、规则、赛场准备 **Chapter 3: PULP FICTION**

要击败我们的对手，我们在选定好比武的场地后，要定下比武的规则，和提升武力的法则，另外，在赛场准备时，仔细观察我们的对手。

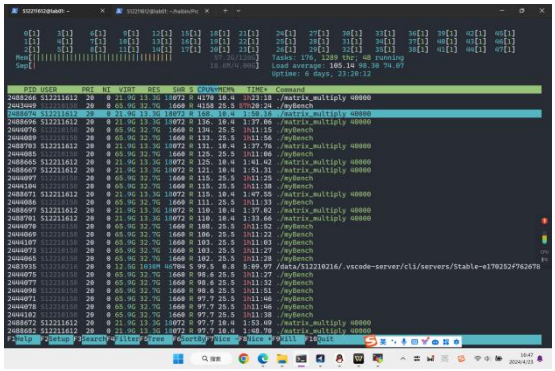
3.1 时间测量 **Butch's Golden Watch**

我们将测量我们的乘法计算运行时间。为此我们使用 High-Performance Linpack Benchmark（HPL）中计算使用时间的函数 HPL_timer_walltime。其使用<sys/time.h>和<sys/resource.h>获取系统时间。

```
double HPL_timer_walltime()
{
   struct timeval              tp;
   static long                 start=0, startu;

   if( !start )
   {
      (void) gettimeofday( &tp, 0 );
      start  = tp.tv_sec;
      startu = tp.tv_usec;
      return( 0 );
   }
   (void) gettimeofday( &tp, 0 );

   return( (double)( tp.tv_sec - start ) +
           ( (double)( tp.tv_usec-startu ) / 1000000.0 ) );
}
```

[6] 启明上也有比较接近 Server gcc 版本的 11.2.0，但是我选择了更高的版本。

计算机在计算矩阵乘法时，会受到许多相互独立的随机因素的影响，比如此时计算机的温度，瞬时时钟频率，当前要计算的浮点数数值等等。相对整体计算而言每个因素变动较小，所产生的影响也较小。我们可以近似地认为在计算系统未受到大干扰（如 CPU 核已全部被占用、程序被人为杀死）情况下，根据**中心极限定理**，程序执行所用时间服从**正态分布**。因而对于每组数据我们进行多次测试，排除时间在 2σ的情况，**取 5 次数据的平均值作为当前输入情况下的测量结果**。



运行程序的时候，经常发现自己的程序会跟别的程序进行竞争。为了较为良好地比较数据，会统一对运行时的环境进行监测。

3.3 借鉴 GEMM 乘法 api 与 矩阵 Structure 设计 ***Vincent And Marsellus' Wife***

我们的乘法优化算是 SGEMM 优化的一部分。它是 BLAS(Basic Linear Algebra Subprograms，基础线性代数程序集)中 Level3 的一个重要子程序。要想更好地达到我们 Project 的目标，我认为我就应该看看它到底是什么，至少**应该看看我们的对手 OpenBLAS 长什么样子，然后再定下标准 api 和优化的方向**。

在 BLAS 库中，线性代数的计算被分为了 3 个 Level,level1 支持向量间的操作，如交换，点乘；level2 支持向量与矩阵之间的操作，level3 支持矩阵与矩阵之间的操作，GEMM 就是里边的重要函数，用的最多的就是 SGEMM 和 DGEMM[26]。GEMM 表示下面这样一种标准运算：

$$C \leftarrow \alpha AB + \beta C \ldots\ldots (3.2.1)$$

从文章[26]查找 GEMM 的输入，一共有这么几个输入参数，下面进行一一解释。

```
        options                          dim      scalar matrix  matrix  scalar matrix
xGEMM (            TRANSA, TRANSB,      M, N, K, ALPHA, A, LDA, B, LDB, BETA,  C, LDC )
```

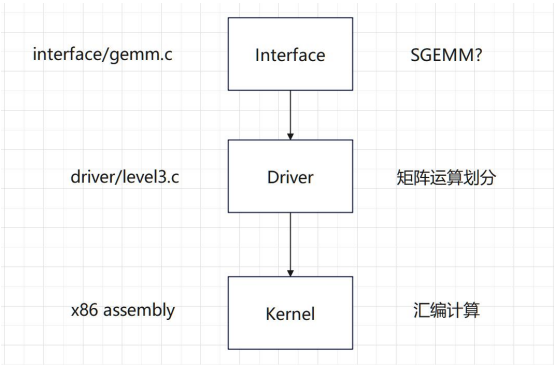
Parameter	Explain
CBLAS_ORDER	是行主序还是列主序完成
TRANSA:	矩阵 A 是否转置
TRANSB:	矩阵 B 是否转置
M	A 的行，C 的行
N	A 的列，B 的行
K	B 的列，C 的列[7]
ALPHA	公式 3.2.1 中的 alpha，单纯的乘法取 1
A	矩阵 A
LDA	矩阵 A 的 leading dimension
B	矩阵 B
LDB	矩阵 B 的 leading dimension
BETA	公式 3.2.1 中的 beta，单纯的乘法取 0
C	矩阵 C
LDC	矩阵 C 的 leading dimension

[7] 注意这里的 MNK 都是 A，B 转置后的数 参考：https://blog.csdn.net/yutianzuijin/article/details/90411622

我们也在函数的实现中加入这些输入的参数，但是我们不会亲手实现里边的每一个参数所需的函数。他们有的不太会在 Server 中的测试用到，我不会也不可能在两周内给老师交一个手搓的 BLAS 出来，这是 Jack Dongarra 的课题，更别提超过现有的 MKL。但是我可以借鉴 OpenBLAS 的代码结构和输入参数，在更好地实现我们自己的矩阵乘法的同时做出一个接口，方便以后拓展，在优化程序上使用到。

我们去查看 OpenBLAS 的源代码[8]，会发现 OpenBLAS 是这样做矩阵乘法的：

BLAS 的底层分为 3 层，接口层，驱动层，以及核心层。



BLAS 在接收到 SGEMM()输入的参数后，首先会进入 Interface 层。在这里 BLAS 会对输入的参数进行解析，决定选择不同的算法和分支。如下图（1）是 gemm.c 中的一段判断矩阵是否转置的部分，随后会根据转置的情况来调用具体执行计算的 Driver。

```
  transA = *TRANSA;
  transB = *TRANSB;

  TOUPPER(transA);
  TOUPPER(transB);

  transa = -1;
  transb = -1;

  if (transA == 'N') transa = 0;
  if (transA == 'T') transa = 1;
#ifndef COMPLEX
  if (transA == 'R') transa = 0;
  if (transA == 'C') transa = 1;
#else
  if (transA == 'R') transa = 2;
  if (transA == 'C') transa = 3;
#endif
```

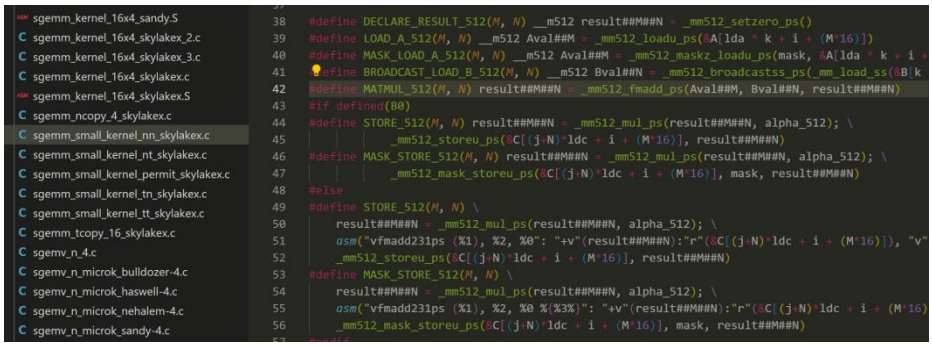
（1）

```
    /* First, we have to move data A to L2 cache */
    min_i = m_to - m_from;
    l1stride = 1;

    if (min_i >= GEMM_P * 2) {
min_i = GEMM_P;
    } else {
if (min_i > GEMM_P) {
  min_i = ((min_i / 2 + GEMM_UNROLL_M - 1)/GEMM_UNROLL_M) * GEMM_UNROLL_M;
} else {
  l1stride = 0;
}
    }
```

（2）

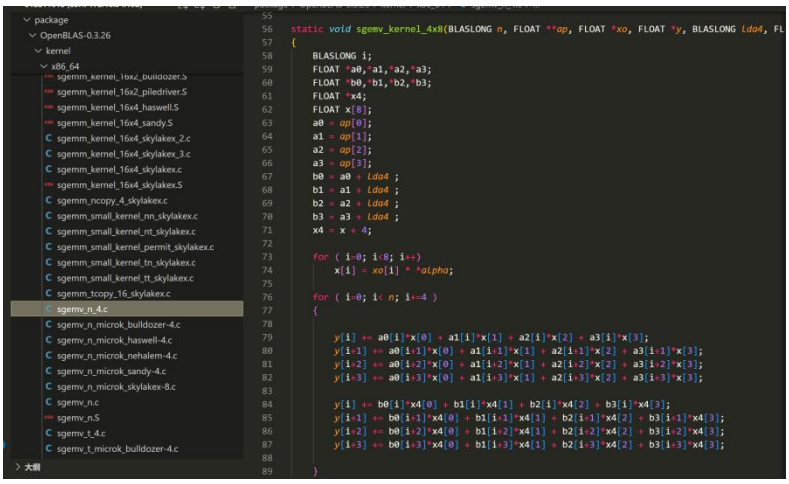
在 Driver 层数据会被预处理和存储，任务会被分配到各个 CPU 上。例如如果有转置，数据会被提前处理；CPU 会接收不同的任务准备并行处理；缓存会被提前分配，如图（2）。在一切准备就绪后，BLAS 会进入 Kernel 层。

Kernel 层占运算的主要部分，为了达到最极致的优化，OpenBLAS 用了多种方法：1.针对小矩阵专门设立小型函数（图 3 左侧 small_kernel）；2.在 x86_64 平台使用 AVX512 指令集（图 3 右侧）；3.面对特殊的输入矩阵采用循环展开（图 4），根据 Project2 的效果，这样可以减少执行的汇编指令；4.不演了！直接使用汇编代码来进行运算（图 5）。



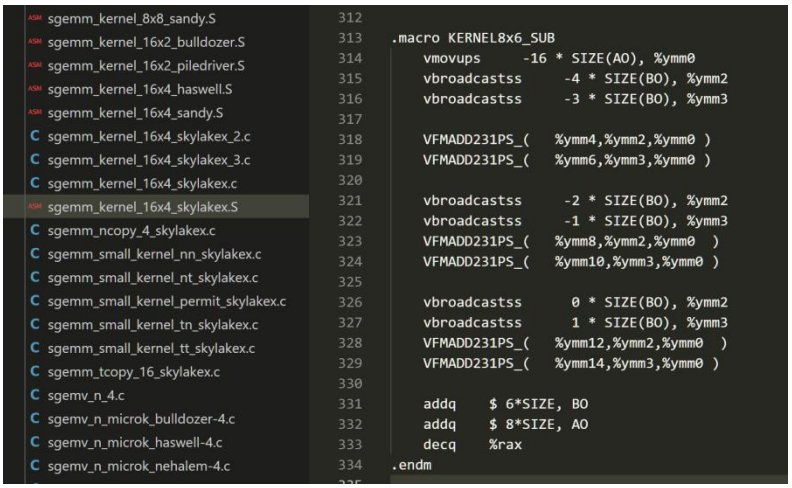
（3）

[8] 可以学习： https://blog.csdn.net/xqch1983/article/details/137585755



（4）

在固定了 MNK 后，对循环结构进行循环展开是一个很经典的 ijk 乘法优化方法。**在 Project2 的工作**中我们知道，ijk 循环在 O3 选项下被编译为使用寄存器存储 ijk 的汇编语言，而展开可以减少这一汇编语言的使用，从而降低循环时间。



（5）

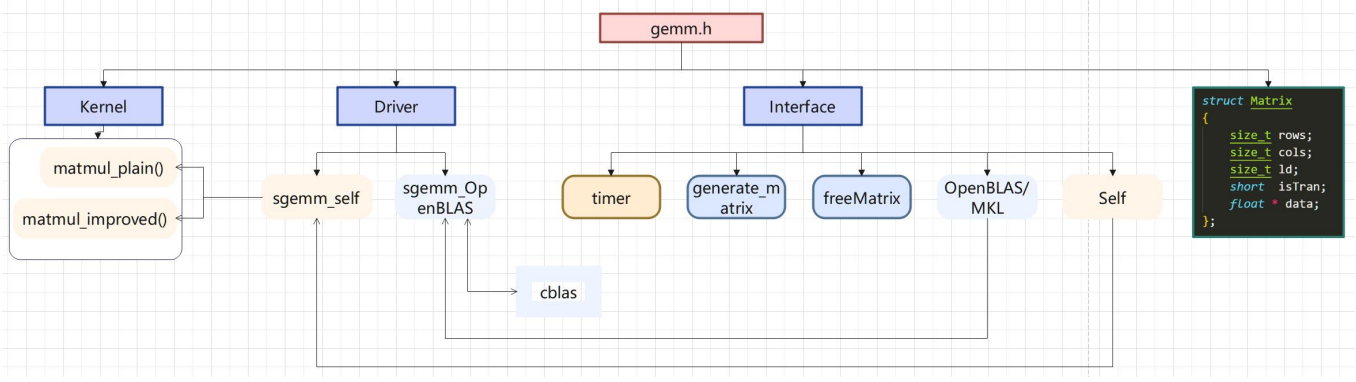
我们的对手是一个高手。它在汇编层面的优化，就像一个永远抽刀比你快的武士，要想打败他，我们要么也成为汇编高手，要么就在算法下功夫，要么就在多节点上下功夫。当然也可以用 GPU 开挂。

讲了那么多，我设计了自己的小 GEMM BLAS，俗称 GBLAS[9]：

```
Matrix
├─ Makefile
├─ driver
│  └─ init.c
├─ gemm.h
├─ interface
│  ├─ interface.c
│  └─ time.c
├─ kernel
│  └─ gemm.c
└─ main.c
```

项目使用 gemm.h 连接，CMake 编译，创建了 Matrix 结构进行封装。在使用时，用户仅需使用 Interface 层面的函数，就可以初始化矩阵，SGEMM 计算，计时，输出。（其实可以再加一个宏文件只暴露 Interface 层，但是我认为项目没有必要做特别复杂，何况只有我一个用户）

在用户选择了选定的数学库后，参数将传入 Driver 层。如果是 MKL 或者 OpenBLAS，Driver 将调动相应的数学库。如果是我实现的 SGEMM，Driver 层将判断矩阵的大小，分配核数，并发配给 Kernel 进行计算。Kernel 层内有多种算法，如单纯 AVX2，满足 Driver 的计算需求。



[9] GBLAS 的 Makefile 和 generate_matrix 方法由 GPT 辅助生成。

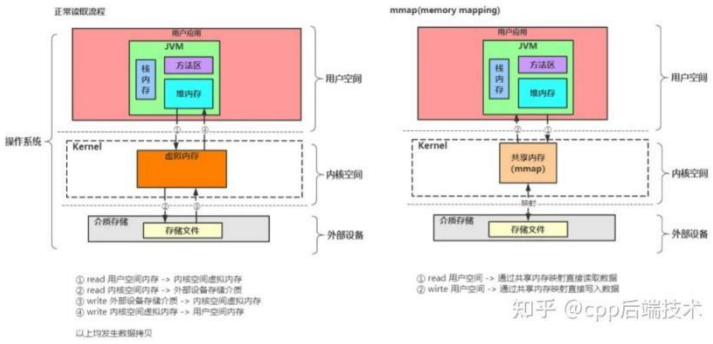
```c
#include <stdio.h>
#include "gemm.h"

int main() {
    struct Matrix *A, *B, *C;
    size_t rows1 = 5;
    size_t cols1 = 5;
    size_t cols2 = 5;

    A = generate_matrix(rows1, cols1);
    B = generate_matrix(cols1, cols2);
    C = generate_matrix(rows1, cols2);

    double time1 = HPL_timer_walltime();
    sgemm_OpenBLAS(A, B, C);
    double time2 = HPL_timer_walltime();

    GEMM_Print("Time elapsed: %f seconds.\n", time2 - time1);

    // print_matrix(C);

    freeMatrix(A);
    freeMatrix(B);
    freeMatrix(C);

    return 0;
}
```

```
> Project3 > HaibinImp > Matrix > M Makefile
CC = gcc
CFLAGS = -Wall -Wextra -Wpedantic
LDFLAGS = -lblas

all: main

main: main.o kernel/gemm.o driver/init.o interface/interface.o interface/time.o
    $(CC) $(CFLAGS) -o $@ $^ $(LDFLAGS)

main.o: main.c gemm.h
    $(CC) $(CFLAGS) -c -o $@ $<

kernel/gemm.o: kernel/gemm.c gemm.h
    $(CC) $(CFLAGS) -c -o $@ $<

driver/init.o: driver/init.c gemm.h
    $(CC) $(CFLAGS) -c -o $@ $<

interface/interface.o: interface/interface.c gemm.h
    $(CC) $(CFLAGS) -c -o $@ $<
interface/time.o: interface/time.c gemm.h
    $(CC) $(CFLAGS) -c -o $@ $<

clean:
    rm -f main main.o kernel/gemm.o driver/init.o interface/interface.o interface/time.o
```

可以看到，我们的 main 函数看起来就比较整洁，可以满足 Project。且通过封装我们可以给矩阵管理是否转置等操作。

在写生成矩阵时，我发现使用 malloc 函数会出现问题：在问题规模到 60K x 60K 左右时，malloc 会撞上已分配的内存，从而导致 segment fault。GPT 给了方法是使用 linux mmap 函数。这个函数会在在当前进程的虚拟地址空间中，寻找一段空闲的满足要求的连续的虚拟地址[10]，跳过已经分配的内存，建立起一个新的“虚拟”映射。不过这样的虚拟化跟 malloc 相比也带来一定的性能损失。不过这对我们加速矩阵乘法影响不大。

```c
struct Matrix* generate_matrix(size_t rows, size_t cols) {
    struct Matrix *mat = (struct Matrix *)malloc(sizeof(struct Matrix));
    mat->rows = rows;
    mat->cols = cols;
    mat->ld = cols;  // 假设不进行矩阵转置
    mat->isTran = 0;  // 假设不进行矩阵转置
    mat->data = (float *)mmap(NULL, rows * cols * sizeof(float), PROT_READ |
                PROT_WRITE, MAP_PRIVATE | MAP_ANONYMOUS, -1, 0);

    if (mat->data == MAP_FAILED) {
        perror("mmap");
        exit(1);
    }

    // 为矩阵的每个元素分配一个随机的float值
    for (size_t i = 0; i < rows * cols; i++) {
        mat->data[i] = (float)rand() / RAND_MAX;
    }

    return mat;
}

void freeMatrix(struct Matrix *mat) {
    munmap(mat->data, mat->rows * mat->cols * sizeof(float));
    free(mat);
}
```



知乎 @cpp后端技术

3.3 计算-通信平衡公式 **Pumpkin And Honey Bunny**

在整场决斗中，我们在单核内要尽可能地减少指令，把数据尽可能地往 Cache 放。而在多核上在并行算力增加的同时，考虑通信的成本。这是一个复杂的系统优化工程，而我们有下面几个分析工具：[11]

1. Parallel Computing

Amdahl's Law: 这个公式帮助我们衡量我们的并行提升的有效能力。

[10] 对 mmap 不错的介绍：https://zhuanlan.zhihu.com/p/691717824

[11] 还有一个访存的公式我找不到了，大概是分析计算机在各内存结构间访问的速度。还有一个公式集是关于 OpenMP 的创建线程以及线程间通信速度，具体可以看参考文献中的文章。

$$S = \frac{1}{1 - \alpha + \frac{\alpha}{k}}$$

2. CPU

$$\text{CPU Time} = \text{Instruction Count(IC)} \times \text{Cycles per Instruction(CPI)} \times \text{Clock Period}(T_c)$$

3. Cache

Cache Rule:

大小为 N 的直接相联和大小为 N/2 的 2 路组相联的缺失率差不多。[12]

Hit Probability in Cache[27]:

$$P_{hit} = \sum_{a=0}^{a=A-1} \binom{D}{a} \left(\frac{A}{B}\right)^a \left(\frac{B-A}{B}\right)^{(D-a)}$$

展示了一块有 A associativity, B block 的 Cache，在栈距离 D 的情况下缓存命中的概率[13]。

Effective Access Time:

Effective Access Time = Hit rate × Cache access time + Miss rate × Lower level access time

4. GPU

GPU 浮点运算理论峰值 **Peak FLOPS**:

$$P = \text{FP32 Core} \times f_{GPU} \times \text{Operations per cycle}$$

Part 4: SGEMM 测试、观察

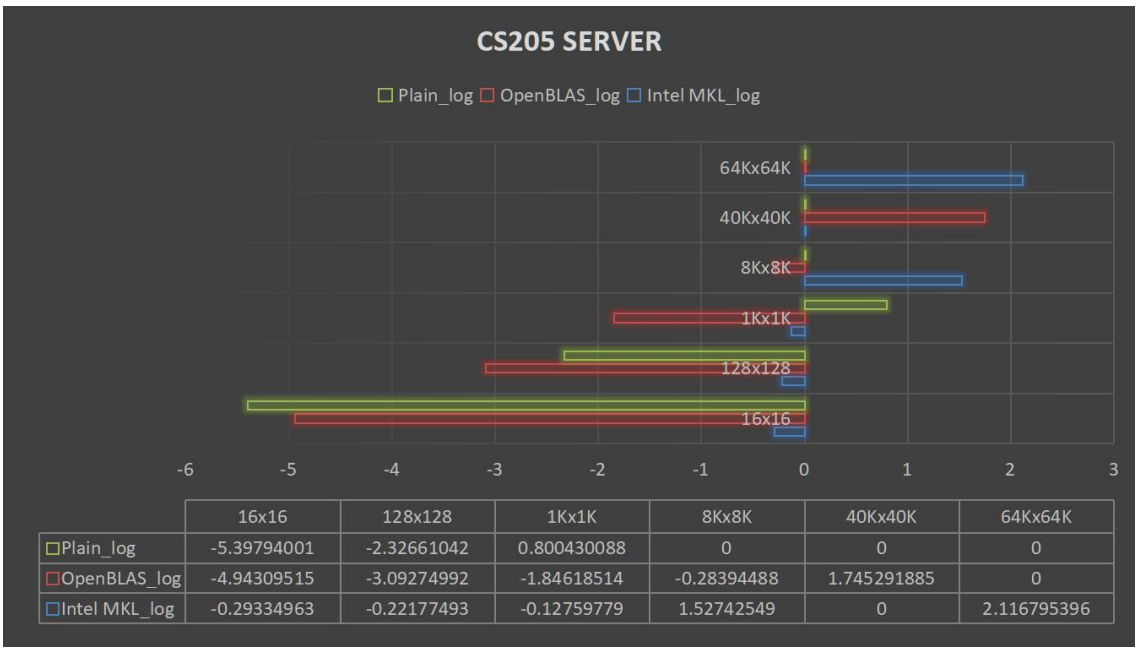
Chapter 4: THE HATEFUL EIGHT

本节我们将进行标准 ijk，OpenBLAS，Intel MKL 针对不同大小的单精度浮点数矩阵的计算时间测试。我们将不同的参数输入到矩阵中，得到基准测试的结果。同时，我们会观察对手的招式：使用核数，分解动作等等。

我们首先看看 Server 的情况。Server 上受到了非常多的干扰，许多次程序都无法正常运行，在 64K x 64K，40K x 40K 上都没有收集到完整数据。再加上老师说过高耗时的程序就不再分析，于是我仅列出下面 5 种矩阵规模中的部分情况。为了使数据方便展示，对条形图进行了底数为 10 的取对数操作。（原始单位为秒）

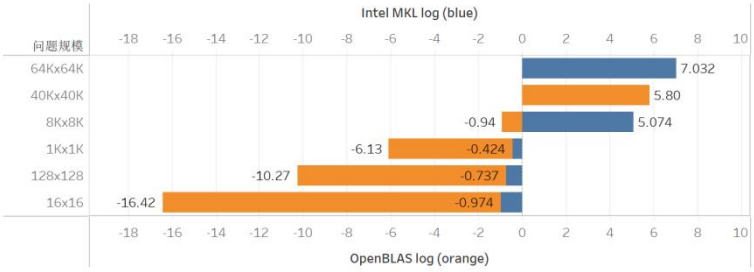
[12] 计算机体系结构 量化研究方法 https://zhuanlan.zhihu.com/p/675410026

[13] 具体的公式推导可看论文[27]的附录。

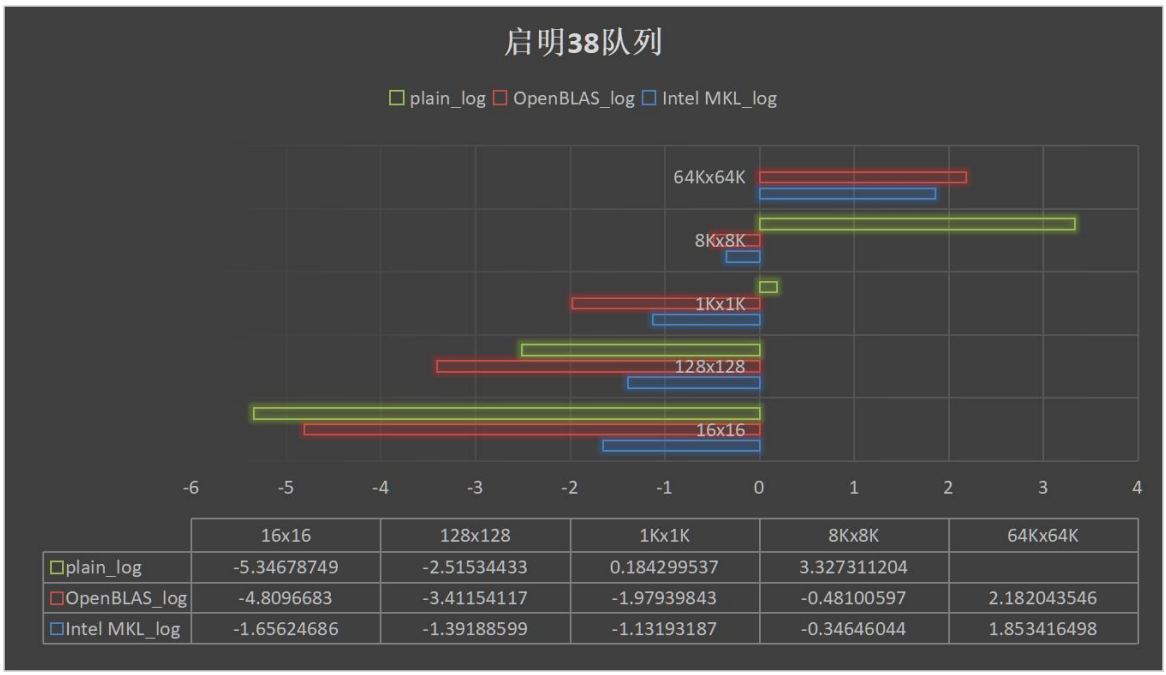


	16x16	128x128	1Kx1K	8Kx8K	40Kx40K	64Kx64K
Plain_log	-5.39794001	-2.32661042	0.800430088	0	0	0
OpenBLAS_log	-4.94309515	-3.09274992	-1.84618514	-0.28394488	1.745291885	0
Intel MKL_log	-0.29334963	-0.22177493	-0.12759779	1.52742549	0	2.116795396

问题规模	OpenBLAS	Intel MKL	Plain
16x16	0.0000114	0.508921	0.000004
128x128	0.0008077	0.600102	0.004714
1Kx1K	0.01425	0.745422	6.315825
8Kx8K	0.520062	33.684142	
40Kx40K	55.6278		
64Kx64K		130.856529	



之后我也在启明上进行了测试：



	16x16	128x128	1Kx1K	8Kx8K	64Kx64K
plain_log	-5.34678749	-2.51534433	0.184299537	3.327311204	
OpenBLAS_log	-4.8096683	-3.41154117	-1.97939843	-0.48100597	2.182043546
Intel MKL_log	-1.65624686	-1.39188599	-1.13193187	-0.34646044	1.853416498

Problem Size	OpenBLAS	Intel MKL	plain
16x16	0.0000155	0.0220675	4.50E-06
128x128	0.000387667	0.0405615	0.0030525
1Kx1K	0.0104858	0.073802	1.52862
8Kx8K	0.330365	0.450339	2124.766469
64Kx64K	152.07	71.3537	

在启明上的数据相对干扰不大，我们也可以看到三种实现方法的区别。在小数据内，直接采用 plain 方法是最快的，这是因为 OpenBLAS 和 MKL 还在传参时，Plain 方法就基本上算完了矩阵。但是很快中型数据规模下 OpenBLAS 占据了主导。从汇编代码分析，这应是因为 OpenBLAS 在小数据规模的矩阵乘法上做出了许多优化，而 Intel 并没有特别聚焦这部分的内容[14]，从文章[15]可以看出，他们居然在用 JIT 做小矩阵的优化，它会将我们写的部分代码转换 AVX2 或者 AVX512，对不同的 CPU 型号采用不同的 OpenMP 线程，不同的策略。它的规模是小于 16 和 128 的矩阵。但是很不幸，这样的优化似乎 OpenBLAS 做的更好……（可能只是他们的员工留下了优化的空间，这样以后自己就不会被过早优化了）

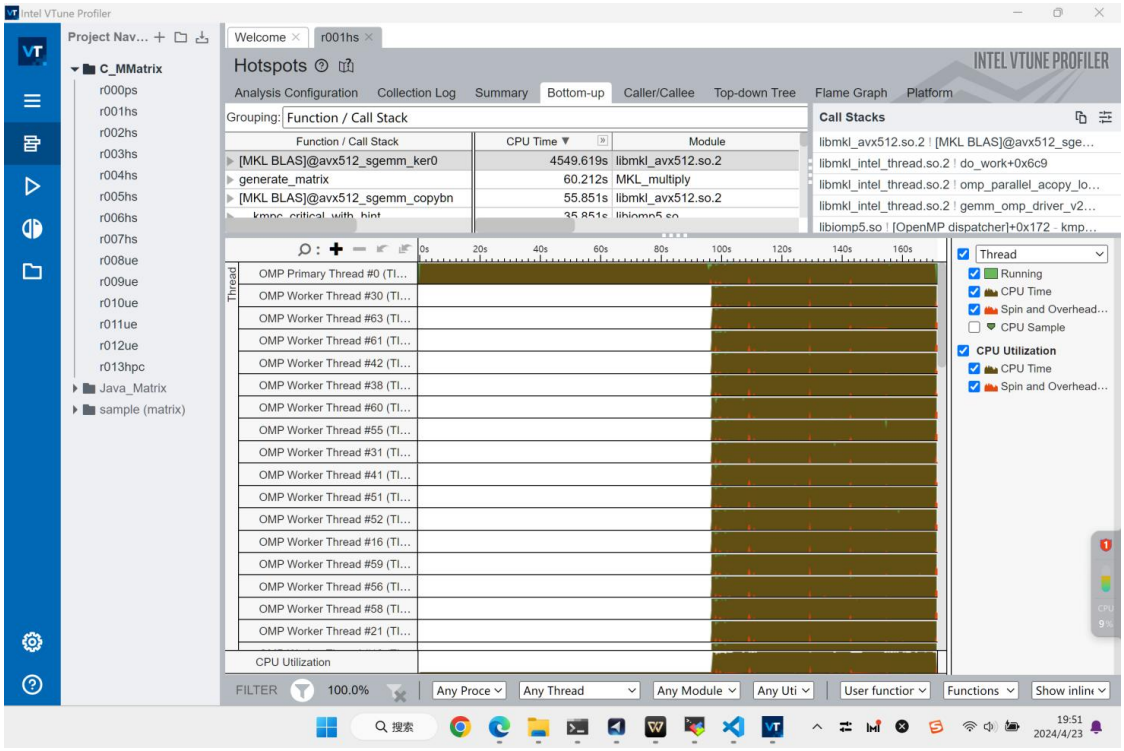
但是 MKL 在大矩阵的计算上超越了 OpenBLAS。在 64K x 64K 上领先对手一倍的速度。根据以前学长的测试，大矩阵的优化都是 MKL 较佳，在跑 HPL 时也是统一用的 Intel MKL。不过 MKL 是怎么做到的？我决定来分析一下。于是首先请出我们的老朋友 VTune。

我们对 Intel MKL 和 OpenBLAS 分别进行分析:

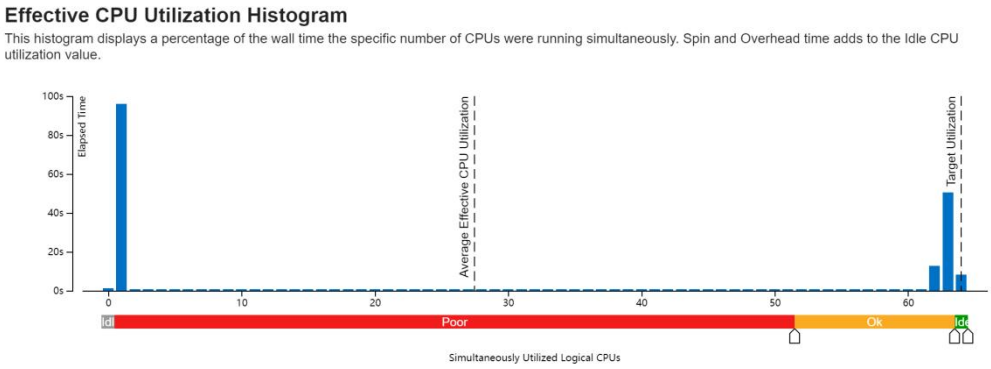
```
Top Hotspots
Function                        Module                  CPU Time  % of CPU Time(%)
------------------------------  ----------------------  --------  ----------------
[MKL BLAS]@avx512_sgemm_ker0    libmkl_avx512.so.2         7.421s             46.4%
wait_for_work                   libmkl_intel_thread.so.2   3.939s             24.6%
generate_matrix                 MKL_multiply               0.812s              5.1%
__kmpc_critical_with_hint       libiomp5.so                0.610s              3.8%
rand                            libc.so.6                  0.498s              3.1%
[Others]                        N/A                        2.700s             16.9%
Effective CPU Utilization: 12.7%
 | The metric value is low, which may signal a poor logical CPU cores
 | utilization caused by load imbalance, threading runtime overhead, contended
 | synchronization, or thread/process underutilization. Explore sub-metrics to
 | estimate the efficiency of MPI and OpenMP parallelism or run the Locks and
 | Waits analysis to identify parallel bottlenecks for other parallel runtimes.
 |
    Average Effective CPU Utilization: 4.049 out of 32
```

```
vtune: Collection stopped.
vtune: Using result path `/data/S12211612/haibin/Project3/OpenBLAS/r001ps'
vtune: Executing actions 19 % Resolving information for `matrix_multiply'
vtune: Warning: Cannot locate debugging information for file `/data/S12211612/haibin/Project3/OpenBLAS/matrix_multiply'.
vtune: Executing actions 19 % Resolving information for `ld-linux-x86-64.so.2'
vtune: Warning: Cannot locate debugging information for file `/lib64/ld-linux-x86-64.so.2'.
vtune: Executing actions 19 % Resolving information for `libc.so.6'
vtune: Warning: Cannot locate debugging information for file `/lib/x86_64-linux-gnu/libc.so.6'.
vtune: Executing actions 21 % Resolving information for `libopenblas.so.0'
vtune: Warning: Cannot locate debugging information for file `/lib/x86_64-linux-gnu/libopenblas.so.0'.
vtune: Executing actions 75 % Generating a report                    Elapsed Time: 1.120s
    Total Thread Count: 48
```

我们来看看 Intel MKL 的热力图：

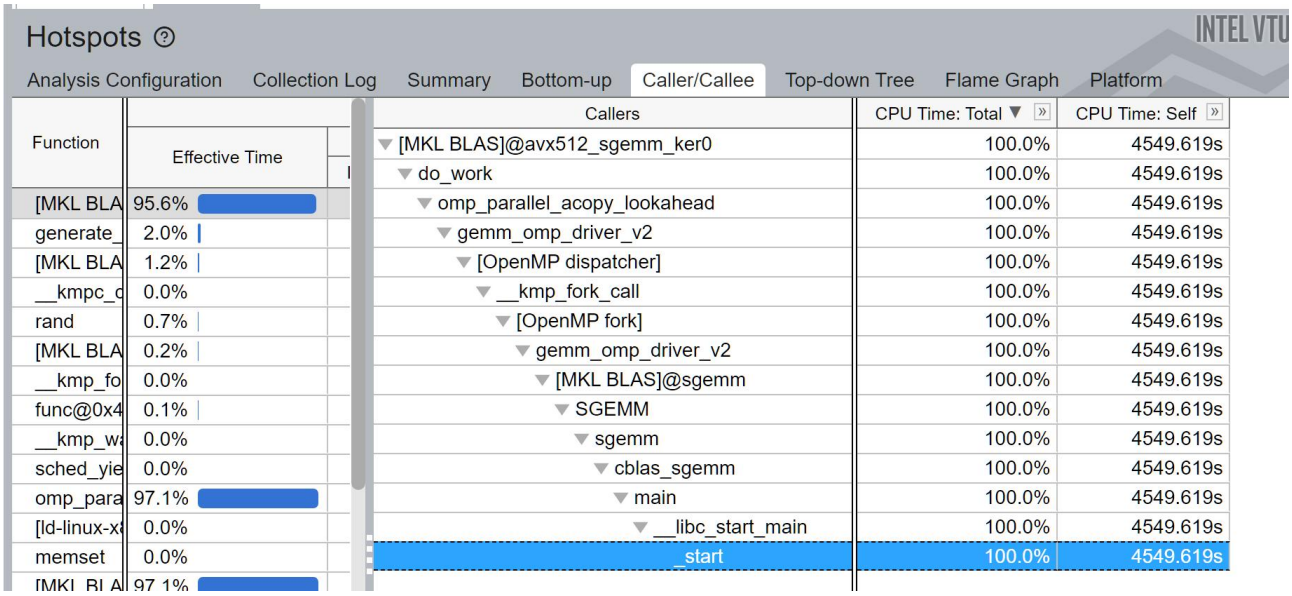


[14] ASC 中山大学队给我们分享了文章[15]，Intel MKL 在小规模矩阵上做出的努力，感谢他们。

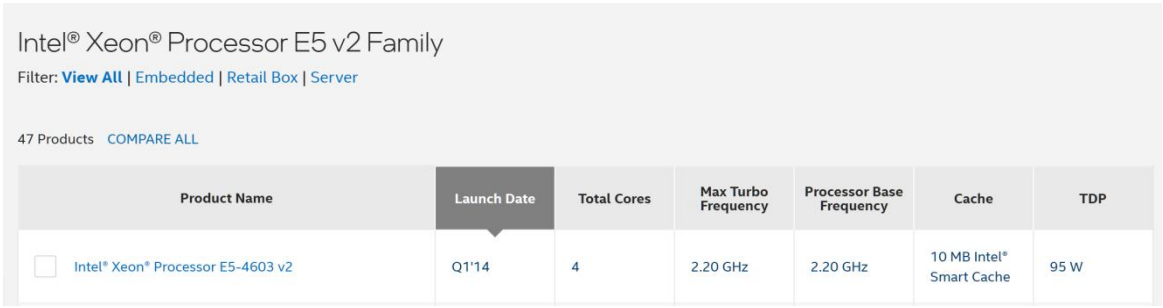


开始的单线程是我写的初始化矩阵，不记入时间。在矩阵初始化完成后，OpenMP 就开始创建 63 个 Worker Thread，他们会同步开始工作。有趣的是，这些 CPU 线程在 Spin 等待状态的时间几乎一致，且在程序终止前也进行了 Spin，推测可能是在进行数据同步。

MKL 似乎很精准地知道我使用了 64 个核，采用了 63 核的计算设计，并且尽可能地将 CPU 的使用频率用到了最满。那么 MKL 用了哪些代码呢？我们看 Caller/Callee。



可以看到，程序先是进入了 Project2 我们探索过的_start 和__libc_start_main 启动函数，然后进入我们的 main 函数中。接着，函数进入到 MKL 的 SGEMM 函数中。[MKL BLAS]@sgemm 是 Intel MKL 的 Interface 层，它启动了 gemm_omp_driver_v2 driver。诶？这个 v2 是什么？是不是志强 2 代处理器的意思？

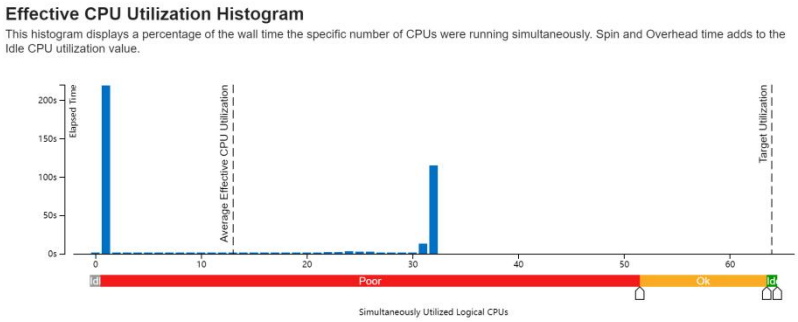


我猜测 MKL 把集群的 6338 当成 2 代处理了，但是这挺矛盾的，因为 6338 是第三代志强，Intel 怎么连自家的 CPU 都会不认识呢？当然这一切是我的推测。

回到正题，v2 driver 开启 OpenMP fork，随后 dispatcher 分发命令（也是 v2driver），acopy 拷贝数据，最后开始调用 avx512 的 sgemm kernel。可以看出，MKL 的技术架构跟 OpenBLAS 一样，是接口-驱动-内核三层操作。

接下来就没有什么好说的，kernel 如同一个强大的分布式机器，将所有的任务都分发下去，中间有几次合并，到最后用一点点的时间进行汇总输出。全程手起刀落，驱动不超过 100ms,分发占用时间不超过 1 秒。MKL 就像是一个合作紧密的传说级大学 Project 小组，在组长发布命令后大家各干各的，结果还都干对了。

我们接下来看看 OpenBLAS。



OMG! OpenBLAS 你在做什么？为什么你只用了 32 个核？你在 Vtune 的运行时间是 384.76s，Intel 是 171.543s，怪不得你输的这么彻底！

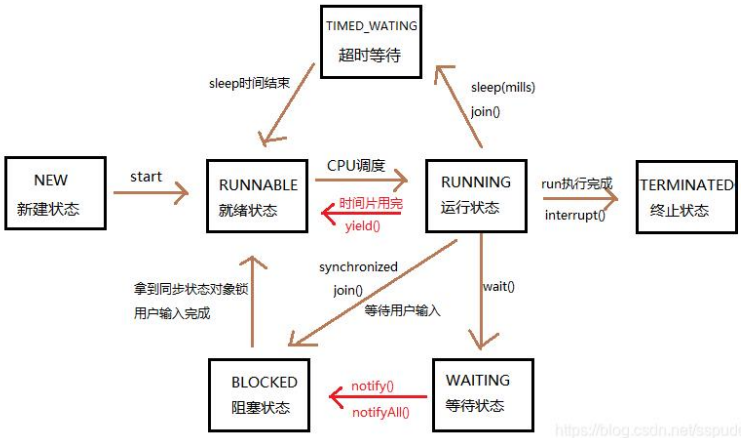
Function	CPU Time: Total			CPU Time: Self ▼	Module	Function (Full)
	Effective Time	Spin Time	Overhead Time			
sgemm_kernel	86.9%	0.0%	0.0%	4734.666s	libopenblas.so.0	sgemm_kernel
sched_yield	0.0%	7.8%	0.0%	425.632s	libc.so.6	sched_yield
rand	3.7%	0.0%	0.0%	200.552s	libc.so.6	rand
sgemm_beta	0.7%	0.0%	0.0%	35.592s	libopenblas.so.0	sgemm_beta
sgemm_itcopy	0.3%	0.0%	0.0%	14.807s	libopenblas.so.0	sgemm_itcopy
generate_matrix	4.0%	0.0%	0.0%	10.780s	matrix_multiply	generate_matrix
sgemm_oncopy	0.2%	0.0%	0.0%	9.314s	libopenblas.so.0	sgemm_oncopy
inner_thread	82.4%	7.4%	0.0%	8.279s	libopenblas.so.0	inner_thread
func@0x4007c0	0.1%	0.0%	0.0%	6.298s	matrix_multiply	func@0x4007c0
func@0x82860	0.0%	0.0%	0.0%	2.270s	libopenblas.so.0	func@0x82860
blas_thread_server	79.7%	7.3%	0.0%	0.230s	libopenblas.so.0	blas_thread_server
exec_blas_async_wait	0.0%	0.0%	0.0%	0.020s	libopenblas.so.0	exec_blas_async_wait
_start	6.7%	0.3%	0.0%	0s	matrix_multiply	_start
__libc_start_main	6.7%	0.3%	0.0%	0s	libc.so.6	__libc_start_main
start_thread	79.7%	7.3%	0.0%	0s	libpthread.so.0	start_thread
main	6.7%	0.3%	0.0%	0s	matrix_multiply	main
clone	79.7%	7.3%	0.0%	0s	libc.so.6	clone
cblas_sgemm	2.7%	0.3%	0.0%	0s	libopenblas.so.0	cblas_sgemm
sgemm_thread_nn	2.7%	0.3%	0.0%	0s	libopenblas.so.0	sgemm_thread_nn
gemm_driver	2.7%	0.3%	0.0%	0s	libopenblas.so.0	gemm_driver
exec_blas	2.7%	0.3%	0.0%	0s	libopenblas.so.0	exec_blas

跟 Intel 不同的是，OpenBLAS 并没有显示出它用了哪个具体的驱动，只有一个简单的 cblas_sgemm 接口与 gemm_driver，之后就进入到矩阵的 oncopy 和 kernel。不过，它比 MKL 多了一个 sched_yield，这是什么玩意？为什么它还会 Spin？

在网上搜索 sched_yield，发现它居然是一个 Linux 函数，它负责高级进程管理。它实现了内核中最重要的功能，当需要执行实际的调度时，直接调用 shedule()，进程就停止了，而另一个新的进程占据了 CPU.[15]。

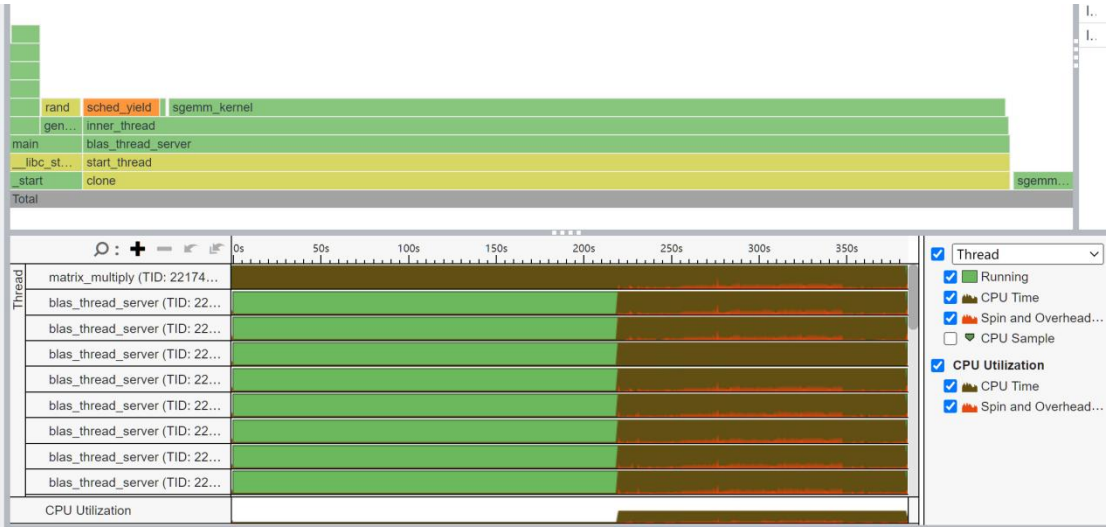
sched_yield 让出 CPU 后，该线程处于就绪状态，而调用 sleep 后，线程处于阻塞状态，唤醒后需要先转为就绪状态才能执行。这一块一眼 OS，所以直接掏出上学期学的操作系统：

[15] https://zhuanlan.zhihu.com/p/363791563



原来如此，原来如此，死去的记忆突然开始攻击我。看来我们的程序在运行途中被终止，然后被另一个线程所占用，这导致了我们的 Spin time 非常的高。为什么呢？

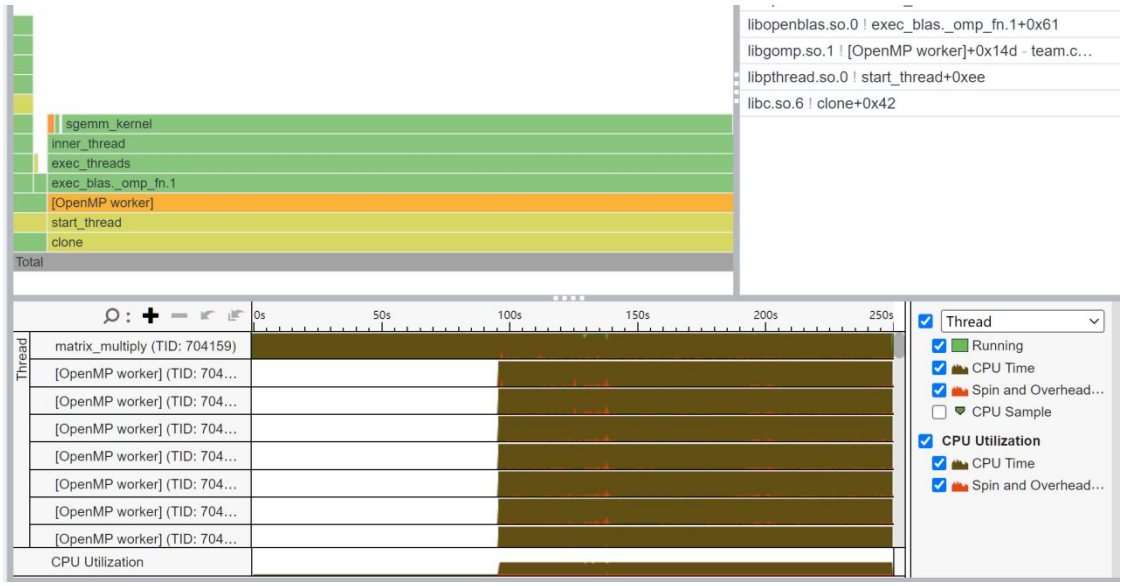
仔细观察，好像这里边 OpenBLAS 没有 OpenMP 的函数？看热力图里也没有调用。并且 CPU 的 Spin 频率非常高，看起来不像是在做纯计算，而是在通信什么的。难道此时 OpenBLAS 是多进程模型？



我又查询了一下，发现网上有人问编译时的问题，总结后我进行了这么一个尝试：

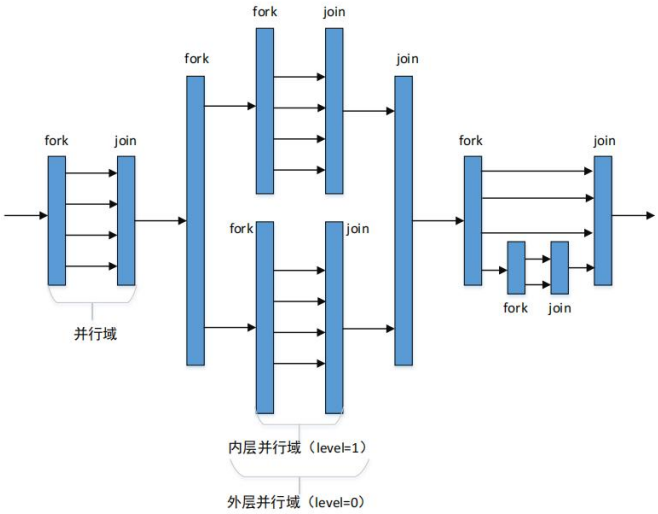
```
make USE_OPENMP=1
```

再来一次：啊？



OpenBLAS 的 OpenMP 居然是手动开的，那我这波还得重测之前的数据。那等等，那 OpenBLAS 在之前的线程并行上用的是谁？

于是我阅读了文章[29]，这篇文章讲述的是 OpenMP 是如何实现的。OpenMP 使用 fork-join 并行机制，程序开始串行执行，此时只有一个主线程，然后在遇到用户定义的并行区域时创建出一组线程。在并行区域之内，多个线程可以执行相同的代码块，或使用工作共享结构体并行执行不同的任务。



从系统层来说，OpenMP 在并行域用的是操作系统的线程，这些线程在物理上会对应到我们服务器的具体的各个 CPU 上。

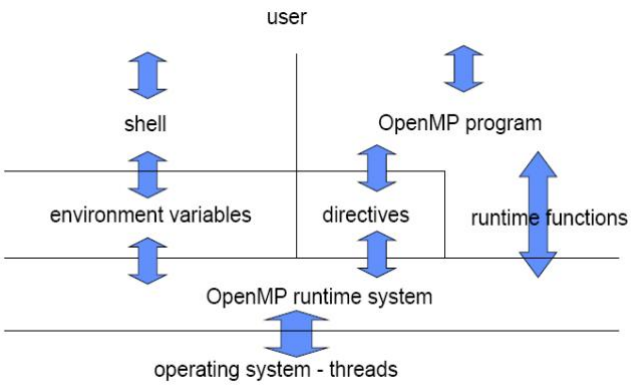


图 2.3 OpenMP 体系结构

但是，具体 OpenMP 是怎么分配线程和数据的呢？我又读了文章，发现了在多处理器上的 DOALL 循环调度算法，这里边有因式分解法调度和梯形自调度算法。这类调度算法应该考虑数据的位置来提高并行循环的性能。但是，DOALL 算法不一定是最好的调度算法，还存在别的几个针对并行的调度算法如 DOACROSS 循环调度策略算法。有兴趣的朋友可以看看文章[30]，这里边分析了几种 OpenMP 调度的机制。

表 2.1　几种调度算法的优缺点对比

算法	调度步数	分块大小	优点	缺点
自调度	N	1	负载均衡	需要 N 个调度步骤。 只能用于访问共享变量的开销较小的系统。 网络和内存争夺的可能性非常高。 网络和内存的争夺是一个主要问题。
固定大小分块调度	P	N/P	需要最少的调度步骤。 块的大小可以在编译时或运行时在执行循环之前确定。	负载均衡不是很好，特别是如果迭代执行时间的差异很大时。
引导自调度	P ln[N/P]	R/P	在负载平衡和调度开销之间的权衡。 调度步骤数在前两种算法之间，并尝试通过平衡工作负载来处理迭代次数的变化。	早期的分块可能会很大，有可能在所有其他块都已经完成的时候还没有完成。 每一步都必须计算当前块的大小。
因式分解	P[1.44ln(N / P)]	R/2P	分配比引导自调度更小的块，以平衡工作量。	比引导自调度更多的调度步骤

因此我们可以看到，在这些机制当中，我们在设计 OpenMP 的时是尽可能希望负载均衡的，也就是让所有的任务基本上在同一时间完成。这听起来挺简单，但是我想起来数据库的 Project 里的负载均衡，我就知道这个策略有多难了，并且一定多多少少会有一些落后的线程。于是我又查到，其实我们写的 OpenMP，背后都会进行**隐式同步**，意思就是说有个 omp barrier,必须所有线程都到达 barrier，程序才能继续下去。

表 3.4 隐式同步显式化

OpenMP 源代码	转换后的 OpenMP 源代码

#pragma omp parallel
{…}
	#pragma omp parallel
	{
	…
	#pragma omp barrier
	}

#pragma omp for
for(…)
{…}
	#pragma omp for nowait
	for(…)
	{…}
	#pragma omp barrier

#pragma omp sections
{
#pragma omp section
{…}
#pragma omp section
{…}
}
	#pragma omp sections nowait
	{
	#pragma omp section
	{…}
	#pragma omp section
	{…}
	}
	#pragma omp barrier

#pragma omp single {…}	#pragma omp single nowait {…} #pragma omp barrier

#pragma omp parallel for
{…}
	#pragma omp parallel
	{
	#pragma omp for nowait
	{…}
	#pragma omp barrier
	}

#pragma omp parallel sections
{…}
	#pragma omp parallel
	{
	#pragma omp sections nowait
	{…}
	#pragma omp barrier
	}

，

这听起来挺有意思的。以前我对 OpenMP 的印象就是咱们加行代码，for 循环就可以噼里啪啦地加速了。但是看完这一行背后的算法、优化还有看不懂的硕士论文，我对它肃然起敬。

好我们递归回到上一个问题：如果不用 OpenMP，OpenBLAS 之前是怎么启动这个 32 个 Worker 线程的呢？我这里将两次不同编译过的 OpenMP 的源码进行了对比，尤其关注在 OpenBLAS 开启线程时的代码：

OpenBLAS - OpenMP

```
1.    Address Source Line Assembly CPU Time: Total CPU Time: Self
2.    0x2aed2e 0 Block 23:
3.    0x2aed2e 0 movq  0x28(%rax), %rdx
4.    0x2aed32 0 movq  0x20(%rax), %rsi
5.    0x2aed36 0 movq  0x8(%rax), %r9
6.    0x2aed3a 0 callq  %r10 60.8% 0s
```

OpenBLAS - pthread

```
1.    Address Source Line Assembly CPU Time: Total CPU Time: Self
2.    0x2aed2e 0 Block 23:
3.    0x2aed2e 0 movq  0x28(%rax), %rdx
4.    0x2aed32 0 movq  0x20(%rax), %rsi
5.    0x2aed36 0 movq  0x8(%rax), %r9
6.    0x2aed3a 0 callq  %r10 60.8% 0s
```

可以看到，两个版本在记录或者打标签后，直接 call 开启线程的方法了。寄存器里发生了什么我们也不得而知[16]。

正当我以为要山穷水尽时，柳暗花明之间我看到了他们的调用的动态库：libpthread.so.0。

看来 OpenBLAS 用的是 POSIX 标准里的 pthread 线程库。POSIX 是可移植操作系统接口(Portable Operating System Interface) 的缩写。它是在 IEEE 1003.1 标准下定义了应用程序（以及命令行 Shell 和实用程序接口）和 UNIX 操作系统之间的语言接口[17]。就像 BLAS 库希望能在各种 CPU 中发挥最大的价值一样，POSIX 的目标就是可移植，而这里边就定义了线程的接口。

但是比较神秘的是，OpenBLAS 在使用了 POSIX 后依然使用了 OpenMP，可能这里边是用 OpenMP 做更高级的通信以及 api。但是具体我没有研究。但是，我可以说说 OpenMP 跟 POSIX 底层里发生了什么。这就要引入我们的 Shared Memory Programming Model 了。

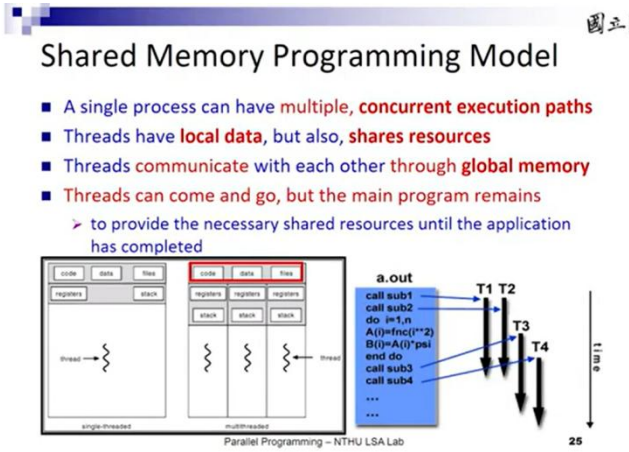
OpenMP 跟 POSIX pthread 有什么区别呢？

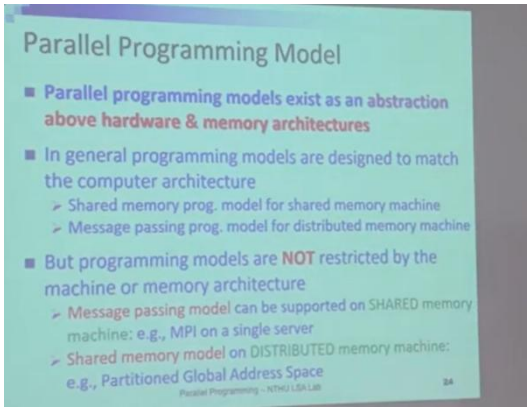
对于一个单独的进程，我们可以有多个执行的线程，每个线程有自己的数据，在全局也可以有共享的数据。每一个线程可以通过修改全局数据来实现相互通信。

---

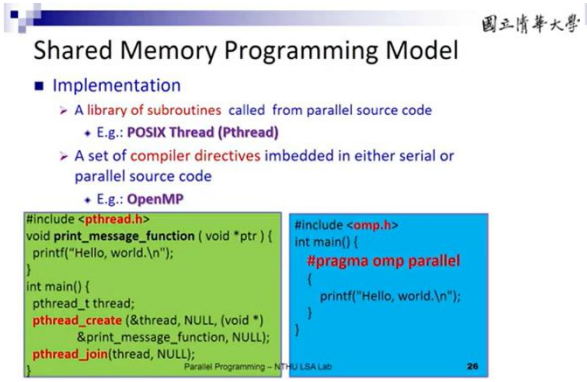
[16] “l”表示长整数（32 位），“w”表示字（16 位），“b”表示字节（8 位），"."点开始的指令一般是伪指令，"$"符号修饰立即数，"%"修饰寄存器

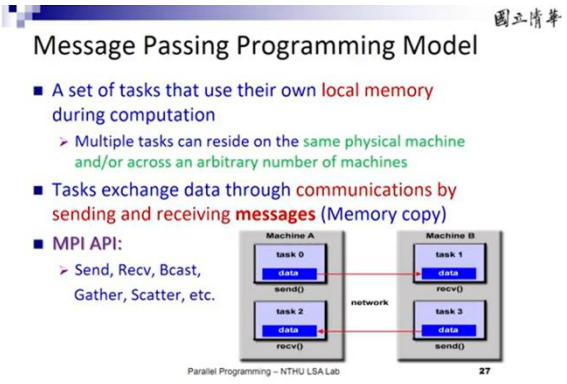
[17] https://zhuanlan.zhihu.com/p/460426074



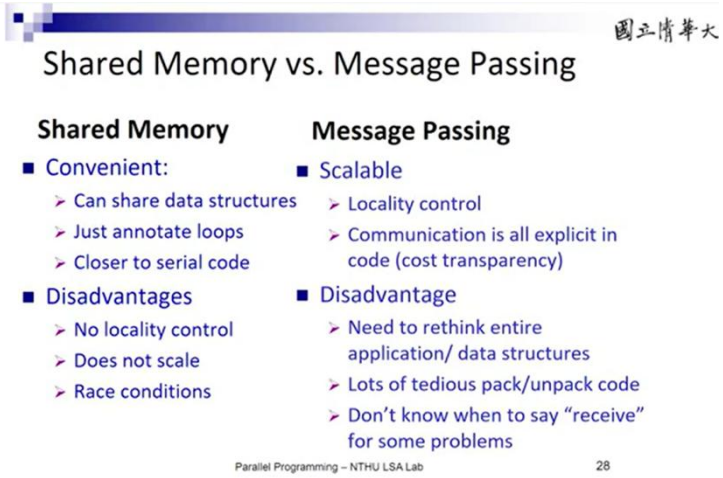


那么在实现层面，有两种实现的方法。方法 1 是创造出 pthread 库，然后运用线程库创造线程，销毁线程等等。另一种实现的方法是在编译的时候交给编译器处理，也就是 OpenMP。





这是 SMPM，共享内存编程模型的实现的想法。另一种并行模型 MPI，是消息通信模型。这个模型比起使用一个共同的共享内存等想法，它更多是使用 Mem copy 来将数据通信到别的线程的内存中。MPI 的好处就在于它不再受到 L2 L3 缓存等物理限制，在网络等连接下，它可以不断地扩展。



这两种模型各有好坏。像 OpenMP 这样的模型对底层更加接近，对 L2L3 Cache 的亲和力更强，在同一台节点内的速度比 MPI 更强，MPI 是直接在相应的线程内复制通信，而 OpenMP 的灵活度更高。但是 OpenMP 就无法进行多节点的设计。因此更多时候程序设计应该是考虑 OpenMP+MPI 的操作，这也是 HPL 的设计模式。

另外值得一提的是，MKL 用的是自研的 iomp，它也实现了 OpenMP 的接口，但是不开放给我们使用。MKL 基本上算是闭源生态。

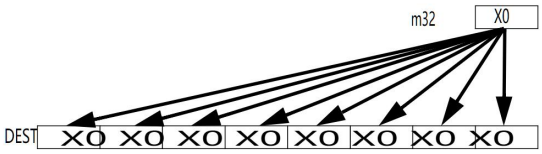
在 OpenBLAS 的 kernel 部分是长长的 SIMD 指令集。vfmadd231ps 是一个融合乘加指令，具体如下[18]：

```
VFMADD231PS DEST, SRC2, SRC3
IF (VEX.128) THEN
    MAXNUM ←4
ELSEIF (VEX.256)
    MAXNUM ← 8
FI
For i = 0 to MAXNUM-1 {
    n ← 32*i;
    DEST[n+31:n] ← RoundFPControl_MXCSR(SRC2[n+31:n]*SRC3[n+31:n] + DEST[n+31:n])
}
IF (VEX.128) THEN
    DEST[MAXVL-1:128] ← 0
ELSEIF (VEX.256)
    DEST[MAXVL-1:256] ← 0
FI
```

VBROADCAST 是从内存中拿出浮点数，然后给各个核计算。

VBROADCAST — Load with Broadcast Floating-Point Data

Opcode/Instruction	Op/En	64/32 Bit Mode Support	CPUID Feature Flag	Description
VEX.128.66.0F38.W0 18 /r VBROADCASTSS xmm1, m32	A	V/V	AVX	Broadcast single precision floating-point element in mem to four locations in xmm1.



然后就是，更新循环（add），load 数据，计算乘加，重复 1 万次[19]。这里边有个道道，就是我们的乘加指令跟我们的 load 指令应该如何分配的问题，是每 2 次加指令然后 load 还是每 3 次加指令然后 load。我们这里就可以对不同的机器、指令进行优化。

```
Address Source Line Assembly CPU Time: Total CPU Time: Self
0x2bd7df 0 vfmadd231ps %zmm4, %zmm7, %zmm18 0.1% 2.890s
0x2bd7e5 0 vfmadd231ps %zmm5, %zmm7, %zmm19 0.0% 0.750s
0x2bd7eb 0 add $0x10, %r8 0.0% 0.340s
0x2bd7ef 0 vbroadcastsdq (%r15), %zmm6 0.0% 0.110s
0x2bd7f5 0 vfmadd231ps %zmm4, %zmm6, %zmm20 0.0% 0.310s
0x2bd7fb 0 vfmadd231ps %zmm5, %zmm6, %zmm21 0.0% 0.480s
0x2bd801 0 vbroadcastsdq 0x8(%r15), %zmm7 0.0% 0.270s
0x2bd808 0 vfmadd231ps %zmm4, %zmm7, %zmm22 0.0% 2.190s
0x2bd80e 0 vfmadd231ps %zmm5, %zmm7, %zmm23 0.0% 0.620s
0x2bd814 0 vbroadcastsdq (%r15,%r12,1), %zmm6 0.0% 0.300s
0x2bd81b 0 vfmadd231ps %zmm4, %zmm6, %zmm24 0.0% 0.310s
0x2bd821 0 vfmadd231ps %zmm5, %zmm6, %zmm25 0.0% 0.330s
0x2bd827 0 vbroadcastsdq 0x8(%r15,%r12,1), %zmm7 0.0% 0.180s
0x2bd82f 0 vfmadd231ps %zmm4, %zmm7, %zmm26 0.0% 1.330s
0x2bd835 0 vfmadd231ps %zmm5, %zmm7, %zmm27 0.0% 0.440s
0x2bd83b 0 vbroadcastsdq (%r15,%r12,2), %zmm6 0.0% 0.250s
0x2bd842 0 vfmadd231ps %zmm4, %zmm6, %zmm28 0.0% 0.320s
0x2bd848 0 vfmadd231ps %zmm5, %zmm6, %zmm29 0.0% 0.390s
0x2bd84e 0 vbroadcastsdq 0x8(%r15,%r12,2), %zmm7 0.0% 0.350s
0x2bd856 0 vfmadd231ps %zmm4, %zmm7, %zmm30 0.0% 0.670s
0x2bd85c 0 vfmadd231ps %zmm5, %zmm7, %zmm31 0.0% 0.340s
```

[18] https://leria-info.univ-angers.fr/~jeanmichel.richer/assembly/doc/vfmadd132ps.pdf

[19] Broadcast 文献来源 https://www.felixcloutier.com/x86/vbroadcast

文章[4]讲述的就是在 HPL 的 BLAS 上进行优化处理。HPL 上使用的是 DGEMM，跟 SGEMM 比较类似。我们可以在后面介绍一下 HPL。在他们的机型上，BLAS 底层的 SIMD 指令中的计算-访存就是 3:1。而文章[28]是中科院对国产申威处理器进行的 BLAS 优化，在这里他们的计算-访存指令比就是 1:1。他们提到，申威处理器硬件提供两条流水线，无数据依赖的计算与访存可同时发射，则指令排布应尽量保证一条计算指令和一条取数指令共同出现，使得两条流水线都占满。

而两条流水线在执行命令的时候，会采用比较经典的 OoO（乱序执行）的操作，因此这样的指令排布会很好地吃尽 CPU 的资源[20]，有课程[19]中进行了专门介绍（安利一下）。

```
.....
1   prefetcht0 64(%%rax)
.....
2   vmovapd 0 * 8(%%rbx),%%xmm1
3   vmovapd 2 * 8(%%rbx),%%xmm2
4   vmovapd 4 * 8(%%rbx),%%xmm3
5   vmovddup 0 * 8(%%rax),%%xmm0
6   vfmadd231pd   %%xmm1,%%xmm0,%%xmm4
7   vfmadd231pd   %%xmm2,%%xmm0,%%xmm5
8   vfmadd231pd   %%xmm3,%%xmm0,%%xmm6
9   vmovddup 1 * 8(%%rax),%%xmm0
10  vfmadd231pd   %%xmm1,%%xmm0,%%xmm7
11  prefetcht0 128(%%rax)
12  vfmadd231pd   %%xmm2,%*%xmm0,%%xmm8
13  vfmadd231pd   %%xmm3,%%xmm0,%%xmm9
14  vmovddup 2 * 8(%%rax),%%xmm0
15  vfmadd231pd    %%xmm1,%%xmm0,%%xmm10
16  vfmadd231pd    %%xmm2,%%xmm0,%%xmm11
17  vfmadd231pd    %%xmm3,%%xmm0,%%xmm12
18  vmovddup 3 * 8(%%rax),%%xmm0
19  vfmadd231pd    %%xmm1,%%xmm0,%%xmm13
20  vfmadd231pd    %%xmm2,%%xmm0,%%xmm14
21  vfmadd231pd    %%xmm3,%%xmm0,%%xmm15
.....
22  vmovddup 8 * 8(%%rax),%%xmm0
```

Fig.6　Code segment of inner-most DGEMM kernel

图 6　DGEMM 核心汇编代码片段

```
VMAD    a0,b1,t04,t04
LDDE    nb1,5*SIZE(B)

VMAD    a4,b0,t01,t01
VLD     na12,12*SIZE(A)

VMAD    a4,b1,t05,t05
VLD     na8,8*SIZE(A)

VMAD    a0,b2,t08,t08
LDDE    nb2,6*SIZE(B)

VMAD    a0,b3,t12,t12
LDDE    nb3,7*SIZE(B)

VMAD    a8,b0,t02,t02
VMAD    a8,b1,t06,t06

VMAD    a4,b2,t09,t09
addl    B,8*SIZE,B
VLD     na0,0*SIZE(A)
```

图 1　核心汇编代码举例

很好，那我们还有最后一个问题：为什么这里 OpenMP 只用了 32 个线程？

答案很愚蠢：因为我是在启明的登录节点上进行的编译。登录节点上只有 32 个核，这导致我们的程序最终只使用了一半的核数。这也告诉我们 OpenBLAS 的一个弱点：必须要在相应的节点上才能产生相应的最好的 driver 和 kernel。但是 MKL 就不是，MKL 作为动态库可以自动选择，不用担心数学库和机型不匹配的问题。

我又重新测了一遍 OpenBLAS，这回，数据变的比较接近了：

```
Generate complete, starting computation using
Job finished, Analyzing the correctness......
Time elapsed: 82.739585 seconds.
Your job look like:
   M:64000, N:64000, K:64000
Starting Matrix initialization......
Generate complete, starting computation using
Job finished, Analyzing the correctness......
Time elapsed: 83.675808 seconds.
Your job look like:
   M:64000, N:64000, K:64000
Starting Matrix initialization......
Generate complete, starting computation using
Job finished, Analyzing the correctness......
Time elapsed: 84.081302 seconds.
Your job look like:
   M:64000, N:64000, K:64000
Starting Matrix initialization......
Generate complete, starting computation using
Job finished, Analyzing the correctness......
Time elapsed: 83.712126 seconds.
Your job look like:
   M:64000, N:64000, K:64000
Starting Matrix initialization......
Generate complete, starting computation using
Job finished, Analyzing the correctness......
Time elapsed: 83.177541 seconds.
```

虽然还是没能打过 MKL 的 70 秒，但是 83 秒的平均成绩也是非常优秀的了。在我意识到编译这个问题之前，我是以为 OpenBLAS 在判断失误。它可能认为 NUMA 节点的通信损失大于了计算带来的优势。

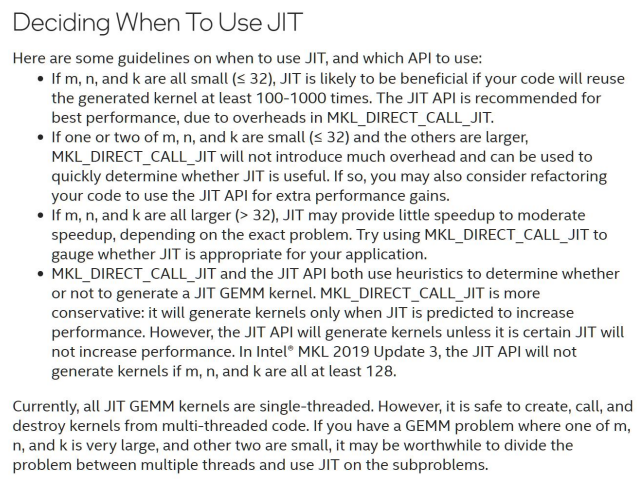
“访存-计算”平衡是我们接下来需要考虑的一个很重要的点。另外就是，MKL 确实作为一个闭源的数学库，在加速方面还是挺有实力的。

[20] 有个教程讲的挺好，虽然是关于 CUDA 的：
https://www.bilibili.com/video/BV1kx411m7Fk/?spm_id_from=333.999.0.0&vd_source=4871cfa497362c1a843af2ecff18ab7f

我们来看看 MKL 还有什么黑科技：

首先第一个，对于稀疏矩阵，MKL 提供了对稀疏矩阵的优化 API[21]，OpenBLAS 似乎没有看到（可能单纯是我没有找到发现）。在稀疏矩阵直接法求解中一个常见的技术是矩阵重排，通过矩阵重排，可以使得矩阵的带宽减小（非 0 元素位置相对更趋近于对角线），从而使得直接求解效率更高，一种常见的矩阵重排方法是 Cuthill_McKee 算法[22]。不过 MKL 在这里用的是基于 FEAST 的子空间投影算法和经典的 Krylov 子空间迭代方法。

第二个黑科技是 ASC 里中山大学的队长告诉我们的：MKL 会使用 JIT 技术对小规模矩阵的乘法进行加速[15]。这个 JIT 并非像我们 Project2 里 Java 的 JIT，把字节码翻译成机器码，它是直接改动底层的自己的机器码。面对小矩阵，我们从刚刚的测试中可以看到 MKL 的表现其实并没有特别的好。我个人推测 MKL 可能它自己在内部有很多的选择分支以及线程分配，导致在小矩阵上通信的成本增加了。而 JIT 技术就会将 Kernel 的大小、循环调整以及我们的计算-访存指令比例为最适合的我们机器的形状，随后进行加速计算。

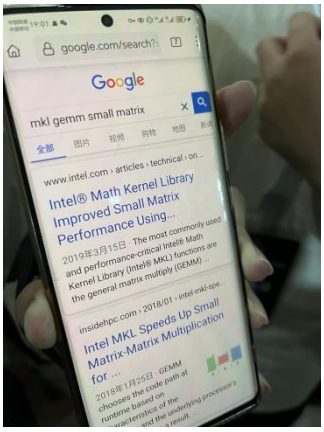


Deciding When To Use JIT

Here are some guidelines on when to use JIT, and which API to use:

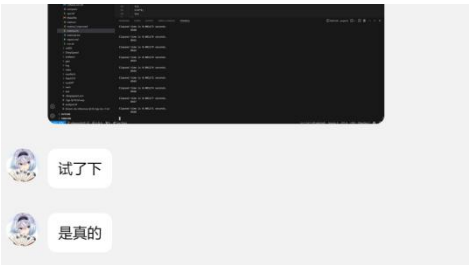
- If m, n, and k are all small (≤ 32), JIT is likely to be beneficial if your code will reuse the generated kernel at least 100-1000 times. The JIT API is recommended for best performance, due to overheads in MKL_DIRECT_CALL_JIT.
- If one or two of m, n, and k are small (≤ 32) and the others are larger, MKL_DIRECT_CALL_JIT will not introduce much overhead and can be used to quickly determine whether JIT is useful. If so, you may also consider refactoring your code to use the JIT API for extra performance gains.
- If m, n, and k are all larger (> 32), JIT may provide little speedup to moderate speedup, depending on the exact problem. Try using MKL_DIRECT_CALL_JIT to gauge whether JIT is appropriate for your application.
- MKL_DIRECT_CALL_JIT and the JIT API both use heuristics to determine whether or not to generate a JIT GEMM kernel. MKL_DIRECT_CALL_JIT is more conservative: it will generate kernels only when JIT is predicted to increase performance. However, the JIT API will generate kernels unless it is certain JIT will not increase performance. In Intel® MKL 2019 Update 3, the JIT API will not generate kernels if m, n, and k are all at least 128.

Currently, all JIT GEMM kernels are single-threaded. However, it is safe to create, call, and destroy kernels from multi-threaded code. If you have a GEMM problem where one of m, n, and k is very large, and other two are small, it may be worthwhile to divide the problem between multiple threads and use JIT on the subproblems.



具体的实验我并没有仔细操作。这里边要做一个对照实验：拿 MKL 计算 1000 次矩阵乘法，比较时间。拿自己实现的乘法计算 1000 次，比较时间。主要目的是看我们的 cache hit 会不会影响到我们的加速。如果在同样的缓存下，MKL 的结果还得到了加速的话，我们可以认为 MKL 就进行了加速。

郑飞扬学长拿 MATLAB 在启明上进行了测试，不过我觉得可能还要做更多的测试 。



综上，我测试了 OpenBLAS 跟 MKL 的平均运行时间。随后分析了他们的内部特性。我认为我们对抗他们的优缺点如下：

	我们的优势	OpenBLAS	MKL
核数	对自身硬件有确定性了解	并非完全了解。需要特定编译选项	对硬件了解合理，可以动态调整。但是会花费一些时间

[21]https://www.intel.com/content/www/us/en/developer/articles/technical/intel-mkl-support-for-largestsmallest-eigenvalue-and-sparse-svd-problem.html

[22] 算法介绍：https://zhuanlan.zhihu.com/p/20506574，论文：https://dl.acm.org/doi/pdf/10.1145/800195.805928

通信	采用 OpenMP，更轻量	采用 POSIX 线程库	采用自研 iomp，多核多节点强
计算访存指令占比	自行设计	2 计算/访存	动态 JIT Kernel
非计算分支	少	中等，如选择 driver	多，如 JIT
SIMD	AVX2/AVX512	AVX512	AVX512
alpha，beta	不需要	需要计算	需要计算

Part 5: 优化方案 **Chapter 5: KILL BILL**

面对 5 个规模的矩阵，我们针对每种矩阵规模，分别对 OpenBLAS 和 Intel MKL 发起挑战。我们要关注自己的访存-计算平衡，尽可能让核数跟访存对应上。下面介绍中的具体图片来自于 CPP github appendix ppt。

5.1 击败 16 x 16 **Vernita Green**

这个矩阵非常小，因此计算的时候其实更像拼刺刀，我们利用对手的分支过多，直接相乘就可以战胜了。当然为了更快，我们使用 SIMD 指令。这里我也进行了测试，用 AVX2 好还是用 AVX512 好。我的结果是 AVX2 性能更佳。

北大的队伍跟我们说，不同机型上的 AVX512 指令实现方式是不一样的。AVX512 指令集是 Intel 提出的，他们是专门做了一个处理单元。在 AVX2 的基础上数据寄存器宽度、数量以及 FMA 单元（乘积累加指令单元）的宽度都增加了一倍，所以在每个时钟周期内可以打包 8 次双精度和 16 次单精度浮点运算。但是 AMD 上的 AVX512 指令其实是靠两个 AVX2 单元组合拼接实现的。这部分的性能有机会可以做一个对比。

```
void SGEMM_AVX2(int rows1, int cols2, int cols1, float *A, float *B, float *C) {

    for (int i = 0; i < rows1; i++) {
        for (int j = 0; j < cols2; j++) {
            __m256 sum = _mm256_setzero_ps();
            for (int k = 0; k < cols1; k += 8) {
                __m256 a = _mm256_loadu_ps(&A[i * cols2 + k]);
                __m256 b = _mm256_loadu_ps(&B[k * cols1 + j]);
                sum = _mm256_add_ps(sum, _mm256_mul_ps(a, b));
            }
            __m128 sum128 = _mm_add_ps(_mm256_extractf128_ps(sum, 0), _mm256_extractf128_ps(sum, 1));
            sum128 = _mm_hadd_ps(sum128, sum128);
            sum128 = _mm_hadd_ps(sum128, sum128);
            C[i * cols2 + j] = _mm_cvtss_f32(sum128);
        }
    }
}
```

5.2 击败 128 x 128 **O-ren Ishi**

128 大小的矩阵经过测试，使用 AVX512 指令会更好。同时，将 j 和 i 的顺序调换，增加了 B 矩阵的 Cache hit 的概率（j 变量的变化更少，使得更多数据得以重复利用）。不要使用 OpenMP，我们的 L1 Cache 大小是 1.5MB，足够我们装下所有的数字（64kb）

```
void SGEMM_AVX512(int rows1, int cols2, int cols1, float *A, float *B, float *C) {
    for (int j = 0; j < cols2; j++) { //  make B's cache usage more regular
        for (int i = 0; i < rows1; i++) { // Then iterate over rows of A
            __m512 sum = _mm512_setzero_ps();
            for (int l = 0; l < cols1; l += 16) {
                __m512 a = _mm512_loadu_ps(&A[i * cols2 + l]);
                __m512 b = _mm512_loadu_ps(&B[j * cols1 + l]);
                sum = _mm512_fmadd_ps(a, b, sum);
            }
            _mm512_storeu_ps(&C[i * cols2 + j], sum);
        }
    }
}
```

Cache (e.g. Matrix Addition)

PerfXLab 澎峰科技

- i-j loop is faster
  - 2D array in C
  - Inner j loop: continuously memory access
  - Cache higher hit rate
    - ✓Matrix element is already in Cache

```
for (size_t i = 0; i < n; i++)
{
  for (size_t j = 0; j < n; j++)
  {
    C[i][j] = A[i][j] + B[i][j]
  }
}
```

```
for (size_t j = 0; j < n; j++)
{
  for (size_t i = 0; i < n; i++)
  {
    C[i][j] = A[i][j] + B[i][j]
  }
}
```

5.3 击败 1K x 1K **Budd**

1K*1K 的矩阵大约占用 3.81MB，访存-计算的平衡发生了移动。分配到每个核的话大概是 3-4 个核。我们使用 OpenMP+循环展开击败：

```
void SGEMM_AVX512_32_Kernel(int rows1, int cols2, int cols1, float* A, float* B, float* C) {
    #pragma omp parallel for
    for (int i = 0; i < rows1; i++) {
        for (int j = 0; j < cols2; j += 4) {
            __m512 sum0 = _mm512_setzero_ps();//16
            __m512 sum1 = _mm512_setzero_ps();
            __m512 sum2 = _mm512_setzero_ps();
            __m512 sum3 = _mm512_setzero_ps();

            for (int k = 0; k < cols1; k += 32) {
                // Load columns of A
                __m512 a0 = _mm512_loadu_ps(&A[i * cols2 + k]);
                __m512 a1 = _mm512_loadu_ps(&A[i * cols2 + k + 16]);

                // Load columns of B
                __m512 b0 = _mm512_loadu_ps(&B[k * cols1 + j]);
                __m512 b1 = _mm512_loadu_ps(&B[k * cols1 + j + 16]);

                // Perform matrix multiplication
                __m512 temp0 = _mm512_mul_ps(a0, b0);
                __m512 temp1 = _mm512_mul_ps(a0, b1);
                __m512 temp2 = _mm512_mul_ps(a1, b0);
                __m512 temp3 = _mm512_mul_ps(a1, b1);
                sum0 = _mm512_add_ps(sum0, temp0);
                sum1 = _mm512_add_ps(sum1, temp1);
                sum2 = _mm512_add_ps(sum2, temp2);
                sum3 = _mm512_add_ps(sum3, temp3);
            }
            // Store the results in C
            _mm512_storeu_ps(&C[i * cols2 + j], sum0);
            _mm512_storeu_ps(&C[i * cols2 + j + 16], sum1);
            _mm512_storeu_ps(&C[i * cols2 + j + 32], sum2);
            _mm512_storeu_ps(&C[i * cols2 + j + 48], sum3);
        }
```

循环展开的目的主要在于使得循环的相关性消除，让编译器在执行命令时让内部循环可以在不同的核内并行运行。同时可以减少分支预测的可能性。

5.4 击败 8K x 8K **Elle**

由于我们的数据量增大，访存-计算的平衡继续发生移动。我们需要更多的核。因此，我们尝试用更大的循环展开核。

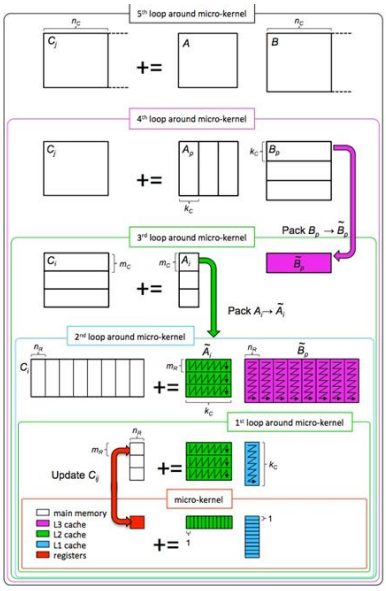
```
1.  void SGEMM_AVX512_256_kernel(int rows1, int cols2, int cols1, float* A, float* B, float* C) {
2.      #pragma omp parallel for
3.      for (int i = 0; i < rows1; i++) {
4.          for (int j = 0; j < cols2; j += 16) {
5.              __m512 sum0 = _mm512_setzero_ps();//16
6.              __m512 sum1 = _mm512_setzero_ps();
7.              __m512 sum2 = _mm512_setzero_ps();
8.              __m512 sum3 = _mm512_setzero_ps();
9.
10.             __m512 sum4 = _mm512_setzero_ps();//16
11.             __m512 sum5 = _mm512_setzero_ps();
12.             __m512 sum6 = _mm512_setzero_ps();
13.             __m512 sum7 = _mm512_setzero_ps();
14.
15.             __m512 sum8 = _mm512_setzero_ps();//16
16.             __m512 sum9 = _mm512_setzero_ps();
17.             __m512 sum10 = _mm512_setzero_ps();
18.             __m512 sum11 = _mm512_setzero_ps();
19.
20.             __m512 sum12 = _mm512_setzero_ps();//16
21.             __m512 sum13 = _mm512_setzero_ps();
22.             __m512 sum14 = _mm512_setzero_ps();
23.             __m512 sum15 = _mm512_setzero_ps();
24.
25.             for (int k = 0; k < cols1; k += 64) {
26.                 // Load columns of A
27.                 __m512 a0 = _mm512_loadu_ps(&A[i * cols1 + k]);
28.                 __m512 a1 = _mm512_loadu_ps(&A[i * cols1 + k + 16]);
29.                 __m512 a2 = _mm512_loadu_ps(&A[i * cols1 + k + 32]);
30.                 __m512 a3 = _mm512_loadu_ps(&A[i * cols1 + k + 48]);
31.
32.                 // Load columns of B
33.                 __m512 b0 = _mm512_loadu_ps(&B[k * cols2 + j]);
34.                 __m512 b1 = _mm512_loadu_ps(&B[k * cols2 + j + 16]);
35.                 __m512 b2 = _mm512_loadu_ps(&B[k * cols2 + j + 32]);
36.                 __m512 b3 = _mm512_loadu_ps(&B[k * cols2 + j + 48]);
37.
38.                 // Perform matrix multiplication
39.                 __m512 temp0 = _mm512_mul_ps(a0, b0);
40.                 __m512 temp1 = _mm512_mul_ps(a0, b1);
41.                 __m512 temp2 = _mm512_mul_ps(a0, b2);
```

```
                __m512 temp3 = _mm512_mul_ps(a0, b3);

                __m512 temp4 = _mm512_mul_ps(a1, b0);
                __m512 temp5 = _mm512_mul_ps(a1, b1);
                __m512 temp6 = _mm512_mul_ps(a1, b2);
                __m512 temp7 = _mm512_mul_ps(a1, b3);

                __m512 temp8 = _mm512_mul_ps(a2, b0);
                __m512 temp9 = _mm512_mul_ps(a2, b1);
                __m512 temp10 = _mm512_mul_ps(a2, b2);
                __m512 temp11 = _mm512_mul_ps(a2, b3);

                __m512 temp12 = _mm512_mul_ps(a3, b0);
                __m512 temp13 = _mm512_mul_ps(a3, b1);
                __m512 temp14 = _mm512_mul_ps(a3, b2);
                __m512 temp15 = _mm512_mul_ps(a3, b3);

                sum0 = _mm512_add_ps(sum0, temp0);
                sum1 = _mm512_add_ps(sum1, temp1);
                sum2 = _mm512_add_ps(sum2, temp2);
                sum3 = _mm512_add_ps(sum3, temp3);
                sum4 = _mm512_add_ps(sum4, temp4);
                sum5 = _mm512_add_ps(sum5, temp5);
                sum6 = _mm512_add_ps(sum6, temp6);
                sum7 = _mm512_add_ps(sum7, temp7);

                sum8 = _mm512_add_ps(sum8, temp8);
                sum9 = _mm512_add_ps(sum9, temp9);
                sum10 = _mm512_add_ps(sum10, temp10);
                sum11 = _mm512_add_ps(sum11, temp11);
                sum12 = _mm512_add_ps(sum12, temp12);
                sum13 = _mm512_add_ps(sum13, temp13);
                sum14 = _mm512_add_ps(sum14, temp14);
                sum15 = _mm512_add_ps(sum15, temp15);
            }

            // Store the results in C
            _mm512_storeu_ps(&C[i * cols2 + j], sum0);
            _mm512_storeu_ps(&C[i * cols2 + j + 16], sum1);
            _mm512_storeu_ps(&C[i * cols2 + j + 32], sum2);
            _mm512_storeu_ps(&C[i * cols2 + j + 48], sum3);
            _mm512_storeu_ps(&C[i * cols2 + j + 64], sum4);
            _mm512_storeu_ps(&C[i * cols2 + j + 80], sum5);
            _mm512_storeu_ps(&C[i * cols2 + j + 96], sum6);
            _mm512_storeu_ps(&C[i * cols2 + j + 112], sum7);

```

```
                _mm512_storeu_ps(&C[i * cols2 + j + 128], sum8);
                _mm512_storeu_ps(&C[i * cols2 + j + 144], sum9);
                _mm512_storeu_ps(&C[i * cols2 + j + 160], sum10);
                _mm512_storeu_ps(&C[i * cols2 + j + 176], sum11);
                _mm512_storeu_ps(&C[i * cols2 + j + 192], sum12);
                _mm512_storeu_ps(&C[i * cols2 + j + 208], sum13);
                _mm512_storeu_ps(&C[i * cols2 + j + 224], sum14);
                _mm512_storeu_ps(&C[i * cols2 + j + 240], sum15);
            }
        }
    }
```

5.5 击败 64K x 64K **Bill**

对于这种大 BOSS，我们肯定是要把计算-访存发挥到极致的。而其中我们要尽可能地将访存的成本尽可能减少。因此，我们可以采用分块的想法，给各个核进行分配，然后计算。



Loop 5 **for** $j_c = 0 : n-1$ **steps of** n_c
  $\mathcal{J}_c = j_c : j_c + n_c - 1$
Loop 4 **for** $p_c = 0 : k-1$ **steps of** k_c
    $\mathcal{P}_c = p_c : p_c + k_c - 1$
    $B(\mathcal{P}_c, \mathcal{J}_c) \rightarrow B_c$ // Pack into B_c
Loop 3 **for** $i_c = 0 : m-1$ **steps of** m_c
      $\mathcal{I}_c = i_c : i_c + m_c - 1$
      $A(\mathcal{I}_c, \mathcal{P}_c) \rightarrow A_c$ // Pack into A_c
      // Macro-kernel
Loop 2 **for** $j_r = 0 : n_c-1$ **steps of** n_r
        $\mathcal{J}_r = j_r : j_r + n_r - 1$
Loop 1 **for** $i_r = 0 : m_c-1$ **steps of** m_r
          $\mathcal{I}_r = i_r : i_r + m_r - 1$
          // Micro-kernel
Loop 0 **for** $k_r = 0 : k_c-1$
            $C_c(\mathcal{I}_r, \mathcal{J}_r)$
              $+= A_c(\mathcal{I}_r, k_r)\ \ B_c(k_r, \mathcal{J}_r)$
          endfor
        endfor
      endfor
    endfor
  endfor
endfor

这里有一个问题就是，我们应该用多大的 Macro Kernel 跟 Micro Kernel？我们当然可以自己拿参数进行测试跟调优。一种基于演化计算的自动调优方法由文章[4]给出。他们将 Kernel 的参数初始化种群，然后进行调优，最后得出他们集群的最佳参数：[792,822,8628]。

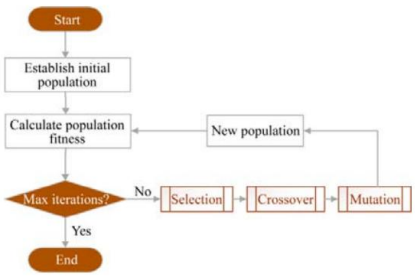


Fig.4 Generic algorithm flow chart
图 4 遗传算法流程图

for $j_c = 0, \ldots, n-1$ in steps of n_c 大kernel
  for $p_c = 0, \ldots, k-1$ in steps of k_c Outer-loops
    $B(p_c : p_c + k_c - 1, j_c : j_c + n_c - 1) \rightarrow B_c$
    for $i_c = 0, \ldots, m-1$ in steps of m_c
      $A(i_c : i_c + m_c - 1, p_c : p_c + k_c - 1) \rightarrow A_c$
      for $j_r = 0, \ldots, n_c-1$ in steps of n_r Inner-loops
        for $i_r = 0, \ldots, m_c-1$ in steps of m_r
          for $p_r = 0, \ldots, k_c-1$ in steps of 1
            $Cc(i_r : i_r + m_r - 1, j_r : j_r + n_r - 1)$
            $+= A_c(i_r : i_r + m_r - 1, p_r)$
            $\cdot\ B_c(p_r, j_r : j_r + n_r - 1)$

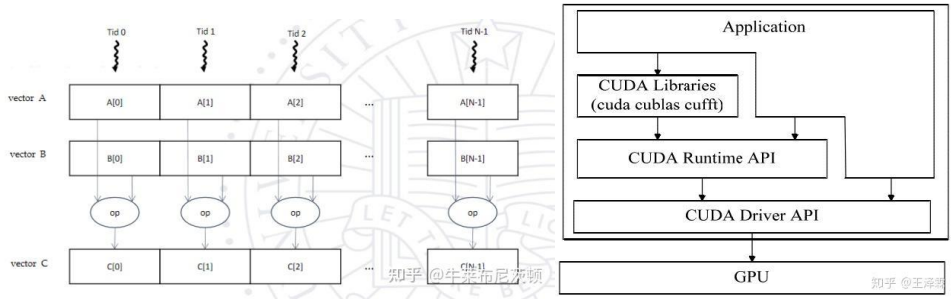
for $j = 0, \ldots, n-1$ in steps of n_r 小kernel
  for $p = 0, \ldots, k-1$ in steps of 1 Packing B
    $B(p, j : j + n_r - 1) \rightarrow B_{packed}$
for $j_r = 0, \ldots, n-1$ in steps of n_r Inner-loops
  for $i_r = 0, \ldots, m-1$ in steps of m_r
    for $p_r = 0, \ldots, k-1$ in steps of 1
      $C(i_r : i_r + m_r - 1, j_r : j_r + n_r - 1)$
      $+= A(i_r : i_r + m_r - 1, p_r)$
      $\cdot\ B_{packed}(p_r, j_r : j_r + n_r - 1)$

Fig.3 High performance GEMM algorithm of big and little kernels
图 3 高性能 GEMM 的大 kernel 算法和小 kernel 算法

```
#define BLOCK_SIZE 256

void gemm_avx512(int M, int N, int K, float *A,  float *B,float *C) {
```

```
        // 遍历矩阵C的每个块
        #pragma omp parallel for
        for (int j = 0; j < M; j += BLOCK_SIZE) {
            for (int i = 0; i < N; i += BLOCK_SIZE) {
                for (int k = 0; k< K; k += BLOCK_SIZE) {
                    // big kernel
                    for (int jj = j; jj < (j + BLOCK_SIZE); jj += BLOCK_SIZE) {
                        for (int ii = i; ii < (i + BLOCK_SIZE); ii += BLOCK_SIZE) {
                            for(int kk = k; kk < k + BLOCK_SIZE; kk += BLOCK_SIZE){
                                // small kernel
                                SGEMM_AVX512_256_kernel(BLOCK_SIZE,BLOCK_SIZE,BLOCK_SIZE,
                                A + ii + kk * M, B + kk + jj * N, C + ii + jj * K );
                            }
                        }
                    }
                }
            }
        }
    }
```

当然，如果我们有 CUDA，当然是用 GPU 更好。对于 CUDA，我们可以编辑一个 CUDA 的核函数，然后进行乘法。GPU 使用的是 SIMT，单指令多线程。面对一条指令，GPU 将启动多个我们之前提到的帕鲁 SM，开启多个线程进行计算。跟 SIMD 相比，SIMT 能够在线程数量很多的时候表现的更好一些。回想一下我们刚刚做那些循环展开和指令重排的工作，其实本质上就是希望同时有更多的 CPU 能执行我们的指令。而 SIMT 就只需要考虑接收指令，开启线程即可，中间很多的指令不仅可以优化，还可以加速到最优点。[23]




```
    // CUDA kernel function for matrix multiplication
    __global__ void matrixMultiply(const float* A, const float* B, float* C, int width) {
        int row = blockIdx.y * blockDim.y + threadIdx.y;
        int col = blockIdx.x * blockDim.x + threadIdx.x;

        float Cvalue = 0.0f;
        for (int k = 0; k < width; ++k) {
            Cvalue += A[row * width + k] * B[k * width + col];
        }
        C[row * width + col] = Cvalue;
    }
```

[23] 在《CUDA C Programming Guide》第 135 页对 SIMT 有详细的介绍。

另外值得注意的是，这里如果就这么简单的像我这样写，是无法发挥 GPU 的性能的，我们还需要进行不断的拓展，在《CUDA C Programming Guide》里就有很好的案例[24]。其实优化的目标和方向基本是一致：利用 Shared Memory 降低访存延迟、循环展开、向量读取指令等等。说到头来，还是“计算-访存”优化。另外，还有一个要注意的点是我们的 GPU Kernel 函数要跟我们的 GPU 相对应。比如 A100 的 108 个 SM 跟我们 2080TI 的 68 个 SM 就必须在线程模型上进行区分。同时 SM 内部 FP16、FP32、FP8 模块的数量都有所不同，如果想极致优化必须考虑到这一点。

更多在异构系统上的优化由文章[3][5][8]给出，他们在我们简单的单节点单 GPU 的基础上，对多 GPU 多节点进行了更深入的优化。比如文章[3]完成了一个轻量级跨平台异构加速框架 HPCX，超越了 NVIDIA 官方的 HPL 基准测试（但是 71.1%的浮点效率其实有点低了，去年我们队在启明上的测试结果在 73%左右，可能是文章时间比较久了，现在 NVIDIA 24.03 版本的 HPL 可能更强大了）他们的其中一个优化方法计算将访存均匀分布，确保 GPU 的取存**“隐藏”在计算中**。

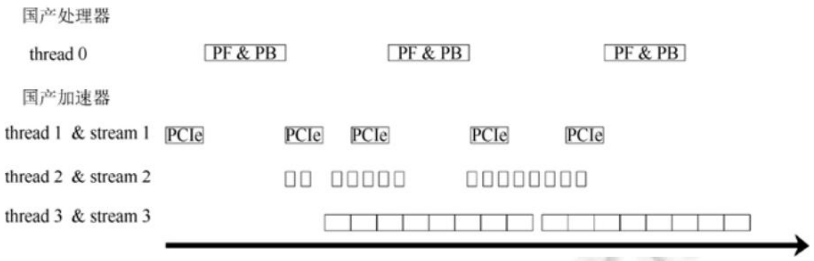


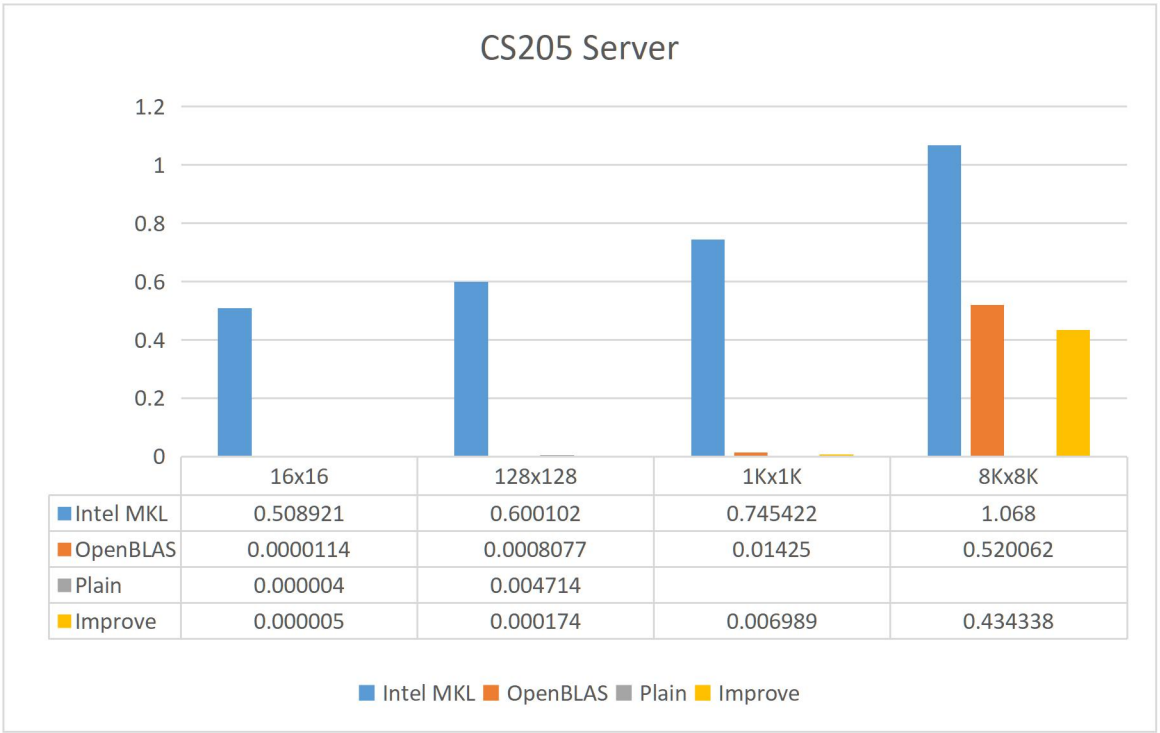
Fig.3 HPL multithread fine-grained parallel algorithm flow

图 3 HPL 多线程细粒度并行算法流程图

Part 6: 性能测试 **Chapter 6: DJANGO UNCHAINED**

我们在 CS205 Server 跟启明上测试。CS205 Server 结果如下。我们的速度基本上都在跟 OpenBLAS 竞争。64K * 64K 的矩阵测试不好做，因此没有进行。

在集群上的测试会受到这么两个干扰：虚拟机和他人的程序。虚拟机的策略在于，尽可能地将程序分配在靠近的节点上。但是就算是这样，虚拟机还是带来了一部分的性能损耗。



[24] 想快速学习，我们可以查看这篇文章：https://zhuanlan.zhihu.com/p/410278370

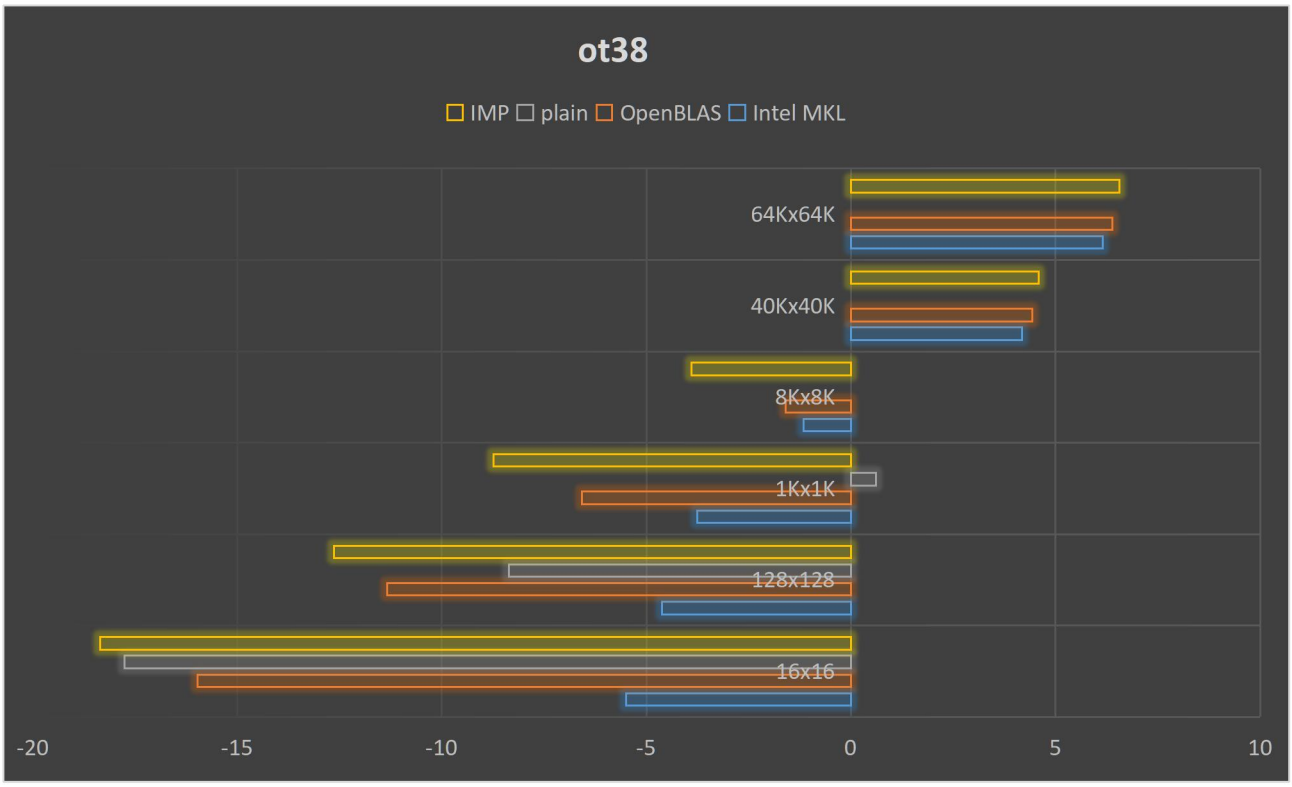
我们可以用 numactl -c 命令查看 NUMA 节点的结构：

```
available: 2 nodes (0-1)
node 0 cpus: 0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29 30 31
node 0 size: 257332 MB
node 0 free: 245606 MB
node 1 cpus: 32 33 34 35 36 37 38 39 40 41 42 43 44 45 46 47 48 49 50 51 52 53 54 55 56 57 58 59 60 61 62 63
node 1 size: 258038 MB
node 1 free: 252969 MB
node distances:
node   0   1
  0:  10  20
  1:  20  10
```

其实通过这个 NUMA，我们是不是可以在分块的时候也可以先分成 A、B 两块，然后再接着计算呢？我觉得这是个不错的想法。不过，这样仅仅适用于占满节点的情况，它还是缺少普适性。

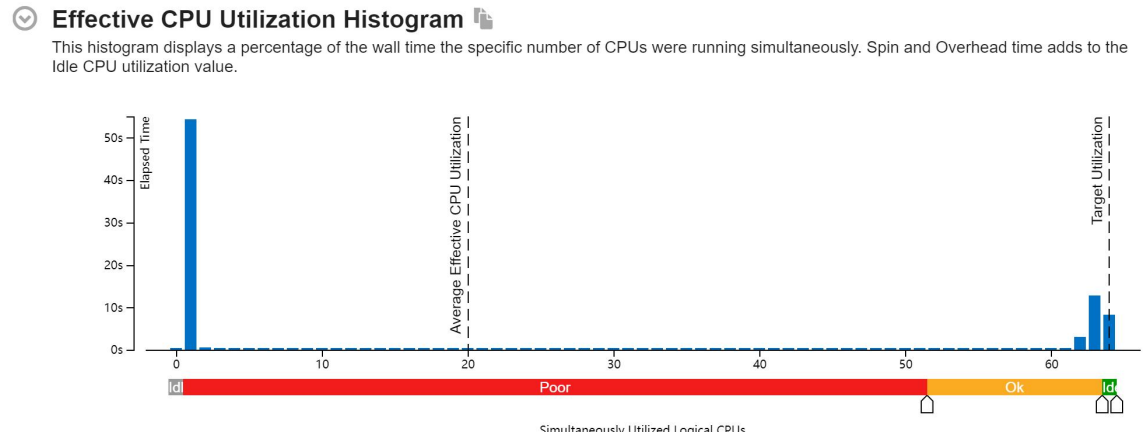
我们也在 38 队列上进行了测试。测试结果如下：

Problem Size	OpenBLAS	Intel MKL	plain	IMP
16x16	0.0000155	0.0220675	4.50E-06	0.000003
128x128	0.000387667	0.0405615	0.0030525	0.000157
1Kx1K	0.0104858	0.073802	1.52862	0.002346
8Kx8K	0.330365	0.450339		0.066904
40Kx40K	21.593331	18.137117		24.063801
64Kx64K	83.4281	71.3537		97.4437



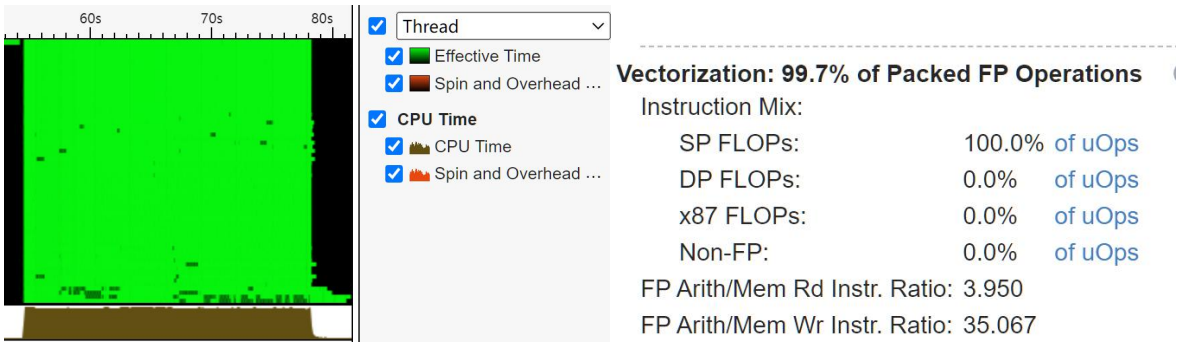
可以看出，在问题规模小于 8000 时，我们的速度基本上是领先于 MKL 和 OpenBLAS 的（没有进行 JIT 测试），不过在规模达到 40K 之后，我们的矩阵运行速度就开始减慢，OpenBLAS 超越了我们（可能是由于）我们的运行速度基本上在他们下面。不过令人鼓舞的是，OpenBLAS 的 83 秒/21.59 秒约等于 3.863，MKL 的 71 秒/18 秒约等于 3.934，我们的大概是在 4.05 左右，这样的增长速度还能接受。

我们的程序在并行化上还算不错，除了开始的随机化矩阵以外，剩下的基本都是在 62-64 核。

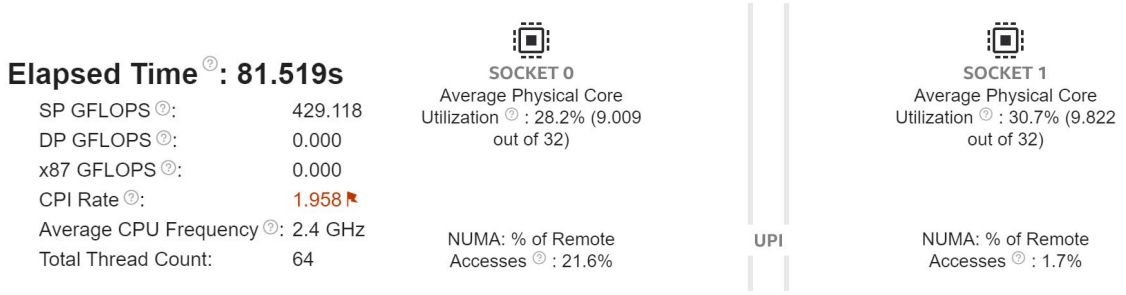


非常不幸的是，我们的程序在变成汇编指令之后，不能像 OpenBLAS 一样做到**访存指令跟计算指令交替进行**。如果做到了，我认为可能我们会有更强大的进步。另外还有一个点是，我在想可能对于 64 核的节点，我们的核函数还是有点小了。我们应该**做更多的循环展开**，就像我们之前看到 OpenBLAS 的几千行 AVX512 命令一样。

0x400f40		**Block 10:**		
0x400f40		vmovupsz 0x80(%rax), %zmm1	0.2%	2.748s
0x400f47		vmovupsz (%rax), %zmm5	0.1%	2.140s
0x400f4d		add $0x100, %rax	0.5%	8.040s
0x400f53		vmovupsz -0x40(%rax), %zmm0	0.0%	0.010s
0x400f5a		vmovupsz (%rdx), %zmm3	0.5%	7.970s
0x400f60		vmovupsz -0xc0(%rax), %zmm4	10.8%	172.943s
0x400f67		vmovupsz 0x40(%rdx), %zmm2	0.8%	12.330s
0x400f6e		add %rdi, %rdx	73.4%	1174.911s
0x400f71		vfmadd231ps %zmm1, %zmm5, %zmm18	0.1%	1.190s
0x400f77		vfmadd231ps %zmm3, %zmm5, %zmm20	0.1%	2.020s
0x400f7d		vfmadd231ps %zmm0, %zmm5, %zmm17	0.9%	14.500s
0x400f83		vfmadd231ps %zmm2, %zmm5, %zmm19	0.6%	9.240s
0x400f89		vfmadd231ps %zmm3, %zmm4, %zmm16	3.6%	58.096s
0x400f8f		vfmadd231ps %zmm2, %zmm4, %zmm15	0.1%	2.230s
0x400f95		vfmadd231ps %zmm1, %zmm4, %zmm14	1.0%	15.607s
0x400f9b		vfmadd231ps %zmm0, %zmm4, %zmm13	0.0%	0.050s
0x400fa1		vfmadd231ps %zmm3, %zmm1, %zmm12	0.3%	4.470s
0x400fa7		vfmadd231ps %zmm2, %zmm1, %zmm11	0.1%	1.420s
0x400fad		vfmadd231ps %zmm1, %zmm1, %zmm10	0.7%	10.960s
0x400fb3		vfmadd231ps %zmm0, %zmm1, %zmm6	0.0%	0.020s
0x400fb9		vfmadd231ps %zmm3, %zmm0, %zmm9	0.4%	6.300s
0x400fbf		vfmadd231ps %zmm2, %zmm0, %zmm8	0.1%	1.280s
0x400fc5		vfmadd231ps %zmm0, %zmm0, %zmm7	0.7%	11.720s
0x400fcb		cmp %rax, %rsi	0.0%	0.030s
0x400fce		jnz 0x400f40 <Block 10>		



看着还不错的是，我们的 CPU 基本上的吃满了，只有在少数的情况下中间才出现 Spin time。我们程序的 CPI 非常高，达到了 1.96，这应该是 SIMD 指令导致的。在超线程技术的影响下，普通的 load 指令 CPI 大概在 0.5 左右（Project2），在 AVX512 指令之后，我们的 16 条 float 一次导入，理论性能的加速可能在 4 倍左右。另外可以看到的是，我们的“计算-访存”的比例在 3.95，也就是 4 左右，这里边如果对比 OpenBLAS 的 2:1，我觉得还是有优化的方向，减少计算的指令，多访存，把存储的效率提高。



还有一个很糟糕的事情是，**我们在跑的时候是跨了 NUMA 节点**的！每个节点各 10 个核，我觉得对此非常的惋惜，如果我就用了一个 SOCKET，可能说不定就跟 OpenBLAS 一样快了，我觉得这也非常的离谱。不过我根据系统平台，我推测产生这样的原因主要是 38 队列的服务器有 64 核，我只申请了 24 核，而 lsf 作业系统就给我跨了 NUMA 节点。可恶，一定是 IBM 没有看过那篇北大的《虚拟化数据中心资源调度研究》文章！

我们还可以做更多的优化。比如对于 Linux 操作系统，我们可能可以换成 Rocky Linux。他们对线程以及性能的管控比 Ubuntu 更加高效。我们可以尝试关闭虚拟化等等。当然大部分软件系统架构的改变可遇不可求，在这里也就是简单提提。

我们还可以做更多的分析，比如我们的核时/理论核时，我们的指令数/理想指令数，我们的花费的 CPU 时间/理论 CPU 时间等等，来比较我们的访存速度，计算速度，计算开销，得到一个更加全面完善的评估结果。

近年来这些测试还是挺多的。大家对于数据中心好像有了在高性能、高扩展之外，还多了一个节能的需求。听说现在美国数据中心合计功率 16.82GW，用电量约 192TWh，占其全社会用电量近 5%，到 2023 年，计算中心的电力需求将拉动美国全社会用电量增长 10.9%-11.9%（以 2023 年基准）[25]。所以大家发现，我们不仅要算的快，还要算的便宜，算的高效。

本来我还想把我的程序改成 ZGEMM 的，计算虚数，然后来加速材料计算的软件。这个软件用于计算石墨烯等材料的朗道能级，它是用 Fortran 写的，我前两周自己把 Fortran 学完了，结果没时间改了。我准备后面继续改好我的 API，这样就能套上去了。

在 ASC 比赛的时候，北京大学的队伍就是优化了下面这个 api：call ZGEMM，然后把这个 api 套到了他们自己的 GPU BLAS 上。我是当时负责这道赛题的同学。我今天回来看，我觉得非常惭愧。我没有在第一时间内对软件进行性能分析，找到软件这里的关键耗时点。同时，北大他们写了一个自研 API，就不同的矩阵大小，把计算分配到 OpenBLAS、MKL、cuBLAS 上。我觉得我们也要有一个，不然我们永远被人家压着打。

不过，这些故事，就可能要慢慢讲了……

```
! performs matrix-matrix multiply
! C=A*B
  subroutine mat_mul(nmatdim,A,B,C)

     use para, only : Dp
      implicit none


     integer,intent(in) :: nmatdim

     complex(Dp) :: ALPHA
     complex(Dp) :: BETA

     complex(Dp), intent(in)  :: A(nmatdim ,nmatdim)
     complex(Dp), intent(in)  :: B(nmatdim ,nmatdim)
     complex(Dp), intent(out) :: C(nmatdim,nmatdim)

     ALPHA=1.0d0
     BETA=0.0D0

     C(:,:)=(0.0d0,0.0d0)

     call ZGEMM('N','N',nmatdim,nmatdim,nmatdim,ALPHA, &
                &  A,nmatdim,B,nmatdim,BETA,C,nmatdim)

     return
  end subroutine mat_mul
```

[25] 相关文章介绍：https://xueqiu.com/4545671674/287082337
市场规模分析：https://www.mordorintelligence.com/zh-CN/industry-reports/north-america-data-center-power-market-industry

图中中间偏右的这位老先生就是大名鼎鼎的 Jack Dongarra，高性能计算 HPC 领域的开创者。他参与的一系列线性代数库软件项目，MPI，BLAS，LINPACK，LAPACK，成为了今天高性能数值计算的基石。而数值计算中非常重要的一环，就是 GEMM。

Jack Dongarra 在他的博士阶段写出了 HPL。HPL（The High-Performance Linpack Benchmark）是测试高性能计算集群系统浮点性能的基准程序。HPL 通过对高性能计算集群采用高斯消元法求解一元 N 次稠密线性代数方程组的测试，评价高性能计算集群的浮点计算能力。目前世界前 500 超算集群榜单（Top500 list），就是按照 HPL 的结果进行的测试。

浮点计算峰值是指计算机每秒可以完成的浮点计算次数，包括理论浮点峰值和实测浮点峰值。理论浮点峰值是该计算机理论上每秒可以完成的浮点计算次数，主要由 CPU 的主频决定。理论浮点峰值＝CPU 主频×CPU 核数×CPU 每周期执行浮点运算的次数。

Jack 的文章[1]讲述的就是 HPL 的历史发展，以及未来展望。近年来 HPL 逐渐随着 GPU 的发展而走向异构系统化，在 GPU 加速方面的研究逐渐变得越来越重要。当然在小规模矩阵上的处理，CPU 仍然以通讯优势占据主导。但是随着 NPU、DCU 等针对性异构系统的出现，我认为计算和计算机体系结构会出现更多样的变化。

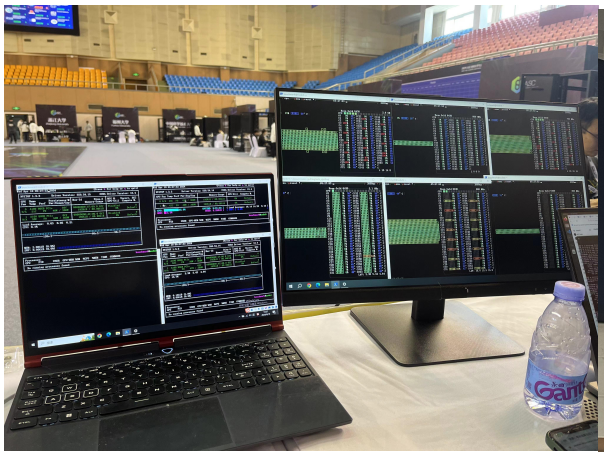
文章[6]对许多高性能计算应用软件进行了介绍，同时对我国高性能计算的发展进行了回顾与展望。

Jack 也是最早的一批确定 BLAS 数学库的先驱者。BLAS（basic linear algebra subroutine） 是一系列基本线性代数运算函数的接口标准． 这里的线性代数运算是指例如矢量的线性组合。BLAS 被广泛用于科学计算和工业界。在更高级的语言和库中，即使我们不直接使用 BLAS 接口，它们也是通过调用 BLAS 来实现的（如 Matlab 中的各种矩阵运算）。

Jack 的实验室写出了第一个 BLAS：Lapack。它是用该死的 Fortran 写的，但是如今我们可以看到这玩意还活跃于 MKL 等等当中。

Jack 还是最早定义了 MPI 标准的一群人，他对于整个高性能计算领域是非常伟大的贡献，另一个伟大贡献人物是发明超算机的 Cray（当然还有一个伟大的人物是李卓钊老师）。Jack Dongarra 由于他的突出贡献，在 2021 年获得了计算机图灵奖。文章[20]讲述了他非凡的生平。

一些看了很想哭，想笑，想回忆的事情：
ASC22-23 回忆录 - Aethoce's Blog



ASC 的现场，测量 HPL

对功耗/Flops 的测量是一个重要指标

大家在现场调试集群，还是出现了很多没想过的 bug[26]

最后很多没有优化，来年看学弟学妹！[27]

最后的晚餐：终于把中大的方案套到手了哈哈

跟周老师的合影 + 欢迎报考上海科技大学！

[26] https://mp.weixin.qq.com/s/s4GBsmvXmFVlrrWsXra3nA
[27] 南科大超算俱乐部交流群：897073438，all are welcome!

我们对矩阵乘法以及 OpenBLAS、MKL 进行了一定的测量和研究。我们造出了自己的 SGEMM 算法，并在一定程度上接近于这两个数学库。但是数学库的庞大让我们明白，我们还有很长很多的路要走。同样的，对底层的学习，对新 GPU 的学习，我还要走很远。CPU 跟 GPU 都是一群无耻混蛋，对此我们必须喊出我们队的口号：

“任何算力，终将被绳之以法！”

[28]

[28] 完蛋！我被超算包围了！——ASC24 南科大超算队_哔哩哔哩_bilibili

Reference

[1] Dongarra, J.J., Luszczek, P., & Petitet, A. (2003). The LINPACK Benchmark: past, present and future. Concurrency and Computation: Practice and Experience, 15. Available: https://netlib.org/utk/people/JackDongarra/PAPERS/hplpaper.pdf

[2] 黎雷生, 杨文浩, 马文静, 张娅, 赵慧, 赵海涛, 李会元, 孙家昶. 复杂异构计算系统 HPL 的优化. 软件学报, 2021, 32(8): 2307-2318. http://www.jos.org.cn/1000-9825/6003.htm

[3] 水超洋, 于献智, 王银山, 谭光明. 国产异构系统上 HPL 的优化与分析. 软件学报, 2021, 32(8): 2319-2328. http://www.jos.org.cn/1000-9825/6004.htm

[4] 蔡雨, 孙成国, 杜朝晖, 刘子行, 康梦博, 李双双. 异构 HPL 算法中 CPU 端高性能 BLAS 库优化. 软件学报, 2021, 32(8): 2289-2306. http://www.jos.org.cn/1000-9825/6002.htm

[5] 孙乔,孙家昶,马文静,赵玉文.面向异构计算机平台的 HPL 方案.软件学报,2021,32(8):2329-2340

[6] 徐顺, 王武, 张鉴, 姜金荣, 金钟, 迟学斌. 面向异构计算的高性能计算算法与软件. 软件学报, 2021, 32(8): 2365-2376. http://www.jos.org.cn/1000-9825/6008.htm

[7] Ma ZX, Jin YY, Tang SZet al. Unified programming models for heterogeneous high-performance computers. JOURNAL OF COMPUTER SCIENCE AND TECHNOLOGY 38(1): 211−218 Jan. 2023. DOI: 10.1007/s11390-023-2888-4

[8] 刘芳芳, 王志军, 汪荃, 吴丽鑫, 马文静, 杨超, 孙家昶. 国产异构系统上的 HPCG 并行算法及高效实现. 软件学报, 2021, 32(8): 2341-2351. http://www.jos.org.cn/1000-9825/6006.htm

[9] R. C. Whaley and J. J. Dongarra, "Automatically Tuned Linear Algebra Software," SC '98: Proceedings of the 1998 ACM/IEEE Conference on Supercomputing, Orlando, FL, USA, 1998, pp. 38-38, doi: 10.1109/SC.1998.10004.

[10] N. Chalmers, J. Kurzak, D. McDougall and P. T. Bauman, "Optimizing High-Performance Linpack for Exascale Accelerated Architectures," SC23: International Conference for High Performance Computing, Networking, Storage and Analysis, Denver, CO, USA, 2023, pp. 1-12, doi: 10.1145/3581784.3607066.

[11] NVIDIA A100 Tensor Core GPU Architecture 架构白皮书, Available: https://images.nvidia.cn/aem-dam/en-zz/Solutions/data-center/nvidia-ampere-architecture-whitepaper.pdf

[12] NVIDIA Geforce RTX Turing 架构白皮书, Available: https://www.nvidia.cn/geforce/news/geforce-rtx-20-series-turing-architecture-whitepaper/

[13] OpenMP 4.5 API C/C++ Syntax Reference Guide, Available: https://www.openmp.org/wp-content/uploads/OpenMP-4.5-1115-CPP-web.pdf

[14] How to optimize DGEMM on x86 CPU platforms, Available: https://github.com/yzhaiustc/Optimizing-DGEMM-on-Intel-CPUs-with-AVX512F

[15] Intel® Math Kernel Library Improved Small Matrix Performance Using Just-in-Time (JIT) Code Generation for Matrix Multiplication (GEMM), Available: https://www.intel.com/content/www/us/en/developer/articles/technical/onemkl-improved-small-matrix-performance-using-just-in-time-jit-code.html

[16] 齐鲁工业大学 ASC24 OpenCAEPoro 团队&个人优化方案. 黄皮耗子 blog: https://blog.sethome.cc/post/asc24-opencaeporo-ge-ren-andtuan-dui-you-hua-fang-an/

[17] J. Choquette, W. Gandhi, O. Giroux, N. Stam and R. Krashinsky, "NVIDIA A100 Tensor Core GPU: Performance and Innovation," in IEEE Micro, vol. 41, no. 2, pp. 29-35, 1 March-April 2021, doi: 10.1109/MM.2021.3061394.

[18] Abstract: NVIDIA A100 Tensor Core GPU is NVIDIA's latest flagship GPU.URL: https://ieeexplore.ieee.org/stamp/stamp.jsp?tp=&arnumber=9361255&isnumber=9388768

[19] 國立清華大學開放式課程 10701 平行程式 周志遠 教授 Available: https://ocw.nthu.edu.tw/ocw/index.php?page=chapter&cid=231&chid=2574&video_url=https%3A%2F%2Focw.nthu.edu.tw%2Fvideosite%2Fi

ndex.php%3Fop%3Dwatch%26id%3D7625%26filename%3D1920_1080_3072.MP4%26type%3Dview%26cid%3D231%26chid%3D2574&name=L1A

[20] The Influence and Contribution of Jack Dongarra to Numerical Linear Algebra https://hammarling.com/sven/pubs/HaHi22_Final.pdf

[21] J. Langou, J. Langou, P. Luszczek, J. Kurzak, A. Buttari and J. Dongarra, "Exploiting the Performance of 32 bit Floating Point Arithmetic in Obtaining 64 bit Accuracy (Revisiting Iterative Refinement for Linear Systems)," SC '06: Proceedings of the 2006 ACM/IEEE Conference on Supercomputing, Tampa, FL, USA, 2006, pp. 50-50, doi: 10.1109/SC.2006.30.

[22] A. Abdelfattah, M. Baboulin, V. Dobrev, J. Dongarra, C. Earl, J. Falcou, A. Haidar, I. Karlin, Tz. Kolev, I. Masliah, S. Tomov, High-performance Tensor Contractions for GPUs, Procedia Computer Science, Volume 80, 2016, Pages 108-118, ISSN 1877-0509, https://doi.org/10.1016/j.procs.2016.05.302.

[23] J. Dongarra, High-Performance Matrix-Matrix Multiplications of Very Small Matrices, https://netlib.org/utk/people/JackDongarra//PAPERS/high-performance-matrix-matrix.pdf

[24] 南科大科学与工程计算中心超级计算机用户手册(启明 2.0 和太乙) https://hpc.sustech.edu.cn/ref/cluster_User_Manual.pdf

[25] 2023 年南科大科学与工程计算中心年报 https://hpc.sustech.edu.cn/ref/2023CCSEAnnualReport.pdf

[26] BLAS Quick Reference Guide, Available: https://www.netlib.org/blas/blasqr.pdf

[27] Brehob, Mark & Enbody, R.J.. An analytical model of locality and caching. https://www.researchgate.net/publication/228814140_An_analytical_model_of_locality_and_caching/figures?lo=1&utm_source=bing&utm_medium=organic

[28] 刘昊,刘芳芳,张鹏,杨超,蒋丽娟.基于申威 1600 的 3 级 BLAS GEMM 函数优化.计算机系统应用,2016,25(12):234-239 http://www.c-s-a.org.cn/csa/article/pdf/20161237

[29] 殷顺昌. OpenMP 并行程序性能分析[D].国防科学技术大学,2007.

[30] 高雨辰. 面向国产处理器的 OpenMP 程序编译优化技术研究[D].战略支援部队信息工程大学,2019.

[31] Smith, T.M., & van de Geijn, R.A. (2017). A Tight I/O Lower Bound for Matrix Multiplication. ACM Transactions on Mathematical Software (TOMS). https://arxiv.org/pdf/1702.02017