中山大学数据科学与计算机院本科生实验报告

(2020 学年秋季学期)

课程名称: 高性能计算程序设计　　任课教师: 黄聃　　批改人:

年级 + 班级	18 计科 8 班	专业（方向）	超算
学号	18340208	姓名	张洪宾
Email	2285075600@qq.com	完成时间	2020 年 11 月 7 号

目录

1 实验目的

- 熟悉 OpenMP 编程，用来优化矩阵乘法，争取较高的加速比。
- 在 static 和 dynamic 两种循环调度方式下实现对 openMP 的并行。
- 构造基于 Pthreads 的并行 for 循环分解、分配和执行机制。

2 实验过程及核心代码

2.1 实验环境

在这里我因为某些原因暂时申请到了一个 KNL 集群的节点，性能相比我的电脑来说强很多，并且可以有更多的优化选择，正好老师上一节课讲过 KNL 集群，因此我在这里就使用该节点来完成此次实验。

2.1.1 硬件

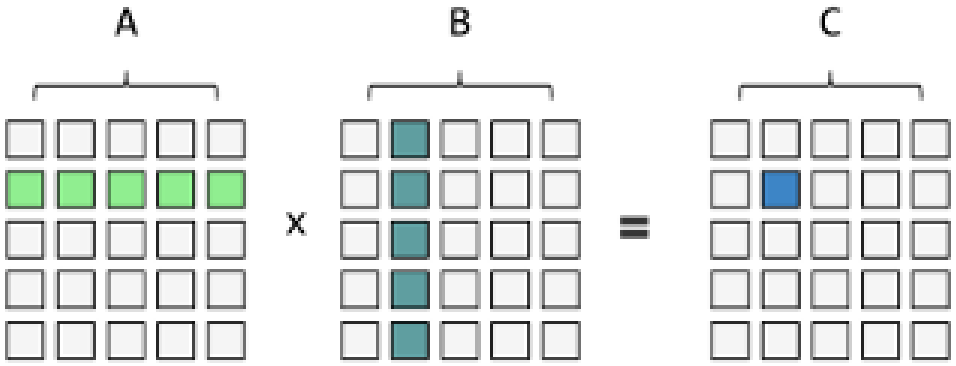
- CPU:Intel(R) Xeon Phi(TM) CPU 7210 @ 1.30GHz 64 核
- 16GB 的 MCDRAM 和 96G 的 DDR4 RAM

2.1.2 软件

- 操作系统：CentOS Linux release 7.8.2003 (Core)
- Intel Parallel Studio XE 2019
- gcc version 8.4.0

2.2 串行版本的 GEMM

GEMM 的原理如下：



即：

$$c_{ix} = \sum_{j=1}^{n} a_{ij} b_{jx}$$

为了显示出优化后的效果，计算出加速比，我们需要写一个串行版本的 GEMM 来做比较。在这里我用之前的代码稍微修改，得到了如下的串行的 GEMM 函数：

```
void Serial(){
    for(int i = 0;i < m;i++){
        for(int j = 0;j < n;j++){
            for(int x = 0;x < k;x++){
                res[i][x] += Matrix_a[i][j] * Matrix_b[j][x];
            }
        }
    }
}
```

再使用之前的 main 函数读取问题规模 m、n、k 并生成随机数，就可以调用该函数进行计算了。

在这里的 Makefile 我使用了 gcc 编译器，不采用任何优化，便于之后计算加速比来查看优化的效果。

2.3 通过 OpenMP 实现通用矩阵乘法

openMP 实现并行很简单，只需要非常无脑地在最外层循环前加上 #pragma omp parallel for 就行了。经过测试这是效果最高的并行的方法，原因如下：

因为所有线程最终都是要映射到多核心上，在这个问题中，如果我们划分最外层循环，内层的循环还是在线程内部串行执行，这样子内部循环的局部性就不会被破坏。如果我们划分的是内层循环的

话，虽然每个线程执行的循环总数还是不变，但是可能会因为内部线程所执行程序的局部性被破坏，造成性能严重下降。

在 Intel 官网中OpenMP* 循环调度中提到，默认状况下，如果循环数为 n, 可以看作采用块大小为 $\frac{n}{thread\_count}$ 。所以问题被如下划分：

在默认的调度方式下，会将矩阵 A 分成 thread_count 块，然后结果矩阵也会对应地被划分为 thread_count 块。具体的代码如下：

```
void parallel_calculate(){
    #pragma omp parallel for num_threads(64)
    for(int i = 0;i < m;i++){
        for(int j = 0;j < n;j++){
            for(int x = 0;x < k;x++){
                res[i][x] += Matrix_a[i][j] * Matrix_b[j][x];
            }
        }
    }
}
```

编写 Makefile，用 gcc 编译该程序的时候开启 O3 优化，让运行变得更加快速。

2.4 基于 OpenMP 的通用矩阵乘法优化

2.4.1 使用 Intel Parallel Studio XE 来编译并进行优化

在使用三种不同的调度方式来优化矩阵乘法之前，我先针对运行环境的特点，对已有的代码做了一些优化，具体的优化方式如下：

看起来开启 O3 优化已经可以非常强力地加快运行，但是还有更多的方式可以改进运行的速度。结合老师上课所讲的 KNL 集群的内容，以及在课余学习的内容，在这里我改用了 Intel 编译器 Intel

Parallel Studio XE，因为是 Intel 自己的编译器，在这种众核处理器下优化的效果非常明显。在这里我在 Intel 官网找到了一篇英特尔 ® 至强融核 ™ 处理器优化教程，较贴切地讲述了在 KNL 集群上优化 OpenMP 程序的方法。

在这里的优化有如下几个方面：

实现代码自动矢量化

在这里我使用了 #pragma vector alway 来指示编译器忽略其他因素，进行向量化，用 #pragma ivdep 来告诉编译器我们的问题之间没有循环依赖关系。

然后在 Makefile 的 CFLAGS 中加入-O3 激活 O3 优化，借助自动矢量化，编译器将 16 个单精度浮点数打包在矢量寄存器中并对该矢量执行运算，而不是在每个迭代循环中一次处理一个元素。相当于在这里开启了 AVX 指令集。

然后再加入–xMIC-AVX512，来充分利用 AVX-512 指令集。然后再在编译参数中加入如下参数： -fp-model fast=2：允许丢精度的浮点数优化，然后我在 Intel 的手册上看到一个参数-par-affinity=compact，用来控制线程亲和性。还要记得加速-qopenmp 来支持 OpenMP。

最终的程序如下：

```
void parallel_calculate(){
    #pragma omp parallel for num_threads(64)
    for(int i = 0;i < m;i++){
                #pragma ivdep
                #pragma vector always
        for(int j = 0;j < n;j++){
            for(int x = 0;x < k;x++){
                res[i][x] += Matrix_a[i][j] * Matrix_b[j][x];
            }
        }
    }
}
```

Makefile 如下：

```
CC=icc
CFLAGS=-std=c++11 -O3 -w -Ofast -qopenmp -fp-model fast=2 -par-affinity=
    compact -Iinclude/
SRCS = $(wildcard *.c)
```

```
TARGET = openmp_matrix_multiply
.PHONY: all clean
all: $(TARGET)
$(TARGET): $(SRCS)
	$(CC) -o $@ $^ $(CFLAGS)
clean:
	rm $(TARGET)
```

使用 numactl 获取 numa 最优性能

如果 MCDRAM 配置为扁平或混合模式，英特尔至强融核处理器将以 2 个 NUMA 节点的形式出现。numa 节点的情况如下：

```
[u18340208@knl04 ~]$ numactl -H
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 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 0 size: 96415 MB
node 0 free: 94331 MB
node 1 cpus:
node 1 size: 16127 MB
node 1 free: 15900 MB
node distances:
node   0   1
  0:  10  31
  1:  31  10
```

所以我们可以看到我们的配置确实是 KNL 的扁平模式。在教程中也提到：如果 MCDRAM 配置为可寻址内存（扁平模式），用户可明确地分配 MCDRAM 中的内存。

由于 MCDRAM 带宽约为 500 GB/秒，而 DDR4 峰值性能带宽约为 90 GB/秒，所以在这里我们可以选择迫使程序将内存分配至 MCDRAM 所在 NUMA 节点。所以我们可以执行如下命令：

numactl -m 1 ./openmp_matrix_multiply

这样子执行的时候会使用 MCDRAM。

2.4.2 采用不同的循环调度方式来加速运行

在这里我将使用三种循环调度的方式，默认的调度方式，静态调度 schedule(static, 1) 和动态调度 schedule(dynamic,1) 三种方式。

首先是默认的调度方式，其代码就是上面经过我优化后的代码，不需要添加任何的子句。这种调度方式下，OpenMP 线程会使用块划分：假设串行循环中有 n 次迭代，那么在并行循环中，前

n/thread_count 个迭代分配给线程 0，接下来的 n/thread_count 个迭代分配给线程 1，以此类推。

OpenMP 提供来 schedule 子句来让我们指定循环划分的方式。当指定为 static 的时候，进行静态划分，并且可以指定块大小 chunksize，在这里 chunksize = 1，也就是说，对于第 i 个迭代，它会分配给线程 id 为 $i\%threads\_count$ 的线程。

这部分代码如下：

```
void parallel_calculate(){
    #pragma omp parallel for num_threads(64) schedule(static,1)
    for(int i = 0;i < m;i++){
        #pragma ivdep
        #pragma vector aligned
        for(int j = 0;j < n;j++){
            for(int x = 0;x < k;x++){
                res[i][x] += Matrix_a[i][j] * Matrix_b[j][x];
            }
        }
    }
}
```

如果采用 dynamic 的方式调度，划分块的时候也是类似的方法，不过调度的时候是根据运行状况进行动态调度。

代码如下：

```
void parallel_calculate(){
    #pragma omp parallel for num_threads(64) schedule(dynamic,1)
    for(int i = 0;i < m;i++){
        #pragma ivdep
        #pragma vector aligned
        for(int j = 0;j < n;j++){
            for(int x = 0;x < k;x++){
                res[i][x] += Matrix_a[i][j] * Matrix_b[j][x];
            }
        }
    }
}
```

2.5 构造基于 Pthreads 的并行 for 循环分解、分配和执行机制

老师在上课的时候也讲过,OpenMP 实际上是基于 pthread 完成的,所以我们可以自己用 pthread 来模拟 OpenMP 的并行化 for 循环的过程。

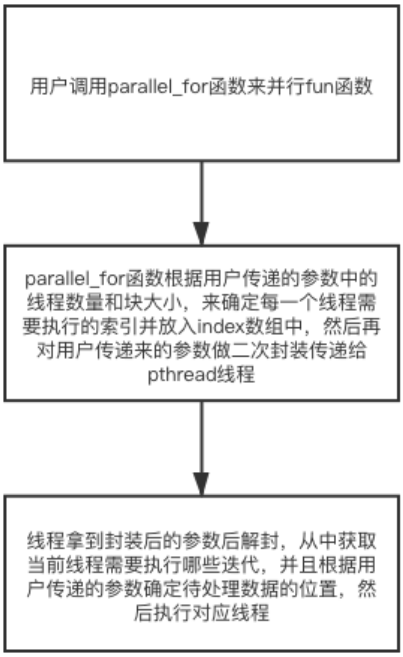
根据题目要求，我们需要实现 openMP 的静态循环划分。题目中给的参数中没有指定块大小，因此我额外指定了划分的块大小 chunksize，如下：

```
void parallel_for(int start,int end,int increment,void *(*functor)(void*),void
    *arg,int num_threads,int chunksize)
```

在这里为了给 pthread 的接口传递由用户传递来的 arg 和其他线程信息，我定义了如下的结构体：

```
struct parallel_arg{
    std::vector<int> index;//当前线程需要执行的循环的索引
    void * fun_arg;//用户传递给parallel_for函数的参数
    int increment;//每次循环增加索引数
};
```

用该结构体可以用来对用户传递的 arg 做进一步的封装来传递给 pthread，具体的流程如下：



实现后的 parallel_for 函数如下：

```
void parallel_for(int start,int end,int increment,void *(*functor)(void*),void
    *arg,int num_threads,int chunksize){
```

```
    struct parallel_arg argument[num_threads];
    for(int i = start;i < end;i += increment){
        argument[((i - start) / (chunksize * increment) ) % num_threads].index.
            push_back(i);//将索引按块划分给指定线程
    }
    pthread_t tid[num_threads];
    for(int i = 0;i < num_threads;i += increment){
        argument[i].fun_arg = arg;
        argument[i].increment = increment;
        pthread_create(&tid[i],NULL,functor,&argument[i]);
    }
    for(int i = 0;i < num_threads;i++){
        pthread_join(tid[i],NULL);
    }
}
```

将 parallel_for 函数编译为.so 文件

在这里我们只要使用 icpc -fPIC -shared -o libparallel_for.so parallel_for.cpp -lpthread 就可以编译得到 libparallel_for.so 文件，配合 parallel_for.h 就可以被用户程序调用了。

将基于 OpenMP 的通用矩阵并行改造成基于 parallel_for 函数并行化

我们首先应该编写 pthread 最终执行的函数，线程需要解封 parallel_for 传递来的参数，获取 pthread 执行的具体的循环的迭代次数，放在 vector<int> 中，然后我们只要遍历 index 中的迭代次数即可。

具体的代码如下：

```
void* parallel(void* arg){
    parallel_arg* parallel_arg = reinterpret_cast<struct parallel_arg*> (arg);
    Matrix_Mul* matrix_arg = (Matrix_Mul*)parallel_arg->fun_arg;
    std::vector<int> index = parallel_arg->index;
    int increment = parallel_arg->increment;
    for(int i = 0;i < index.size();i += increment){
        for(int j = 0;j < matrix_arg->k;j++){
            for(int q = 0;q < matrix_arg->n;q++){
                matrix_arg->res[index[i]][j] += matrix_arg->Matrix_a[index[i]][
                    q] * matrix_arg->Matrix_b[q][j];
```

```
            }
        }
    }
}
```

在这里，我把 lab1 的串行代码的 calculate(Matrix_a, Matrix_b, res, m, k, n); 去掉，换成了如下代码即可：

```
parallel_for(0,m,1,parallel,&matrix_arg,thread_count,1);
```

这样就完成了代码的编写，然后我们只需要执行

```
icpc My_Parallel_MatrixMul.cpp -L. -lparallel_for -o
    my_parallel_for_matrix_mul -O3 -w -lpthread
LD_LIBRARY_PATH=$LD_LIBRARY_PATH:.
numactl -m 1 ./my_parallel_for_matrix_mul
```

就可以将程序运行起来了。

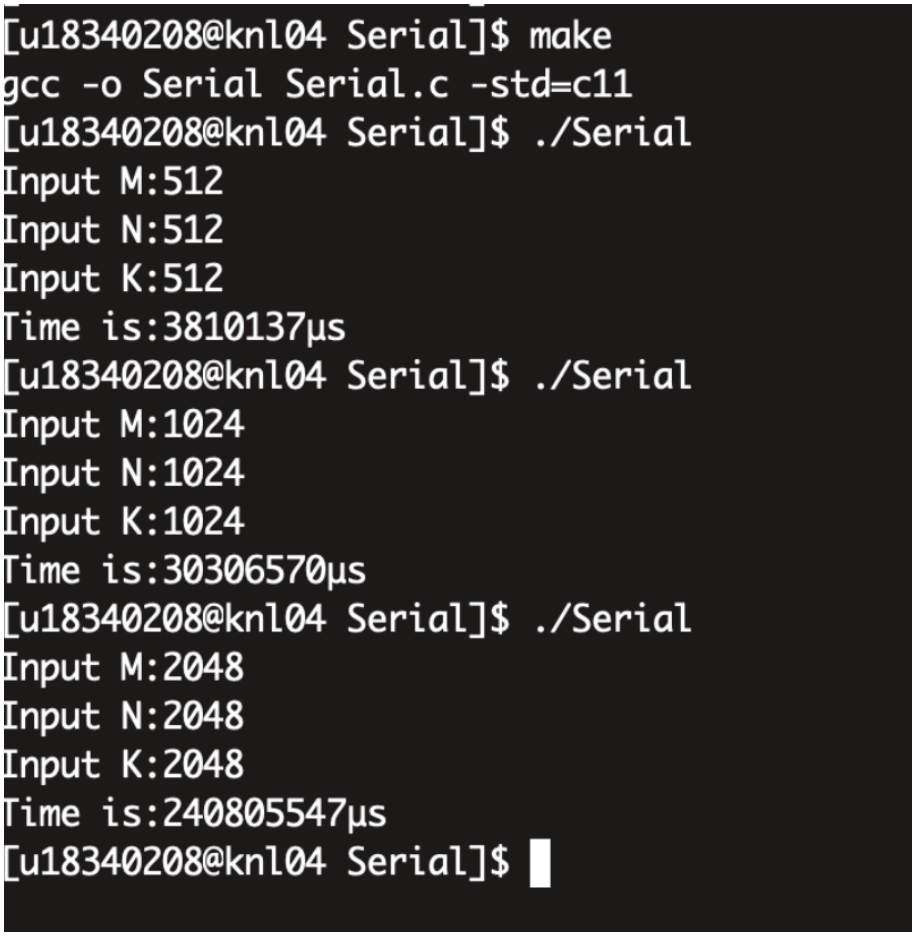
为了方便，我写了一个脚本，从头到尾来编译程序，如下：

```
dir=$(pwd)
icpc -fPIC -shared -o libparallel_for.so parallel_for.cpp -lpthread
icpc My_Parallel_MatrixMul.cpp -L. -lparallel_for -o
    my_parallel_for_matrix_mul -O3 -w -lpthread
LD_LIBRARY_PATH=$LD_LIBRARY_PATH:$dir
numactl -m 1 ./my_parallel_for_matrix_mul
```

3 实验结果

3.1 串行版本的 GEMM

在代码所在文件夹输入 make，用如下方式指定问题规模，如下：



通过这样的方式就可以指定问题规模了。

3.2 通过 OpenMP 实现通用矩阵乘法

用 OpenMP 实现来通用矩阵乘法后，可以用相同的方式执行并获得结果，如下：

```
[u18340208@knl04 Q1]$ make
gcc -o openmp_matrix_multiply openmp_matrix_multiply.c -std=c11 -O3
[u18340208@knl04 Q1]$ ./openmp_matrix_multiply
Input M:512
Input N:512
Input K:512
Time is:263011μs
[u18340208@knl04 Q1]$ ./openmp_matrix_multiply
Input M:1024
Input N:1024
Input K:1024
Time is:1992734μs
[u18340208@knl04 Q1]$ ./openmp_matrix_multiply
Input M:2048
Input N:2048
Input K:2048
Time is:13815210μs
[u18340208@knl04 Q1]$
```

我们可以看到，在使用 OpenMP 优化后，性能有了比较明显的提升。

3.3 基于 OpenMP 的通用矩阵乘法优化

3.3.1 使用 Intel Parallel Studio XE 来编译并进行优化

在这里不改变默认的循环调度方式，使用 Intel 编译器优化后的结果如下：

```
[u18340208@knl04 default]$ numactl -m 1 ./openmp_matrix_multiply
Input M:512
Input N:512
Input K:512
Time is:45576μs
[u18340208@knl04 default]$ numactl -m 1 ./openmp_matrix_multiply
Input M:1024
Input N:1024
Input K:1024
Time is:60414μs
[u18340208@knl04 default]$ numactl -m 1 ./openmp_matrix_multiply
Input M:2048
Input N:2048
Input K:2048
Time is:206328μs
[u18340208@knl04 default]$
```

发现运行时间减小了非常非常多，之前等待问题规模为 2048 的时候等待了很久，在这里一下子就可以跑出结果，说明优化的效果较好。

3.3.2 采用不同的循环调度方式来加速运行

在这里使用静态调度的结果如下：

```
[u18340208@knl04 static]$ ./openmp_matrix_multiply
Input M:512
Input N:512
Input K:512
Time is:75635μs
[u18340208@knl04 static]$ ./openmp_matrix_multiply
Input M:1024
Input N:1024
Input K:1024
Time is:91630μs
[u18340208@knl04 static]$ ./openmp_matrix_multiply
Input M:2048
Input N:2048
Input K:2048
Time is:414974μs
[u18340208@knl04 static]$
```

采用动态调度的结果如下：

```
[u18340208@knl04 dynamic]$ ./openmp_matrix_multiply
Input M:512
Input N:512
Input K:512
Time is:86251μs
[u18340208@knl04 dynamic]$ ./openmp_matrix_multiply
Input M:1024
Input N:1024
Input K:1024
Time is:65588μs
[u18340208@knl04 dynamic]$ ./openmp_matrix_multiply
Input M:2048
Input N:2048
Input K:2048
Time is:222674μs
[u18340208@knl04 dynamic]$
```

不难发现，似乎二者相较默认的调度方式没有什么提升，反而会出现性能的下降。

3.3.3 优化结果分析

为了统计的时候方便，我在测试性能的时候，取 $m = n = k$，并取问题规模为 512，1024 和 2048 三种情况。并且，由于我使用的节点的 CPU 有 64 核心，所以我使用了 64 个线程来适配它。经过 5 次测试后，取平均值，结果如下：

为了方便书写我们给上述所有优化方式都进行编号：

1. 串行 (无优化)
2. 用 #pragma omp parallel for 优化
3. 用 Intel 优化编译器并采用默认调度方式
4. 用 Intel 优化编译器并采用 schedule(static,1)
5. 用 Intel 优化编译器并采用 schedule(dynamic,1)

问题规模 / 优化方式编号	512	1024	2048
1	3820462μs	30392558μs	241381934μs
2	264532μs	1983423 μs	13905294 μs
3	45678 μs	60176 μs	198727 μs
4	78835 μs	91688 μs	424085 μs
5	56254 μs	65058 μs	232985 μs

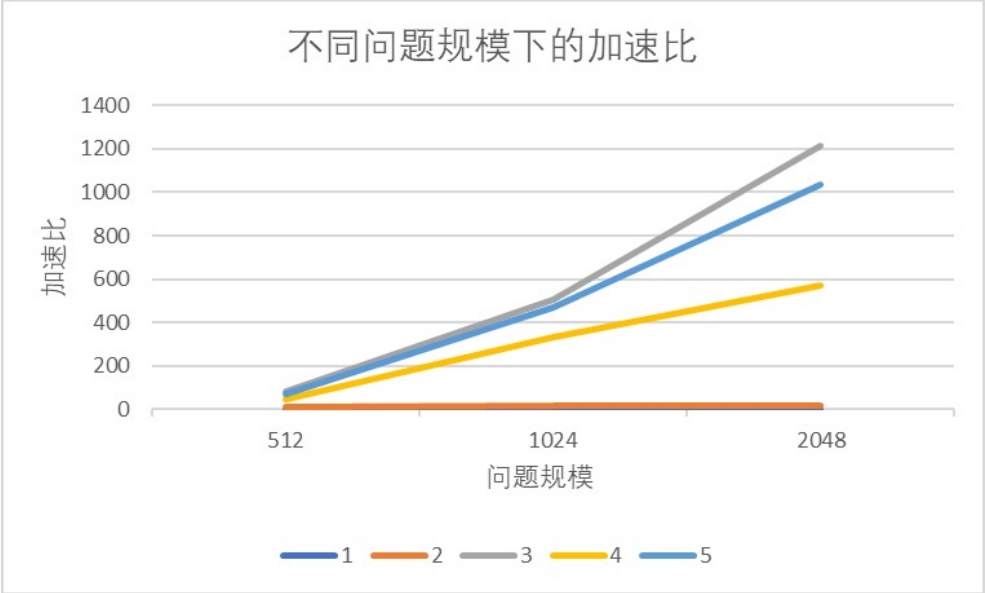
表 1: 不同优化方式下的结果

由此我们可以计算出加速比：

问题规模 / 优化方式编号	512	1024	2048
1	1	1	1
2	14.44	15.32	17.36
3	83.64	505.06	1214.64
4	48.46	331.48	569.18
5	67.91	467.16	1036.04

表 2: 不同优化方式下的结果

用更直观的方式看待加速比：

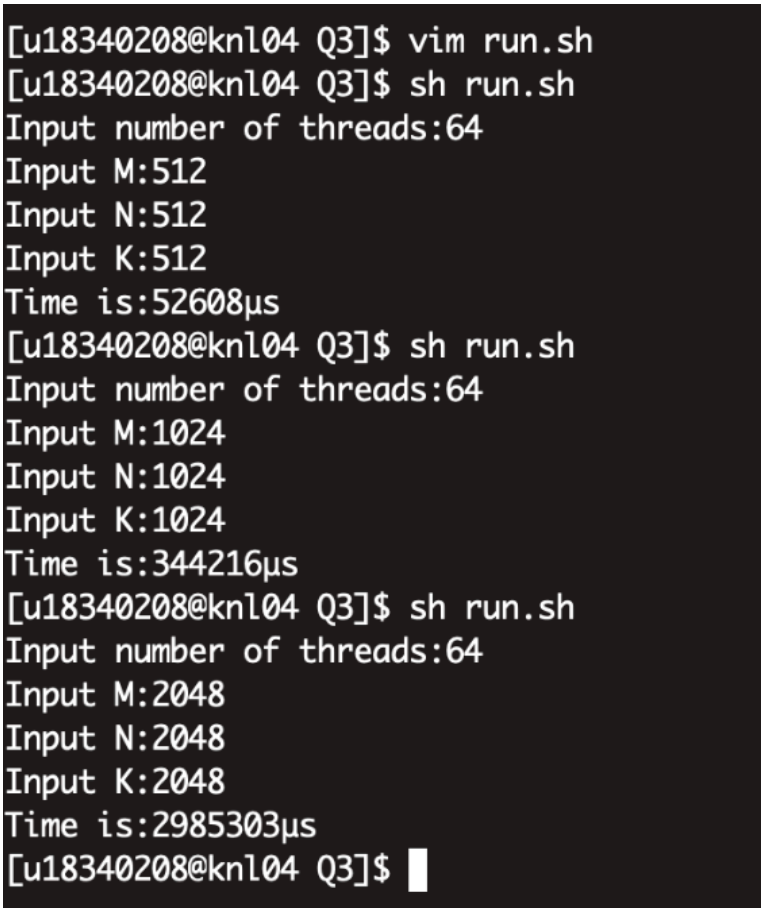


我们可以发现，在用 Intel 优化编译器并采用默认调度方式时加速比最高，问题规模为 2048 的时候，加速比可达 1214.64。这个其实非常直观，因为我等待串行执行非常长的时间，而在这种情况下一眨眼就可以算出结果。这也说明了 Intel 的优化非常强大。再加上向量化，我觉得就是在用 OpenMP 写 CUDA。

另外一个角度，在相同的条件下，使用默认的调度方式比 schedule(static,1) 和 schedule(dynamic,1) 的性能更好。这也是可以理解的，因为在通用矩阵乘法这个问题中，每次迭代的问题规模都是一样的，所以不存在线程负载不均的问题。而如果采用 schedule(static,1) 或 schedule(dynamic,1) 后，由于块大小都是 1，所以会导致局部性的丧失。而默认的调度方式下，所有线程负载均衡，且每个线程所执行的程序的迭代彼此之间都有局部性，所以会使得程序的性能比其他两种方式好。

3.4 构造基于 Pthreads 的并行 for 循环分解、分配和执行机制

在这里我们只需要执行 sh run.sh 程序即可，如下：



可以看到, 相对于串行的执行, 我们调用自己实现的 parallel_for 时性能还是有很大的提升的, 只不过相比用 Intel 有针对性的优化还是差了许多, 但是效果还是让人满意的。这也从侧面表现出 Intel 编译器充分利用了硬件潜能。这也是我们实现高性能计算的一个角度吧。

4 实验感想

- OpenMP 的优化比较简单，只要加上简单的 #pragma omp parallel for 就行了
- 采用 Intel 的编译器优化会使得性能提升一个台阶，原因是它充分利用了硬件的性能，大大地提高了程序性能。
- 最开始我优化串行的代码的时候，等了 4 分钟才出现结果，当我优化到最佳状态的时候一瞬间就计算完毕，这样的巨大差别也是高性能计算的魅力所在吧。
- 矩阵乘法这样的问题完全可以用 CUDA 编程在 GPU 上运行得到更高的加速，也期待接下来学习 GPU 编程后可以对该问题有进一步的优化。