Threads	1	3	6	12	16	20	24	28	32
GFLOPs	72.2	203	371	611	616	680	699	613	613
Threads	36	40	44	48					
GFLOPs	548	529	583	411					

Table. 1

Threads	1	3	6	12	16	20	24	28	32
GFLOPs	68.8	208	369	612	639	684	651	707	533
Threads	36	40	44	48					
GFLOPs	543	759	546	421					

Table. 2

Threads	1	3	6	12	16	20	24	28	32
GFLOPs	46.1	208	364	604	599	530	611	586	584
Threads	36	40	44	48					
GFLOPs	498	594	468	310					

Table. 3

Threads	1	3	6	12	16	20	24	28	32
GFLOPs	72.3	204	370	618	683	698	700	519	648
Threads	36	40	44	48					
GFLOPs	864	576	471	638					

Table. 4

Threads	1	3	6	12	16	20	24	28	32
GFLOPs	69.1	208	362	622	623	565	501	656	523
Threads	36	40	44	48					
GFLOPs	644	892	460	640					

Table. 5

Trending for total Gflops:

1- 12	12-40	40 - 48
Increasing	Fluctuating	Decreasing

Table. 6 Trending for dpotrf performance

Q1. For N = 5000, wT = 5000, nT = 1, I do the same experiments for 5 times. The result are in GFLOPs. As long as nT equals to 1, we can only run the dpotrf_() since the function will not satisfy "less than nT" condition. So the function will not run into cblas_dtrsm,cblas_dsyrk or cblas_dgemm. As a result, I can get performance of dpotrf directly.
As the tables show above, the calculation ability linearly increase as threads increase from 1 to 12. Meanwhile, the average Gflops per thread falls between 50 - 70. But as the threads go beyond 12, the average calculations goes down to around 20 – 30 Gflops.

The peak performance is not stable and we cannot tell how many threads we use can get the highest performance. But the highest performance mostly occurs when use 24 threads number or 40 threads but it is quite unstable.

Q2.

1. First use p=1 to find the best wT.

Try wT = 12,24,48,96,120,144,168 when N = 5000 and p=1

Threads	12	24	48	96	120	144	168
Factorization time(s)	4.28	1.42	0.836	0.674	0.561	0.550	0.550

Table. WT evaluation

It shows that 144 is a reasonable tile size we can use.

Threads	6	12	24	30	36	48
Loop(Gflops)	271	419	195	129	83.9	228
Fac time(s)	0.154	0.0995	0.017	0.323	0.497	0.183

Table. CholeskyLoop evaluation

Experiment environment:

N = 5000, p = 6,12,24,30,36,48 wT = 144, loop means choleskyLoop's Gflops for factorization and Fac time means Factorization time.

Directives choose:

The dtrsm step use static schedule with granularity 1.

Reasons:

1. dtrsm is a small step and each threads get dtrsm task will use similar time. So the work load is balanced.
2. Use static can reduce the overhead from distributing the tasks at run time. If we use dynamic, there is some overhead for work distribution.

So use static schedule is preferred.

The dsyrk and dgemm step use dynamic schedule with granularity 1
Reason:
1. Compared to dtrsm step, dsyrk and dgemm steps introduce heavier work due to the nested loop. Because a thread has to count for a single loop with if clauses, the work loading is unbalanced. Use dynamic scheduling can deal with this unbalance.
2. granularity *1 can ensure the balanced working load. Idle threads will be allocated a task in runtime.*

Because the the time that each thread excutes dsyrk and dgemm may be different(due to nested loop included), in order to balance the work loading, I use dynamic scheduling for that region.

Q3.

Threads	6	12	24	30	36	48
Cyclic(Gflops)	248	333	389	419	299	185
Cyclic Fac time(s)	0.168	0.125	0.107	0.0993	0.139	0.225
Block(P*Q) (Gflops)	142	189	283	335	287	307
Block Fac time(s)	0.293	0.220	0.147	0.124	0.145	0.136
Random(Gflops)	328	388	489	648	379	424
Random Fac time(s)	0.125	0.107	0.0853	0.0643	0.110	0.0983

Table. CholeskyRegion evaluation

Experiment environment:
N = 5000, p = 6,12,24,30,36,48 wT = 144, loop means choleskyRegion's Gflops for factorization and Fac time means Factorization time. Make P * Q nearly square. close and P<Q.
For example, when p= 6 , P=2 and Q=3, p= 12 P=3 and Q=4 and etc.

Because in openMP we cannot specify a task to a designated thread in default, using loop through ownerIdTile is the only way to complete this task, though it will introduce extra overhead when looping through the ownIdTile and judging whether the task belongs to the thread or not.
I choose wrap whole the code withnin the k's loop. Variable i and j should be private for each thread and k use firstprivate since it should be initialized by outside loop. And other variable should be shared.

Important synchronization:
According to the dependency, the execution should be divided into three parts – 1. dpotrf 2. dtrsm 3. dsyrk and dgemm. Part 2 depends on 1 and part 3 depends on 2. So there should be a barrier between each part, otherwise the result will be false. For example, most thread can skip the part 1 dpotrf, but they have to wait until a thread does A[k][k] updating. Besides before executing part 3, part 2 has to be completed.
Observation:

As threads increase from 6 to 30, the performance is going up. If threads is more than 30, the performance will go down. And random distribution gets better performance over blocks and cyclic distribution since the work load for each thread is more balanced.

Q4.

Threads	12(master)	12(ownerTileId）	24(master)	24(ownerTileId)	84(master)	84(ownerTileId)
Cyclic (Gflops)	2.94	1.58	1.82	1.48	2.05	0.647
Cyclic RHS time(s)	0.017	0.0317	0.0275	0.0338	0.0244	0.0773
Block(P*Q) (Gflops)	2.5	1.3	2.21	0.931	2.03	0.353
Block RHS time(s)	0.02	0.0385	0.0227	0.0537	0.0246	0.142
Random(Gflops)	2.18	0.615	2.03	1.37	2.96	1.51
Random RHS time(s)	0.0229	0.0813	0.0246	0.0366	0.0169	0.0332

Table. Comparison between master allocation and ownerTileId allocation

	p=12	p=24	p=84
Region(s)	0.0317	0.0338	0.0773
Loop(s)	0.0237	0.0205	0.0225

Table. RHS comparison between choleskyregion and choleskyloop

Experiment environment:

N = 5000, p = 6,12,24,30,36,48 wT = 144, three distributions are included. Evaluations are done by factorization time and Gflops. Make P * Q nearly square. close and P<Q.
For example, when p= 6 , P=2 and Q=3, p= 12 P=3 and Q=4 and etc.

As p goes up, initialization made by master thread does not change a lot. And it is obvious that increasing means nothing for the master thread. But for some ownerId determined initialization(block and cyclic distribution), the time for initialization slightly decreases. The random distribution seems not effected by p’s number.

In general, the performance for ownerId determined initialization has poor performance compared to master determined initialization since the cost for assigning number is quite slight. Besides, launching threads and finding a right task take time(caused by too many if clauses to judge). As a result, the cholesky region finally gets a poor performance compared to cholesky loop.

Q5.

Q6.
In order to reduce ts overhead, I can use a single thread to dynamically distribute tasks. And all tasks – dpotrf, dtrsm, dsyrk and dgemm can launch in parallel since I specify the dependency in task level. Even the task in different iteration(with different k) can be launched together as long as the meeting the dependency requirements.
Moreover, the dtrsm and dsyrk can be executed by same thread. So it helps to reduce some dependency checking and synchronization overhead. So the ts can be reduced given this strategy.

Threads	6	12	24	30	36	48	96
Loop(Gflops)	271	419	195	129	83.9	228	119
Fac time(s)	0.154	0.0995	0.017	0.323	0.497	0.183	0.35
Random(Gflops)	328	388	489	648	379	424	172
Random Fac time(s)	0.125	0.107	0.0853	0.0643	0.110	0.0983	0.243
Extra(Gflops)	380	495	854	833	924	875	372
Extra Fac time(s)	0.11	0.0841	0.0488	0.0500	0.0421	0.0476	0.127

Table. **Performance for loop, region and extra cholesky decomposition**

Experiment environment:
N = 5000, p = 6,12,24,30,36,48,96 wT = 144, three distributions are included. Evaluations uses cholesky loop, fast region cholesky(random distributed) and extra cholesky decomposition(based on task dependency).

Result:
According to the table, we find the task based decomposition overshadows other implementation. When a reasonable threads number selected, it runs multiple times faster than the region and loop implementation. And the p=36 gives the best performance. As the p goes beyond 36, the performance for the extra goes down. It may occur due to introducing parallel overhead – as system should spend more time on synchronization, checking thread's status and launching threads.

Q7.

Threads	12	24	36	48
Paralleled check dynamic (Gflops)	2.34	2.44	2.30	2.74
Paralleled check static(Gflops)	2.32	2.29	2.06	2.99

Original check(Gflops)	2.29	2.28	2.23	1.6

Table. **Paralleled check function and original check function**

Experiment environment:
N = 5000, p = 12,24,36,48 wT = 144, granularity for dynamic and static scheduling is 1 for load balance.

Approach:
For triMatVecMultPar function, I found two for loop has fixed length nT, and the y is the only one we should modified. So the first step is to flatten the nested for loop to eliminate branch clauses, which can reduce unnecessary overhead. I also try to use dynamical task scheduling and static scheduling to explore the best performance.
For triMatVecSolvePar, I find the possible parallel region is the nested loop, L[i][j] * y[j*wT..j*wT+wT-1] can be calculated in parallel, and yi -= L[i][j] * y[j*wT..j*wT+wT-1] should be atomic, using the reduction clauses can help.

Performance:
According to the experiments, I find dynamical scheduling has slightly better performance than static scheduling and linear checking. However, the performance of static scheduling is quite unstable and sometimes it will outperform others. I think there's a potential unbalanced loading problem within the static scheduling, which leads to unstable performance. So dynamic schedule is preferred here.

Part 2:

Q 9.

wT	2	4	8	16	32	64	128	256	512	1024	5000
Factorization(Gflops)	27.9	34.5	39.4	42	42.3	42.1	42.1	42.0	42.6	41.7	42.1

Experiment environment:
N = 5000, try wT = 2, 4, 8, 16, 32, 64, 128, 256, 512, 1024, 5000
Discussion:
As the wT increases to from 2 to 16, the factorization performance haves an obviously improvement. But once it beyonds 16, the performance does not have an obvious improvement and keep stable. The reason is that the CUBLAS uses a maximum tile with size 16. If wT is larger than 16, than maximum 16*16 threads will use tile algorithm to iteratively cover the whole matrix. But if wT is smaller than 16, use wT as the block(tile) size and wT*wT threads can process the a wT*wT matrix . The reason why use 16 as threshold tile size is that the maximum threads in a single block is 1024. The developer does not want to exhaust all units in a block since some additional algorithms can be applied to boost the performance. For example, using asynchronous copy to realize the pipelines can greatly overlap the processing and datacopy overhead.
As a result, the factorization performance does not change a lot after wT=16.

NVIDIA introduce these libraries to show Cuda's incredible paralleled calculation performance – multiple times faster than serial calculation. Because the client may not notice the higher performance is from well designed paralleling code, they will get a huge shock for the GPU's performance . As a result, NVIDIA can boast their GPUs and attract customers.
Moreover, these libraries provide scientists with calculations' APIs. For example, matrix decomposition, backsolve in a research are quite common. These libraries provides scientific with a easy but powerful tool to do computation even they know little about cuda programming.

Q10.

	wT	N	bF(tuneParam)	Kernel 1	Kernel 2	Kernel 3
Experiment 1	256	256	1	1	0	0

Experiment 2	256	768	1	3	3	4
Experiment 3	256	768	2	3	2	3

Table. Environment setting

In order to get single kernel performance, we need to use 3 equations to solve 3 different kernel's overhead. bF=1 means complete serialization. So I start with the experiments setting.

Setting:

For experiment 1, I set wT=N = 256. As a result nT is 1.

For experiment 2 and 3, I set wT to 256 and N = 768, so nT is fixed 3. Change the tune parameter to get the time.

The I use nT = 1 get one equations from experiment 1 and nT = 3 to get two equations from experiment 1 and 2, wT should be fixed, and it will be better if wT divides 32(wrap size) without reminder.

	Total(s)	Kernel 1(s)	Kernel 2(s)	Kernel 3(s)
Experiment 1	0.272	0.272	0	0
Experiment 2	5.07	3*0.272	3 * 0.746	4* 0.504
Experiment 3	3.82	3*0.272	2 * 0.746	3* 0.504

By solving the equations:

{
k1 = 0.272
3k1 + 3k2+4k3 = 5.07
3k1 + 2k2 + 3k3 = 3.84
}

we get the overhead k1=0.272 k2= 0.746 k3 = 0.504 for each kernel

So we can see use the "Environment setting" table with well selected wT, we can easily calculate every kernel's overhead.

Q11.

wT	2	4	8	16	32
1 X 1 block (Gflops)	27.9	34.5	39.4	42	42.3
wT X wT Block(GFlops)	2.9	11.0	43.1	121.9	161.1

Table. 1 X 1 tile comparing to wT by wT block

Experiment environment:

N = 5000, try wT = 2, 4, 8, 16, 32

Discussion:

wT=64 will use 64*64 threads in a block but it exceeds the maximum threads number in a block. It will lead to an failure since the threads in different block cannot use same global memory.

Q12.

wT	2	4	8	16	32
Global memory Block	2.9	11.0	43.1	121.9	161.1
Shared memory Block	2.2	10.4	39.6	146	175

Experiment environment:
N = 5000, try wT = 2, 4, 8, 16, 32

Discussion:
If the wT is not bigger enough(less than 4), using shared memory does not help a lot and may introduce extra overhead when dumping the global memory to shared memory and dumping back. But as wT goes to 32, the performance improvement is quite exciting (almost 10%).
Shared memory is worth to apply on dpotrf and dtrsm kernels since they all have a loop and should access the global memory for many times. Shared memory is nearly 10 times faster than the global memory. Besides, implementing shared memory is quite easy. So it is worth to do that.

Q13.
In extra implementation, I implement pipelines technology by stream. Using different streams can overlap data transferring and calculation.

wT	2	4	8	16	32
Optimized block	2.2	10.4	39.6	146	175
Extra algorithm	1.8	8.8	37.2	135	223.3

Description:
The extra algorithm based on stream has highest performance when wT is 32. Only when all threads in a block keep busy can this pipeline strategy work well. Otherwise, the automatic pipeline optimization does not work well since each block does not keep load balance.

Q14.

N	5000	10000	Setting
OpenMPI	1030Gflops (0.017 s)	1970Gflops(0.338s)	P=48; block size= 6*8 wT=144
OpenMP	831Gflops(0.0548s)	584Glops(0.571s)	P=48; wT=144
CUDA	259Glops(0.161 s)	375Gflops(0.890 s)	wT=32

Experiment:
Decompose a N*N symmetric matrix. N = 5000, 10000

OpenMPI code is the most different since the message passing logic should be well designed otherwise it will introduce some tricky bugs that we cannot handle and debug. And we are not sure which thread goes wrong. Because each thread (master and slave) use the same code, how to make the code available to both master thread and slaves thread becomes a big challenge.

For OpenMP, it is the friendly for programmers since it just adds some directives and clauses original code. So, the logic is quite clear.
As for CUDA, some rumtime error is quite tricky to deal with since sometimes the hints given are meaningless or point to some irrelevant lines. For example, some illegal memory access are quite confusing. Moreover, programmer should take care of data processing across blocks since the block does not share the global memory.

According to the speedup, OpenMPI outperforms the other two paradigms. For OpenMPI, the bandwidth for message passing is a bottleneck, so the performance is bad. As for CUDA , there are still a lot strategies to apply. For example, passing message to multiple GPU to do paralleled computing, applying redundant calculation to eliminate if clauses and so on. In terms of OpenMP, it has better performance than others when N=10000. Anyway, the OpenMP that keep the high performance and simple and flexible programming style is preferred by programmers.

U6715243 Yuchao Zhang
I declare I fully contribute my work and these’s no significant errors or bugs in my code. All the codes have passed the test and work well.