LU Decomposition

Riley Weber and Vincent Popp

github.com/rileyweber13/fast-lu-decomposition

Why LU Decomposition?

- Allows $Ax = b$ to be computed in linear time if A is constant
- Common in machine learning
  - Model is a matrix
  - Must be applied to many data points (vectors)

Algorithm

- Find A = LU
- Change everything under the diagonal to zero using gaussian elimination. Matrix U is left
- Factors used to eliminate entries become the parts of matrix L

$$A = \begin{bmatrix} -5 & 3 & 4 \\ 10 & -8 & -9 \\ 15 & 1 & 2 \end{bmatrix}$$

$$L = \begin{bmatrix} 1 & 0 & 0 \\ -2 & 1 & 0 \\ -3 & -5 & 1 \end{bmatrix}$$

$$U = \begin{bmatrix} -5 & 3 & 4 \\ 0 & -2 & -1 \\ 0 & 0 & 9 \end{bmatrix}$$

Algorithm

$$
\begin{array}{c c}
\text{column } 0 & \\
\left[\begin{array}{ccc} -5 & 3 & 4 \\ 10 & -8 & -9 \\ 15 & 1 & 2 \end{array}\right] & \begin{array}{l} \\ \text{row } 1 \\ \\ \end{array}
\end{array}
\longrightarrow
\begin{bmatrix} -5 & 3 & 4 \\ 0 & -2 & -1 \\ 15 & 1 & 2 \end{bmatrix}
$$

$$
\begin{bmatrix} 1 & 0 & 0 \\ -2 & 1 & 0 \\ & & 1 \end{bmatrix}
$$

$$\text{multiplier} = -10 / -5 = 2$$

$$\text{row } 1 = \text{row } 0 \cdot \text{multiplier} + \text{row } 1$$

for every entry in the row

Model: flops

- Disclaimer: modelling this problem is complicated. Model is approximate
- flops
    - compute multiplier n^2/2 times
    - multiply and add for for every entry: 2n^2 times
    - invert mul for lower matrix n^2/2 time
- ~ 3 flop per entry

Model: memops

- Memops
  - read 2 for mul for lower triangle entries: n^2
  - write mul once: n^2 (could be combined with one of the reads above for a r/w)
  - read above row for mul/add: n^2
  - read/write self: n^2
- ~ 3n^2 * (data type size) reads
- ~ 2n^2 * (data type size) writes
- Simplified: 2n^2 read/writes
  - Likely to be the bottleneck

Model: Summary

- Model
  - Num flops: 3n^2
  - Bytes r/w: 2n^2 * 8
- Measured rates:
  - 1.7 TFlop/s
  - 136 GB/s
- Expected performance
  - Flop limit: 566 Giga-entries/second
  - Memory limit: 8.5 Giga-entries/second

Naive Implementation

```
void lu_factorize_sequential(Matrix &m){
 double diag, target, multiplier;
 size_t n = m.size();

 for (size_t col = 0; col < n; col++){
   diag = m[col][col];
   for (size_t row = col+1; row < n; row++){
     target = m[row][col];
     multiplier = -target/diag;
     for (size_t col_2 = col; col_2 < n; col_2++){
       m[row][col_2] = m[col][col_2] * multiplier + m[row][col_2];
     }
     m[row][col] = -multiplier;
   } } }
```

Peak performance:
Matrix size 32
0.02 Gigaentries/second

Expected:
8.5 Gigaentries/second

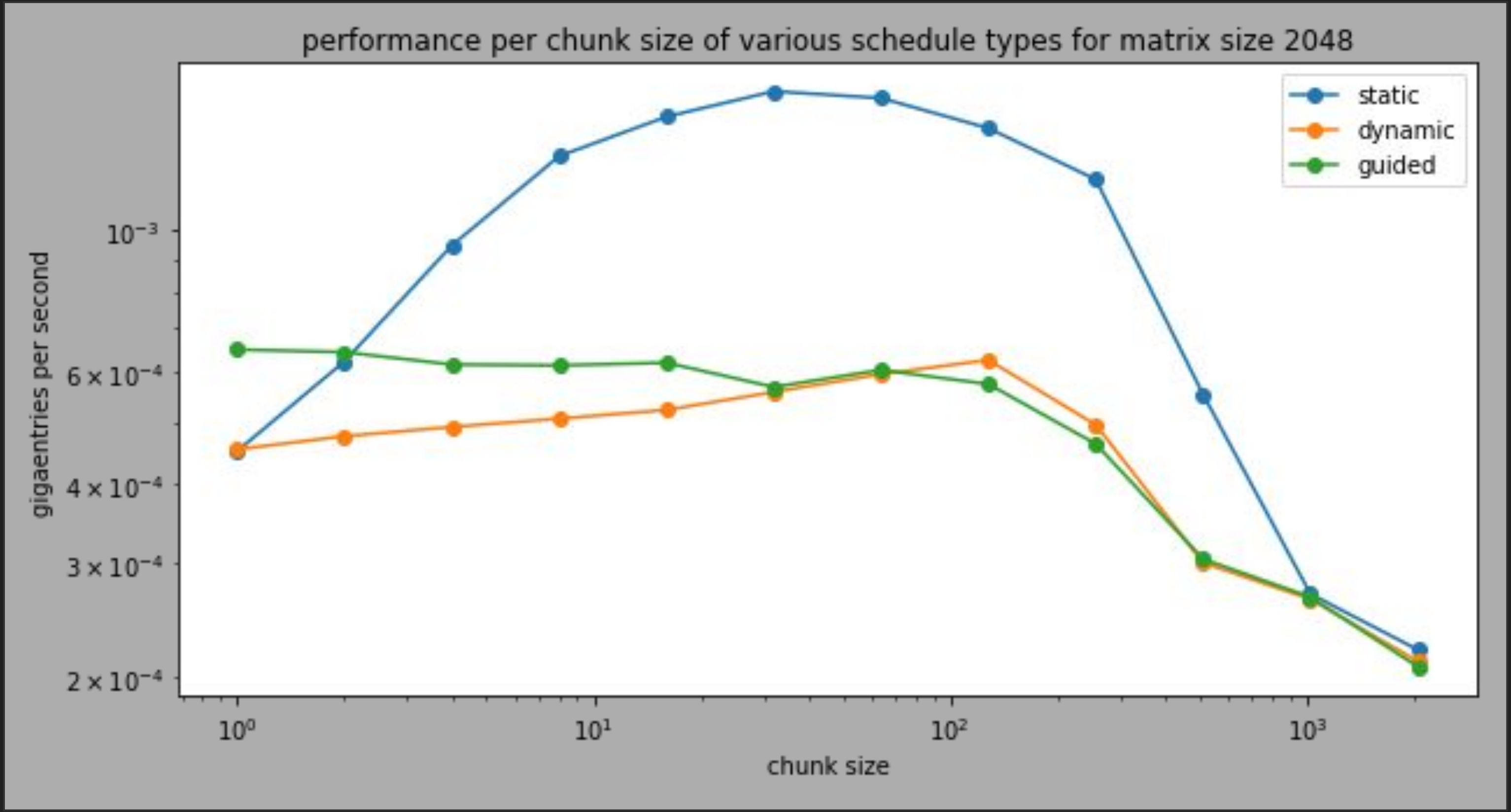
Parallel With OpenMP

```
void lu_factorize_parallel(Matrix &m, omp_sched_t sched_type, size_t chunk_size){
 double diag, target, multiplier;
 for (size_t col = 0; col < m[0].size(); col++){
   diag = m[col][col];
   #pragma omp parallel
   {
   omp_set_schedule(sched_type, chunk_size);
   #pragma omp for collapse(1) schedule(runtime)
   for (size_t row = col+1; row < m.size(); row++){
     target = m[row][col];
     multiplier = -target/diag;
     for (size_t col_2 = col; col_2 < m[0].size(); col_2++){
       m[row][col_2] = m[col][col_2] * multiplier + m[row][col_2];
     }
     m[row][col] = -multiplier;
   }}}}
```

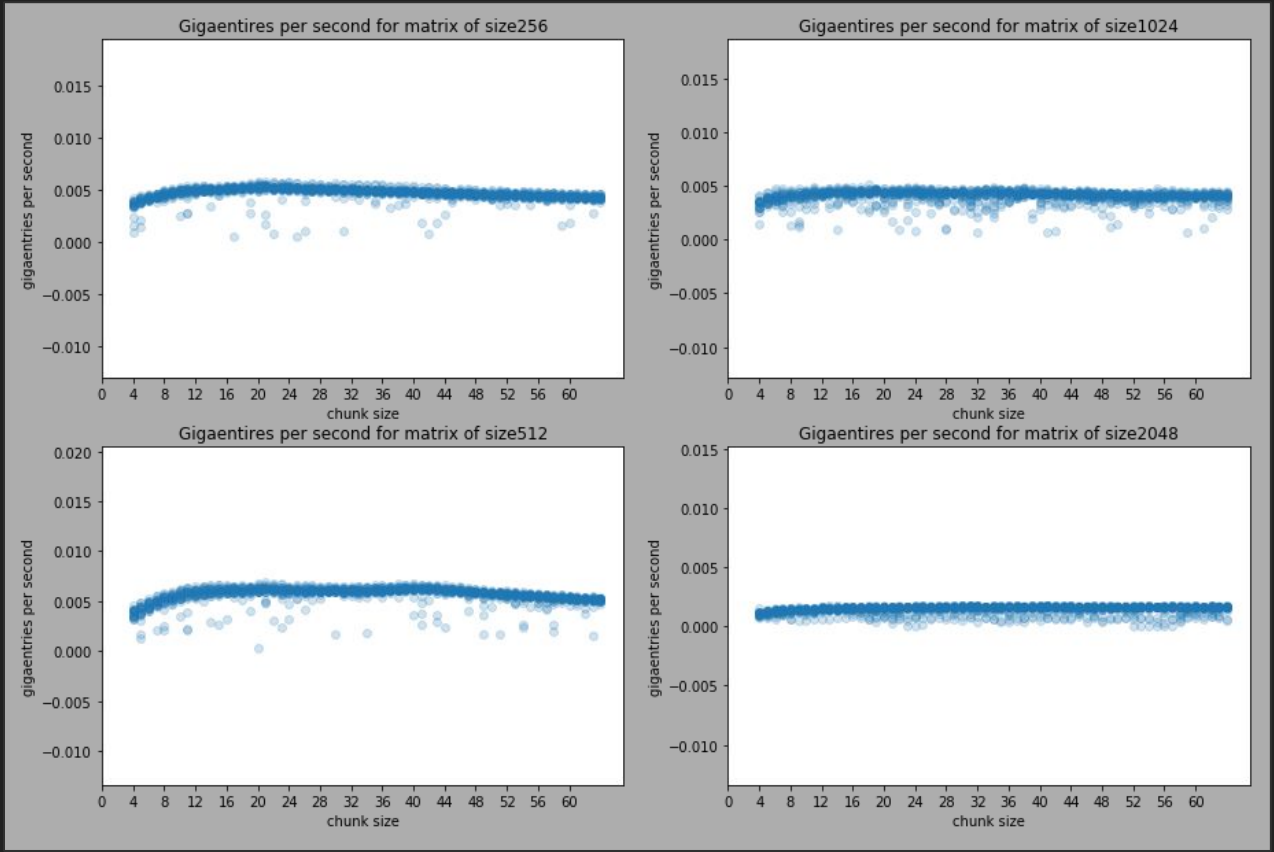
Peak performance:
Matrix size 1024
0.0047 Gigaentries/second

But, did show speedup of 13.4x on matrix size 2048!

Parallel Schedule Tests



Parallel Schedule Tests

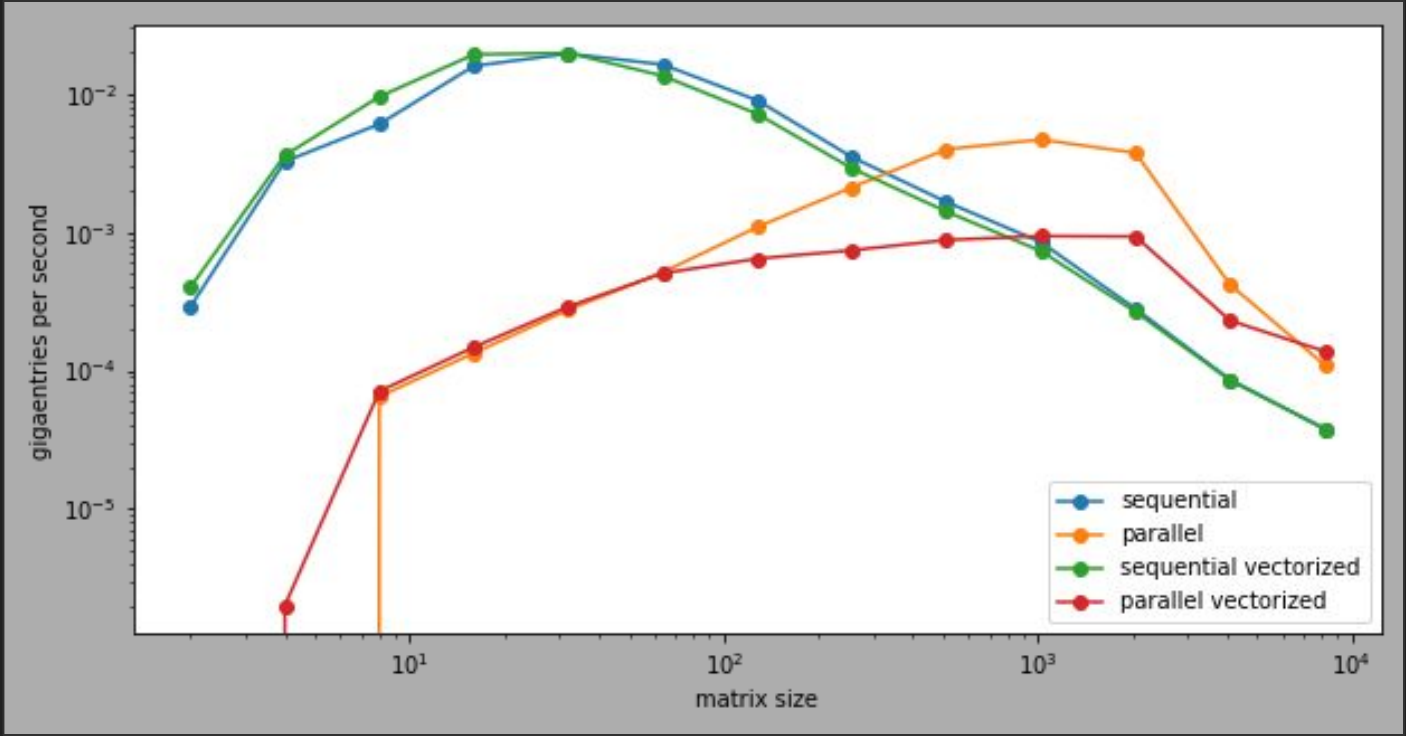


Vectorized

```
for (size_t row = col+1; row < n; row++){
  target = m[row][col];
  multiplier = -target/diag;
  multiplier_vector = _mm256_broadcast_sd(&multiplier);
  for (size_t col_2 = col - offset; col_2 < n; col_2 += 4){
    a = _mm256_load_pd(&m[col][col_2]);
    c = _mm256_load_pd(&m[row][col_2]);
    result = _mm256_fmadd_pd(a, multiplier_vector, c);
    _mm256_store_pd(&m[row][col_2], result);
  }
  m[row][col] = -multiplier;
}
```

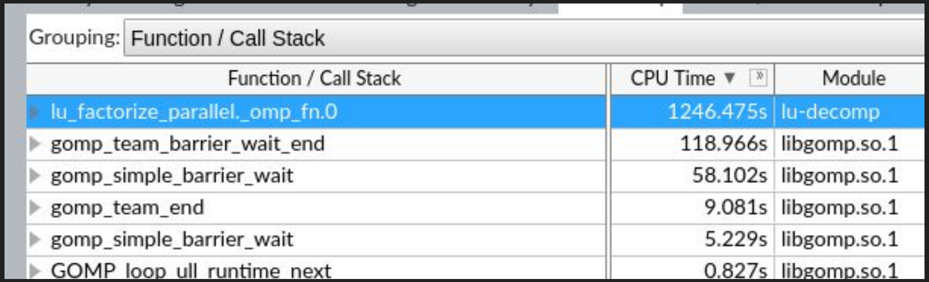
Vectorizing this for loop made no change in sequential code and actually worsened performance

Pipeline Comparison



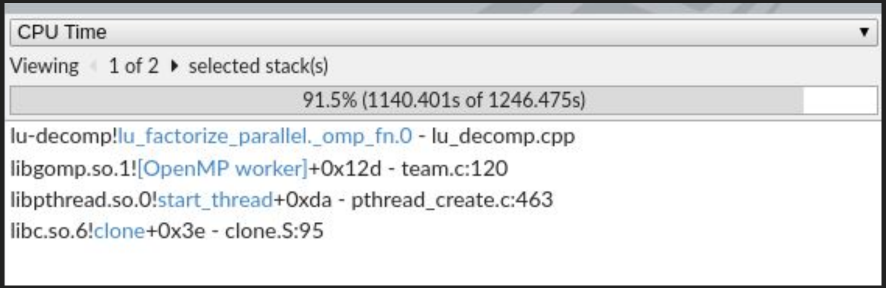
Investigation

- Using Intel VTune
- lu_factorize_parallel (my parallel implementation) makes up 91.5% of execution time



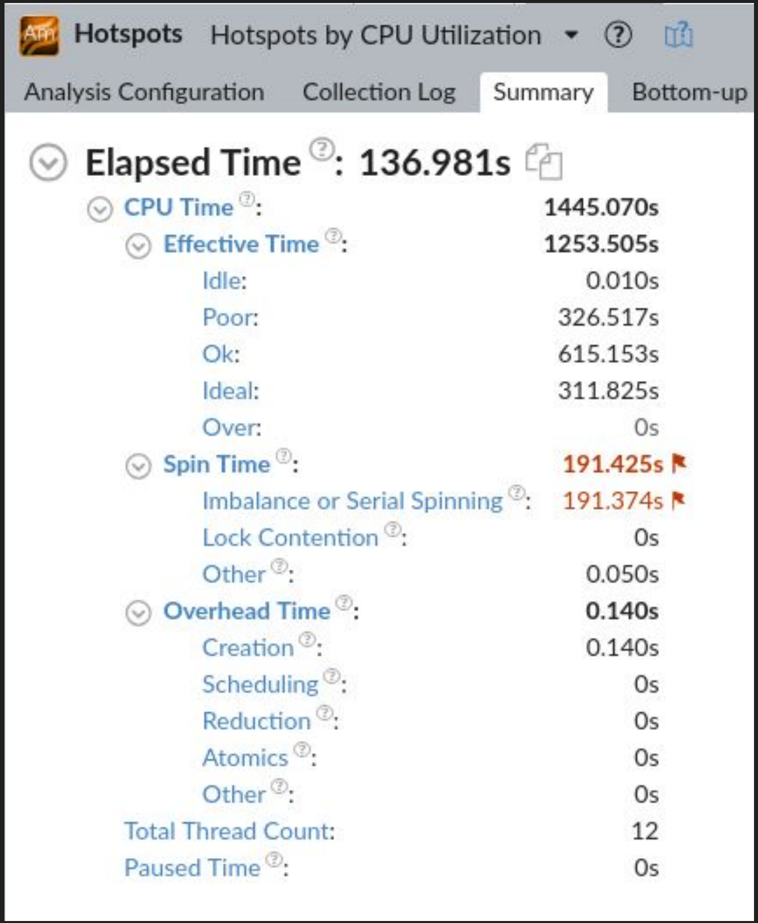
Grouping: Function / Call Stack

Function / Call Stack	CPU Time ▼	Module
lu_factorize_parallel._omp_fn.0	1246.475s	lu-decomp
gomp_team_barrier_wait_end	118.966s	libgomp.so.1
gomp_simple_barrier_wait	58.102s	libgomp.so.1
gomp_team_end	9.081s	libgomp.so.1
gomp_simple_barrier_wait	5.229s	libgomp.so.1
GOMP_loop_ull_runtime_next	0.827s	libgomp.so.1

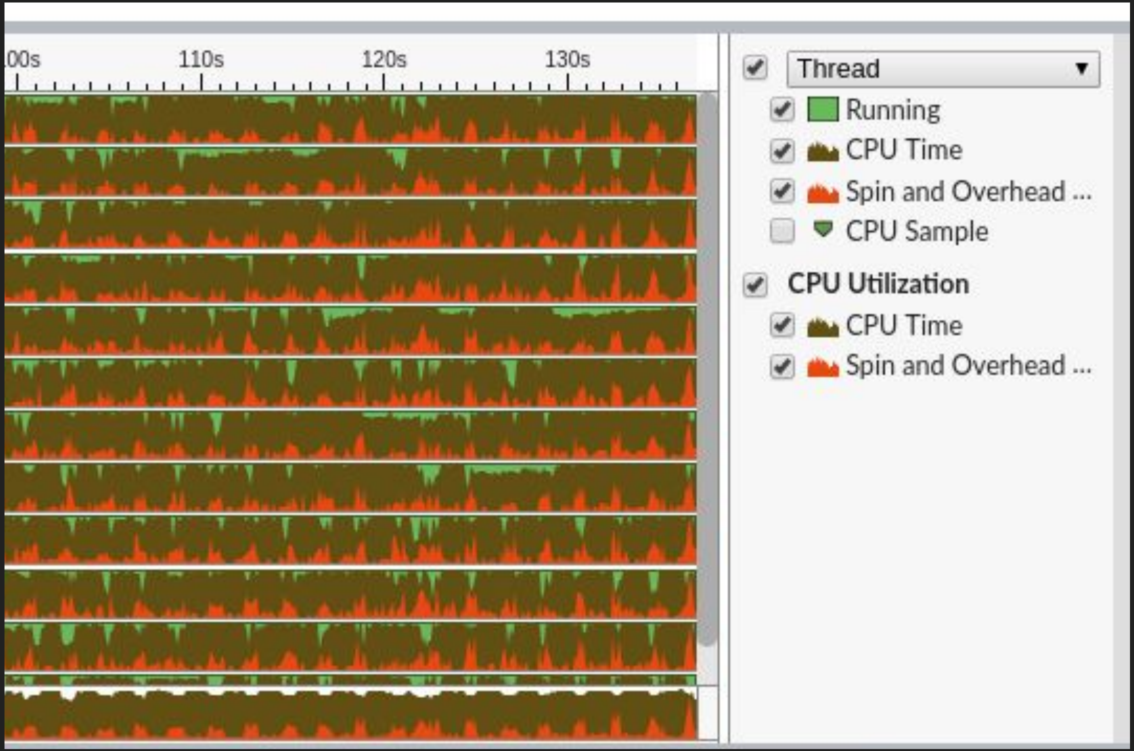


Investigation

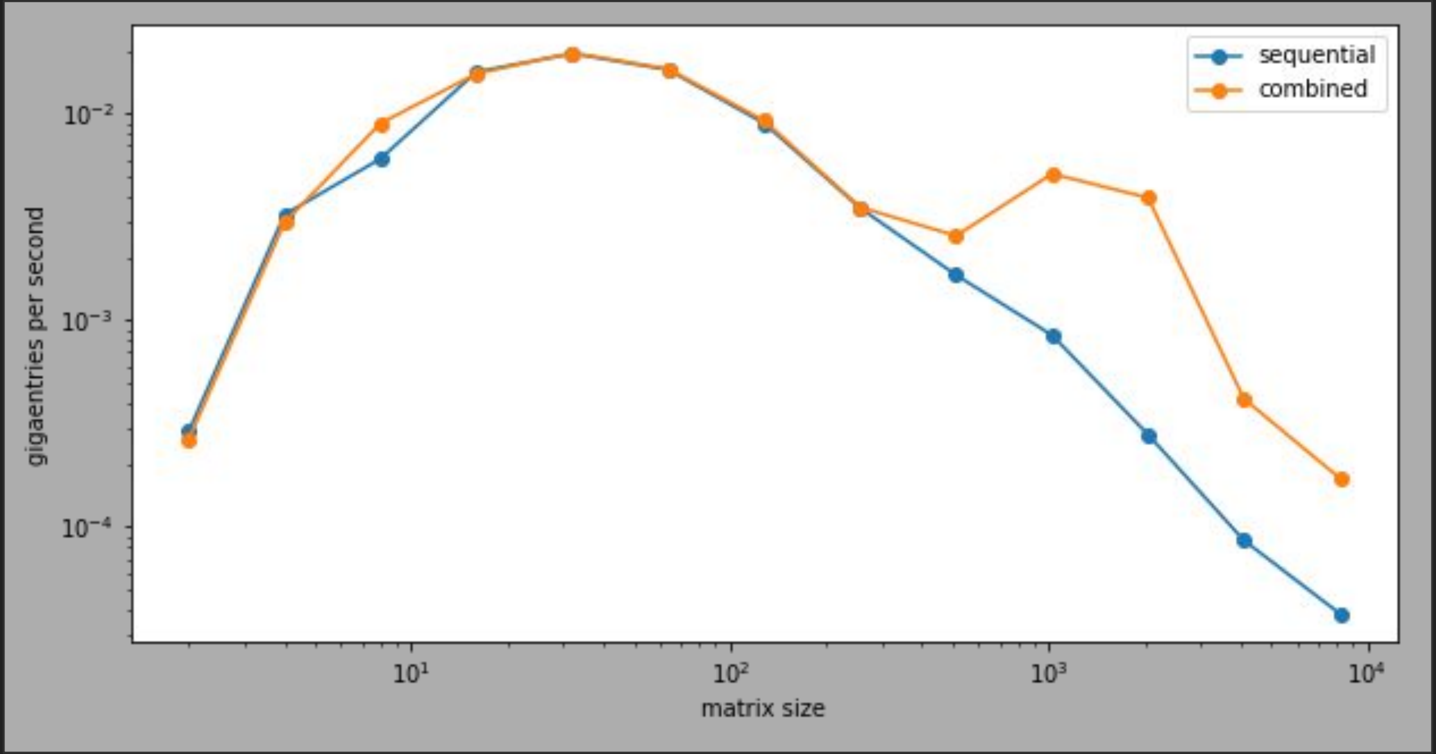
- Very little effective time is “Ideal”
- Spin time makes up about 15% of computation time
  - Should be reduced, but not first priority



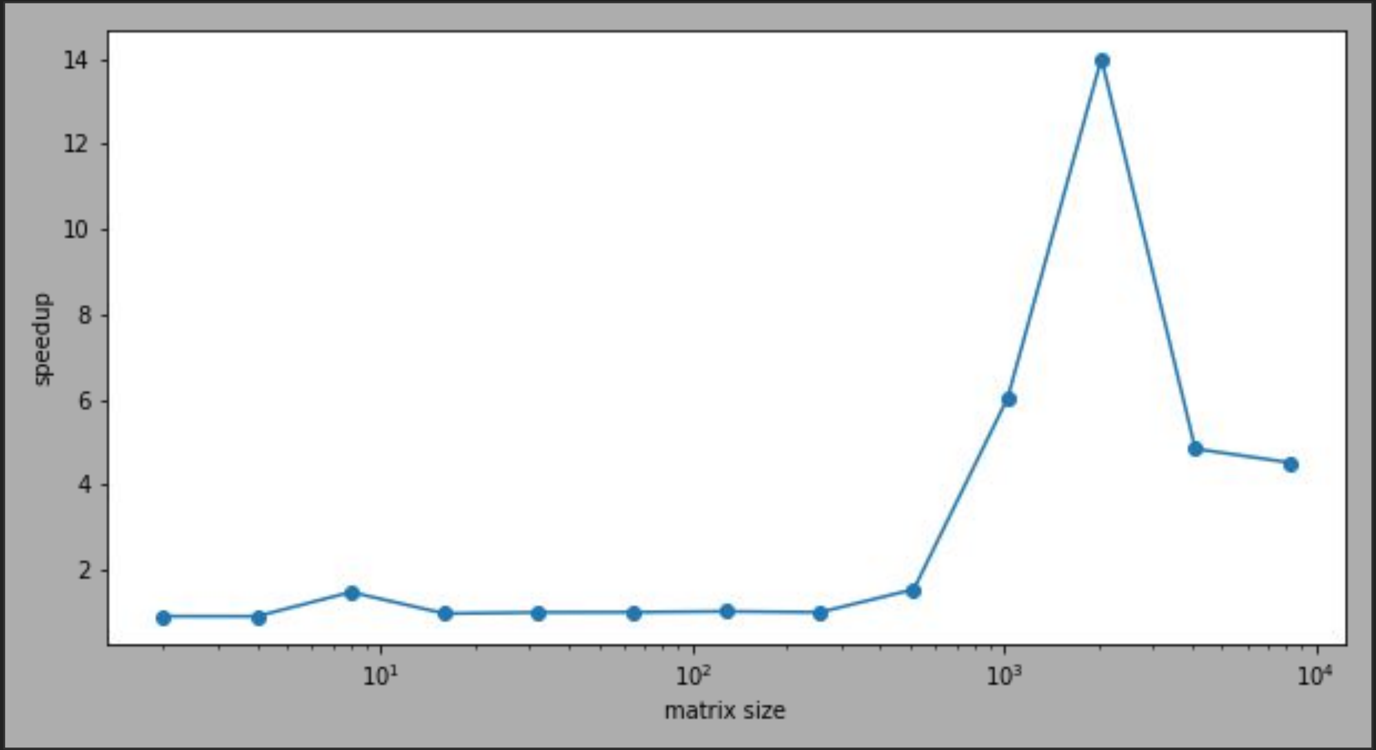
Investigation



Final Results



Final Results



Future Work

- Investigate what Intel VTune considers “Ideal” usage
- Investigate memory as the bottleneck
- Reduce spin time and improve work balance
    - Not sure how this can be reduced: many schedules and chunk sizes have already been evaluated

Takeaways

- Compiler is VERY smart
- OpenMP schedules make a huge difference

GPU

CUDA Idea?

We also explored doing a CUDA implementation.

The attempted approach:
  for each diagonal (pivot):
    calculate L's pivot column (sequential)
    update all entries in source matrix (parallel)
      using pivot-column of L * entries in respective rows

This would work because calculating columns of L are depend on one another, but each row of L is independent.

The reverse is true for Matrix U.

GPU Model

Recall:

Num flops: 3n^2

Bytes r/w: 2n^2 * 8

GPU Stats:

FLOPS: 4 TB/s
Bandwidth: 4TB/s

Thank You!

Riley Weber and Vincent Popp

github.com/rileyweber13/fast-lu-decomposition