Regression, chaos and misfortune

A wrong way towards dense-sparse matrix multiplication optimization

Leedagee

June 2023

Environment

Hardware: `hustcpu01`

1. 2x Xeon Gold 6338 @ 2.0GHz
2. 256G mem @ unknown speed

Software:

1. Ubuntu Focal 20.04 LTS
2. nixpkgs-23.05 (cmake & numactl)
3. IntelLLVM 2022.1.0
4. vtune 2023.1.0

Configuration:

1. 16 threads
2. `-O3 -ffast-math`
3. Inter-procedure Optimization
4. IntelLLVM icpx
5. Intel OpenMP `-fiopenmp`
6. `numactl --membind=0 --cpubind=0`
7. `/tmp/dataset` matrices

Pattern & Size	Baseline	Optimized	Opt Avg of 1024-runs
banded-1024	11.777s	18.291ms@643.9x	2.038ms@5779x
banded-2048	94.319s	37.324ms@2527x	9.344ms@10094x
diagonal-1024	11.893s	17.009ms@699.2x	2.027ms@5867x
diagonal-2048	94.609s	37.552ms@2520x	9.327ms@10144x
general-1024	11.869s	21.257ms@558.4x	3.718ms@3192x
general-2048	94.636s	45.591ms@2076x	16.658ms@5681x
symmetric-1024	11.717s	19.800ms@591.8x	2.913ms@4022x
symmetric-2048	94.188s	52.412ms@1797x	13.180ms@7146x
triangular-1024	11.861s	14.692ms@807.3x	3.777ms@3140x
triangular-2048	94.364s	44.413ms@2125x	16.850ms@5600x

The problem and thoughts

1. Matrix B is sparse
2. For each non-zero element in $B_{r,c}$, the computation involves reading $A_{*,c}$ and writing $C_{*,r}$
3. For memory contiguous, instead of computing AB^T, compute $\left(BA^T\right)^T$
4. To avoid cache miss:
    - Each row in C^T will be written by exactly one thread in the parallel region.
    - Memory Blocking techniques should be used on A^T (but proved to be wrong >_<)

The initial implementation

commit `9d5381`

- Two transpose operation.
- Naive iterating through B.
- SIMD by compiler (it seems)

Results are premising actually:

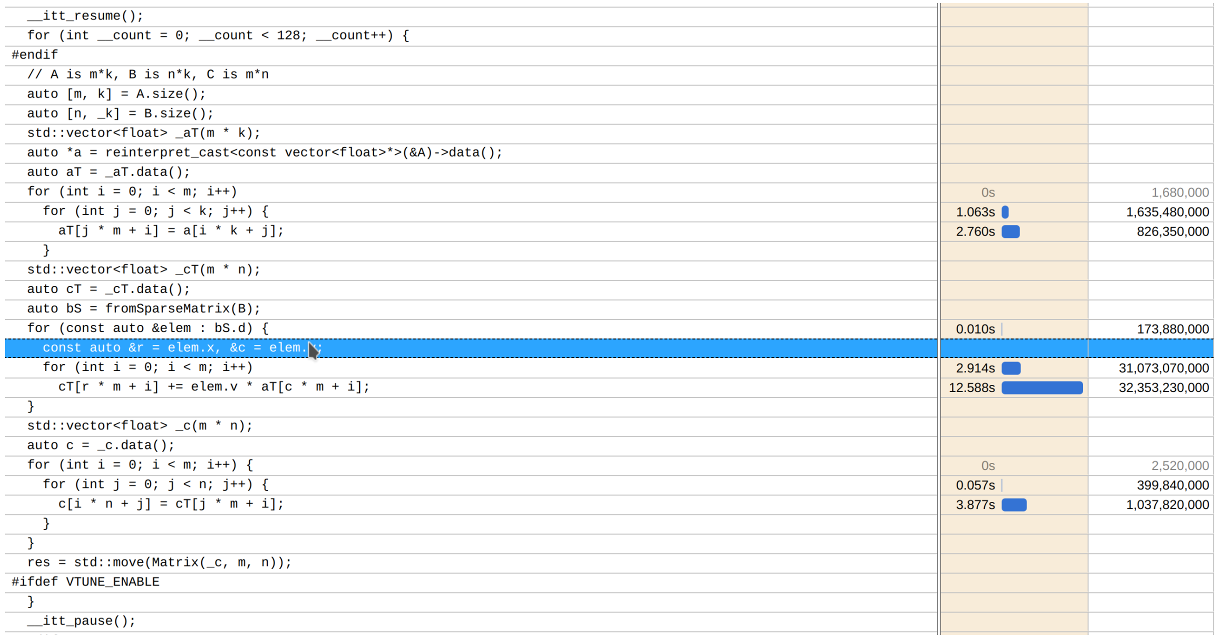
1. 39.7ms / 287ms on banded
2. 381ms / 2942ms on general

Note:

1. I misnamed COO as CSR, until BCSR was implemented, the naming are wrong all the code.
2. The performance test here are done on my i5-1240P laptop, performance may be inconsistent.

Profiling on the naive impl shows that the `Matrix#at` function and `[]` operator are not optimized out. Given how `Matrix` / `SparseMatrix` are implemented, a mind-explosive `reinterpret_cast` is used.

Profiling after that:



```
  __itt_resume();
  for (int __count = 0; __count < 128; __count++) {
#endif
  // A is m*k, B is n*k, C is m*n
  auto [m, k] = A.size();
  auto [n, _k] = B.size();
  std::vector<float> _aT(m * k);
  auto *a = reinterpret_cast<const vector<float>*>(&A)->data();
  auto aT = _aT.data();
  for (int i = 0; i < m; i++)                                        0s         1,680,000
    for (int j = 0; j < k; j++) {                                1.063s     1,635,480,000
      aT[j * m + i] = a[i * k + j];                              2.760s       826,350,000
    }
  std::vector<float> _cT(m * n);
  auto cT = _cT.data();
  auto bS = fromSparseMatrix(B);
  for (const auto &elem : bS.d) {                                0.010s       173,880,000
    const auto &r = elem.x, &c = elem.y;
    for (int i = 0; i < m; i++)                                  2.914s    31,073,070,000
      cT[r * m + i] += elem.v * aT[c * m + i];                  12.588s    32,353,230,000
  }
  std::vector<float> _c(m * n);
  auto c = _c.data();
  for (int i = 0; i < m; i++) {                                      0s         2,520,000
    for (int j = 0; j < n; j++) {                                0.057s       399,840,000
      c[i * n + j] = cT[j * m + i];                              3.877s     1,037,820,000
    }
  }
  res = std::move(Matrix(_c, m, n));
#ifdef VTUNE_ENABLE
  }
  __itt_pause();
```

It is highly memory bound

- Transpose ~80%, CPI goes 13
- Calculation ~50%, mainly random write

Next steps:

1. Memory blocking applied on transposing
2. Scan B in different order
    - Transposed, significant improvement (about 1.5x), calculation is now random reading
    - Looping as if doing memory blocking, actually not very useful

As algo changes later, more details on this implementation are omitted.

ALBUS-like partition

- Implement CSR, in this case, BCSR.
- Column blocking is enabled at this time (128 columns as a block)
- Switched to icpx & `-fiopenmp` for vtune analyze
- Paralleled transposing of A and scanning of B
- Still memory bound (of course for sparse ones)

AVX512 intrinsics

Just write intrinsics (quite exciting, isn't it?), and get optimized out by compiler (confused and disappointed this time)

```
// calculation
__m128 vv = _mm_broadcast_ss(&v);
__m512 vvv = _mm512_broadcast_f32x2(vv);
for (int j = 0; j < m; j += 16) {
    __m512 cv = _mm512_load_ps(&cLine[j]);
    __m512 av = _mm512_load_ps(&aT[c * m + j]);
    __m512 tmpv;
    cv = _mm512_fmadd_ps(cv, av, tmpv);
    _mm512_store_ps(&cLine[j], cv);
    // cLine[j] += v * aT[c * m + j];
}

// non-temporal memory store
for (int j = 0; j < m; j += 16) {
__m512 cv = _mm512_load_ps(&cLine[j]);
_mm512_stream_ps(&cT[r * m + j], cv);
}
```

Finding correct blocking size

Just write script to enumerate through possibly sizes.

TB / CB	128	256	512	1024
16	38.661ms	30.793ms	28.092ms	26.331ms
32	37.508ms	34.084ms	28.355ms	25.406ms
64	37.422ms	30.036ms	29.278ms	25.146ms
128	42.755ms	30.115ms	27.183ms	25.194ms
256	40.039ms	31.425ms	27.270ms	26.480ms

Disabled column blocking, use 64 as transpose blocking size.

NUMA and number of threads

Due to small matrices, the performance stopped increasing after 16 cores. Profiling later suggests maybe it's NUMA issue or just thread creation overhead.

Profiler suggests there's inter-NUMA (and socket) communication. Since 16 cores are enough, with `numactl`, the performance gets almost 2x speedup.

Toolchain

Tons of time spent on CMake, IntelLLVM and OpenMP

CMake version is critical.

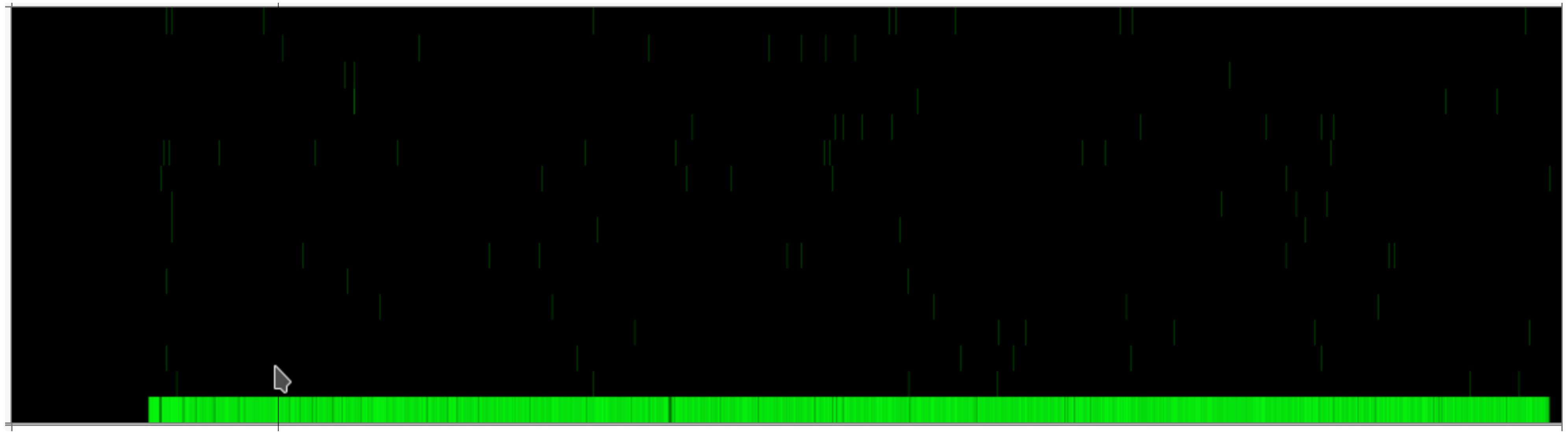
The nix environment pollutes the building, causing glibc version conflict with Intel OpenMP.

`ld`, `ranlib`, `ar` need to be set to IntelLLVM's to correctly handle the IPO/LTO objects (which are actually LLVM IR Bitcode)

Strange `-lrt`, `-ldl`, `-pthread` missing.

The Final Profiling

1. 93.5% Effective time on start_thread / clone@gilbc (?), filtered out in discussion later
2. 34.5% Memory bound on computation
3. Multi-threading is a joke after filtering out the start_thread calls.



Pushing it further

- Larger matrices
- Can the transpose be optimized out in actual application?
- How icpx optimize my code and gives no vfmadd instructions?

创

1. Slow cmake generation because of the matrices generation
2. Old cmake without icpx IPO support
3. Old nix / nixpkgs
4. clangd crashes on omp pragmas
5. icpx/clang crashes
6. No sepdk kernel modules on `hustcpu01`, unable to do memory bandwidth analyze
7. VTune cannot give any result under some circumstance
8. Broken typst bibliography support

Chaos

1. Local machine performance inconsistency (A 12th-Gen Intel Laptop)
2. Git, but just as snapshots of code, build script is not well managed
3. The baseline gets great speed up after IPO, comparing with baseline is meaningless somehow.
4. Local coding & debug, `hustcpu01` only for testing, not sure about the actual performance speedup when doing optimization
5. Because of all of these, I cannot form a report / presentation quickly (I have to go through all the work I have done to recall and gather all the details and performance metrics)
6. Different vtune profiler report on static / dynamic linked libiomp (same performance)

Conclusion: my workflow is just bullshit

Question:

1. The right way to characterize / summarize a single optimization?
2. Did I do too much environment tuning (or DevOps, if it is), and too few code optimization?
3. With collaboration on a much more complex project? I cannot image how chaotic it will be.
4. How to form the report / presentation efficiently? What should I collect in development?

References

1. heptagonhust, spmv的优化, 2022
2. H. Bian, J. Huang et al, ALBUS: A method for efficiently processing SpMV using SIMD and Load balancing, FGCS 2021 Volume 116, 2020
3. P. Koanantakool et al., Communication-Avoiding Parallel Sparse-Dense Matrix-Matrix Multiplication, 2016 IEEE IPDPS
4. G. Ruetsch, P. Micikevicius, Optimizing Matrix Transpose in CUDA, 2009
5. D. Yan, T. Wu, Y. Liu and Y. Gao, An efficient sparse-dense matrix multiplication on a multicore system, 2017 IEEE ICCT
6. Intel Crop., Intel 64 and IA-32 Architectures Optimization Reference Manual, 2023
7. OpenMP Org., OpenMP API Reference Guide, 2021