Tiling Matrix Multiplication

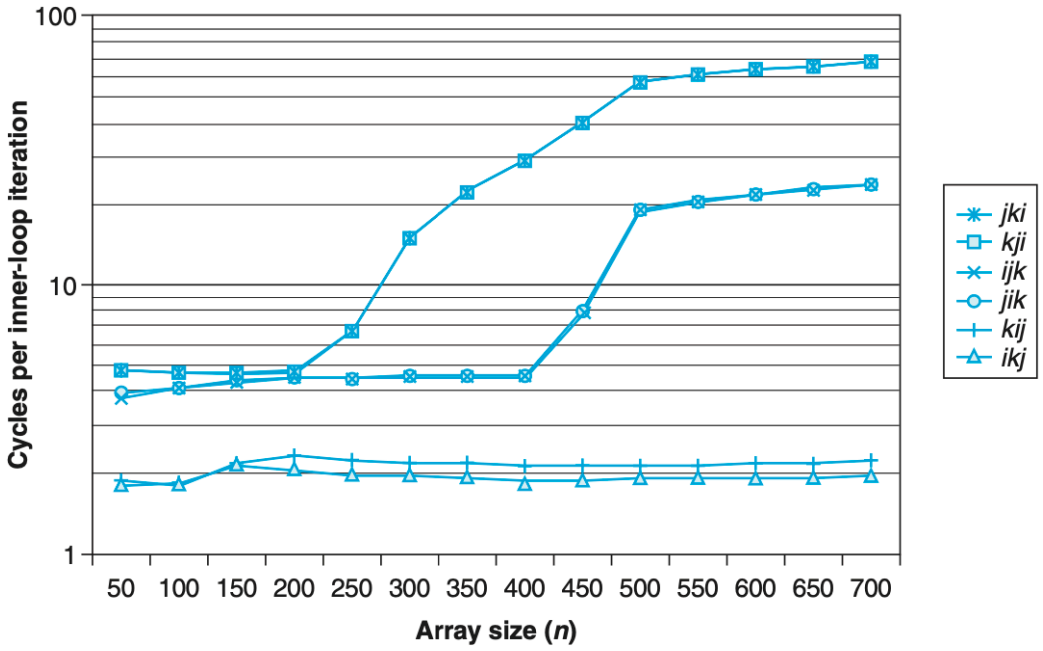
Parallelization and Performance Evaluation

早稲田大学大学院 基幹理工学研究科
情報理工・情報通信専攻
上田研究室 修士課程学生
5121F099 鍾中

Overview

Previous Research

- Rearranging loops can increase spatial locality so that improve performance of matrix multiplication program
  - For $A[i][j]$, $B[j][k]$, $C[i][k]$:
    - ikj has the best performance
    - As shown on the right[1]



[1] §6.6.2 of *Computer Systems A Programmers Perspective 3rd Edition*

Motivation

- Naive matrix multiplication program has a low throughput[1]:
  - Time:

1thread ikj	2threads ikj	4threads ikj	8threads ikj	16threads ikj
23.9	17.6	10.4	5.7	3.4
149.8	80.7	45.5	28.3	18.5
1698.3	1494.7	1150.1	889.9	461.8
13856.1	11665.1	8132.7	7174.9	5830.1
116180.8	88233.8	64998.1	60842.2	51850.9
936094.1	754361.0	495086.0	504249.3	428548.8

[1] All data (include the performance evaluation subsequent) get from *Parmigiano* machine of Ueda Lab

Motivation

- Naive matrix multiplication program has a low throughput:
  - Throughput (GOP/s - **G**iga [billion] **O**perations **P**er **S**econd):

1thread ikj	2threads ikj	4threads ikj	8threads ikj	16threads ikj
11	15	26	47	78
14	27	47	76	116
10	11	15	19	37
10	12	17	19	24
9	12	17	18	21
9	12	18	17	21

Motivation

- Theoretical performance (throughput):

$$T_{theoretical} = CPUClock \times CPUCore \times \frac{Width_{SIMD}}{Width_{float}} \times N_{SIMDUnit} \times N_{OperationPerSIMD(=FMA)}$$

- *Parmigiano* with AMD Ryzen Threadripper 2990WX 32-Core Processor[1]:
  - Base Clock (3.0GHz): $T_{theoretical-16} = 3.0 \times 16 \times \frac{256}{32} \times 2 \times 2 = 1536$
  - Max Boost Clock (4.2GHz): $T_{theoretical-16} = 4.2 \times 16 \times \frac{256}{32} \times 2 \times 2 = 2150.4$

[1] The specifications of CPU: https://www.amd.com/en/products/cpu/amd-ryzen-threadripper-2990wx

Motivation

- Theoretical performance of *Parmigiano*:

Theoretical Performance [GFLOP/s]							
SIMD width	clock[1]	# thread(s)					
256		1	2	4	8	16	32
float width	3	**96**	**192**	**384**	**768**	**1536**	**3072**
32	4.2	**134.4**	**268.8**	**537.6**	**1075.2**	**2150.4**	**4300.8**
# of SIMD Unit							
2							
# of operation per							
2							

[1] Clock: GHz

Motivation

- The throughput of naive matrix multiplication program is very low compared to the theoretical value
- Improve the throughput of matrix multiplication program is necessary
- There is a simple model to show how to improve the throughput of matrix multiplication program

Model

- Assume the computer has 2 levels of memory, fast and slow (etc. L1 cache and main memory)
- All data initially in slow memory
- Define:
  - m: number of memory elements (words) moved between fast and slow memory
  - t_m: time per slow memory operation
  - f: number of arithmetic operations
  - t_f: = time per arithmetic operation ($t_f \ll t_m$)

Model

- Get:
  - $q = f/m$: average number of flops (floating-point operations) per slow memory access
  - $Time_{min} = f \times t_f$: minimum possible time, when all data in fast memory
  - $Time_{actual}$: actual time, computation cost + data fetch cost

$$Time_{actual} = f \times t_f + m \times t_m = f \times t_f \times (1 + \frac{t_m}{t_f} \times \frac{1}{q})$$

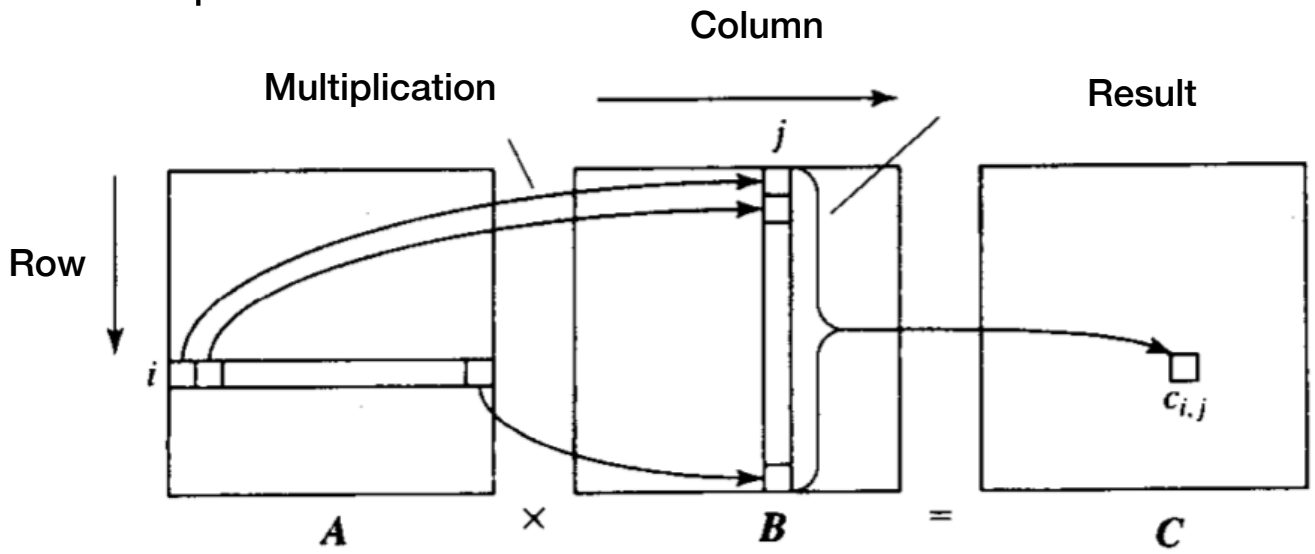
Model

$$Time_{actual} = f \times t_f + m \times t_m = f \times t_f \times (1 + \frac{t_m}{t_f} \times \frac{1}{q})$$

- From this formula,
  - Larger q means actual time closer to minimum time
- Tiling matrix multiplication program could have a larger q than naive one

Tiling Matrix Multiplication

- Naive matrix multiplication[1]:

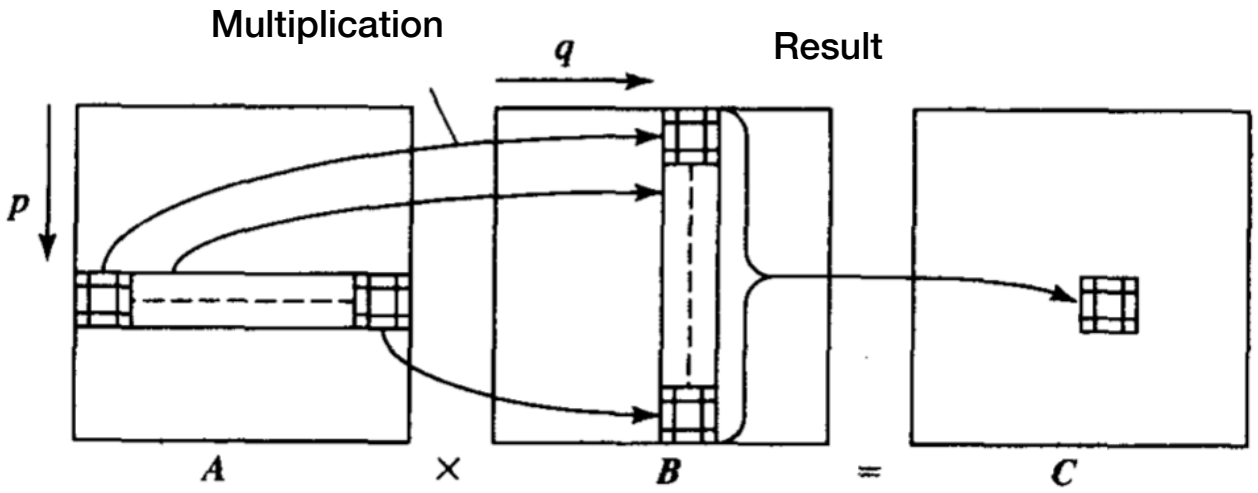


- $C_{i,j} = \sum_{k} A_{i,k} B_{k,j}$

[1] Picture and pseudocode from *Parallel Programming Techniques and Applications Using Networked Workstations and Parallel Computers (2nd Edition)*

Tiling Matrix Multiplication

- Tiling matrix multiplication:



Tiling Matrix Multiplication

- Pseudocode:

```
for (p = 0; p < s; p++)
   for (q = 0; q < s; q++) {
      C_p,q = 0;                          /* clear elements of submatrix */
      for (r = 0; r < m; r++)             /* submatrix multiplication and */
         C_p,q = C_p,q + A_p,r * B_r,q;   /* add to accumulating submatrix */
   }
```

means: product of submatrix $A_{p,r}$, $B_{r,q}$ add submatrix $C_{p,q}$

Compare

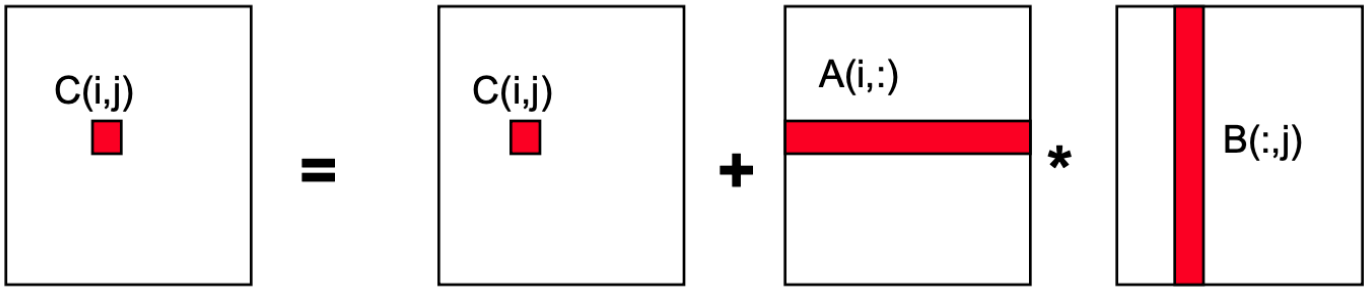
- Naive matrix multiplication:

```
for i = 1 to n
  [read row i of A into fast memory]
  for j = 1 to n
    [read C(i,j) into fast memory]
    [read column j of B into fast memory]
    for k = 1 to n
      C(i,j) = C(i,j) + A(i,k) * B(k,j)
      [write C(i,j) back to slow memory]
```

- n is the size of matrix
- So, $f = 2n^3$

Compare

- Naive matrix multiplication:



- So, $m = n^3 + n^2 + 2n^2 = n^3 + 3n^2$

read each column of B n times

read each row of A once

read and write each element of C once

Compare

- Naive matrix multiplication:
  - $q = f/m = 2n^3/(n^3 + 3n^2)$
  - When n is very large, $q \approx 2$

Compare

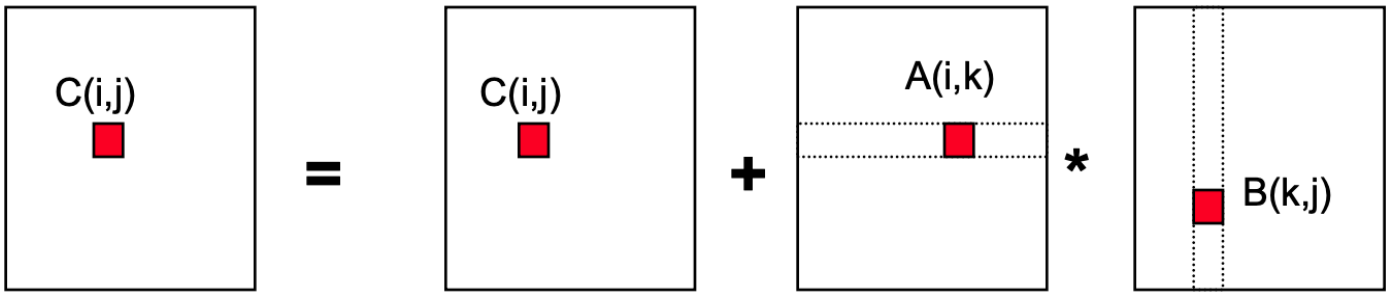
- Tiling matrix multiplication:

```
for i = 1 to N
  for j = 1 to N
    [read block C(i,j) into fast memory]
    for k = 1 to N
      [read block A(i,k) into fast memory]
      [read block B(k,j) into fast memory]
      C(i,j) = C(i,j) + A(i,k) * B(k,j) -> on blocks
    [write block C(i,j) back to slow memory]
```

- n is the size of matrix, N is the number of submatrix on row (or column), $b = n/N$ is called the block size. And $f = 2n^3$

Compare

- Tiling matrix multiplication:



- So, $m = N \times n^2 + N \times n^2 + 2n^2 = (2N + 2) \times n^2$

read each block of B N^3 times,

$N^3 \times b^2 = N^3 3 \times (n/N)^2 = N \times n^2$

read each block of A N^3 times

read and write each block of C once

Compare

- Tiling matrix multiplication:
  - $q = f/m = 2n^3/((2N+2) \times n^2)$
  - When n is very large, $q \approx n/N = b$
- So, increasing the block size b can improve performance

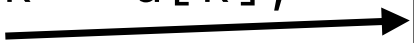
Program

Tiling Program

```
void tiled_gemm(const float *__restrict__ a, size_t lda,
                const float *__restrict__ b, size_t ldb,
                float       *__restrict__ c, size_t ldc,
                int M, int N, int K)
{
    size_t fillsize = sizeof(float) * BLK_N;
    for (int mb = 0; mb < M; mb += BLK_M)
        for (int nb = 0; nb < N; nb += BLK_N) {
            for (int m = 0; m < BLK_M; m++)
                memset(c + mb * ldc + nb, 0, fillsize);
            for (int kb = 0; kb < K; kb += BLK_K) {
                // Do Multiplication on each submatrix
            }
        }
}
```

Multiplication Program

```
static __attribute__((always_inline)) inline
void blk_gemm(const float *__restrict__ a, size_t lda,
              const float *__restrict__ b, size_t ldb,
              float       *__restrict__ c, size_t ldc)
{
    for (int m = 0; m < BLK_M; m++, c += ldc, a += lda)
        for (int k = 0; k < BLK_K; k++) {
            const float *bk = b + k * ldb;
            const float amk = a[k];
#pragma GCC ivdep
            for (int n = 0; n < BLK_N; n++)
                c[n] += amk * bk[n];
        }
}
```



Multiplication Program [manually vectorize]

```
static __attribute__((always_inline)) inline
void blk_gemm_avx(const float *__restrict__ a, size_t lda,
                  const float *__restrict__ b, size_t ldb,
                  float       *__restrict__ c, size_t ldc)
{
    for (int m = 0; m < BLK_M; m++, c += ldc, a += lda)
        for (int k = 0; k < BLK_K; k++) {
            const float *bk = b + k * ldb;
            __m256 amk = _mm256_set1_ps(a[k]);
            for (int n = 0; n < BLK_N; n += 8) {
                __m256 cmn = _mm256_loadu_ps(c + n);
                __m256 bkn = _mm256_loadu_ps(bk + n);
                cmn = _mm256_fmadd_ps(amk, bkn, cmn);
                _mm256_storeu_ps(c + n, cmn);
            }
        }
}
```

use AVX instructions manually

to vectorize

Leading Dimensions

- Parameters lda, ldb, ldc is called leading dimensions which can:
  - give a lot of flexibility so that programmer can work with smaller tiles inside a larger matrix
  - adjust the address alignment of each row
  - avoid cache line conflicts
- And, $lda \geq A_{size}$, $ldb \geq B_{size}$, $ldc \geq C_{size}$
- Example: $lda = 4096 + 32 = 4128, (A_{size} = 4096)$

Performance Evaluation

Compare with Naive GEMM[1]

float GEMM times [ms]												
N	Naive # thread(s)						Tiling # thread(s)[2]					
	1	2	4	8	16	32	1	2	4	8	16	32
512	21.9	16.9	9.7	5.2	3.7	6.6	14.0	9.5	6.5	5.0	4.7	5.1
1024	146.4	78.8	44.9	27.1	18.8	17.5	96.5	54.2	32.4	21.6	18.2	17.1
2048	1785.4	1584.4	1039.0	984.7	499.7	261.3	730.1	369.9	193.3	109.8	83.1	46.3
4096	14117.2	11659.9	8270.4	7182.5	5283.3	3398.2	5634.6	2871.5	1482.8	855.6	487.1	350.6
8192	134330.6	92644.2	68039.2	61682.7	46725.7	34160.2	45421.6	22759.0	11592.5	6090.3	3433.1	2177.4
16384	1037196.	788163.5	468816.1	508940.2	351270.6	319852.6	366541.7	183905.7	93005.1	48172.2	26391.1	16009.8

[1] Compile with options: -mavx -mavx2 -mfma -funroll-loops
[2] Block size is 64

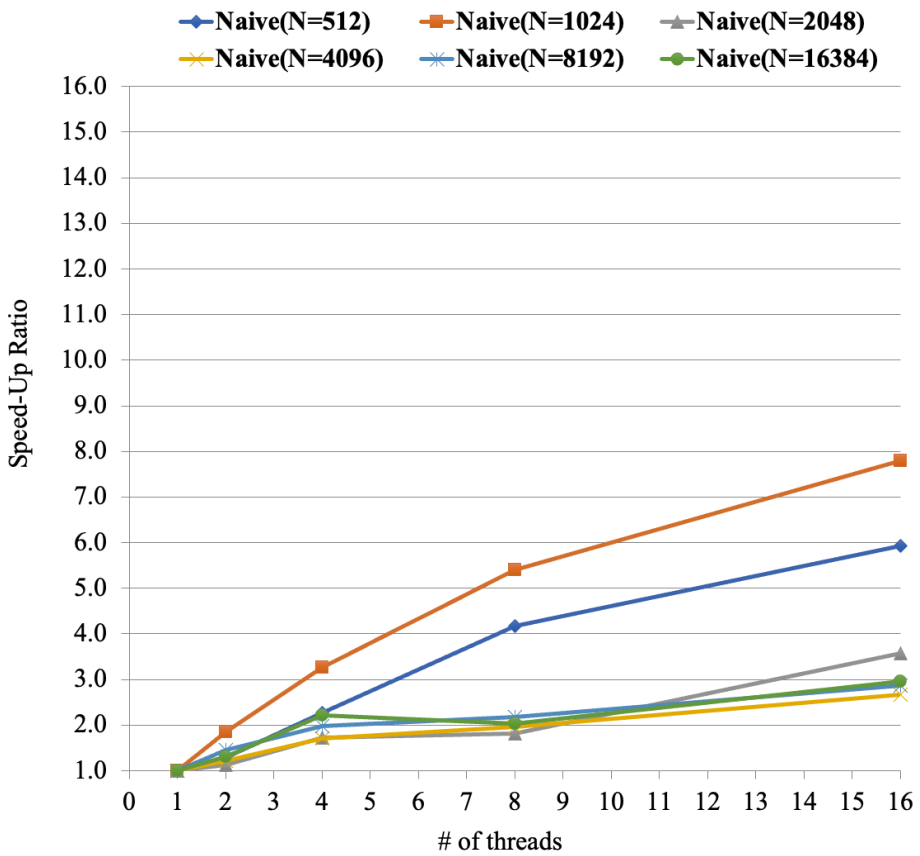
Compare with Naive GEMM

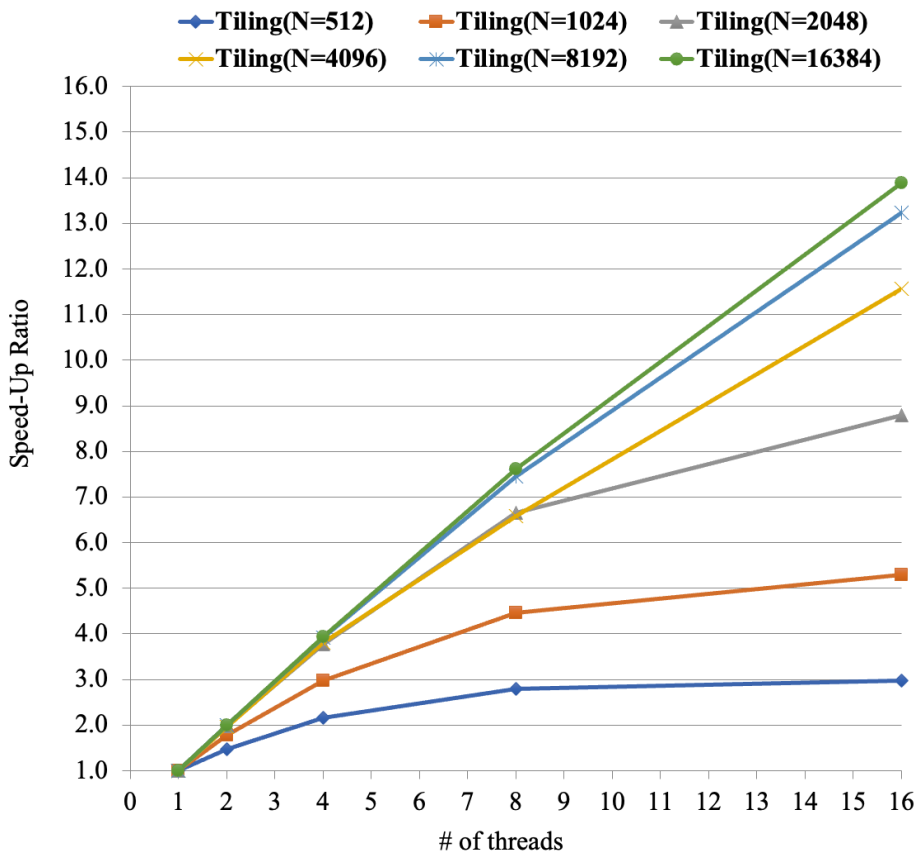
Speedup												
N	Naive # thread(s)						Tiling # thread(s)					
	1	2	4	8	16	32	1	2	4	8	16	32
512	1.0	1.3	2.3	4.2	5.9	3.3	1.0	1.5	2.2	2.8	3.0	2.8
1024	1.0	1.9	3.3	5.4	7.8	8.4	1.0	1.8	3.0	4.5	5.3	5.7
2048	1.0	1.1	1.7	1.8	3.6	6.8	1.0	2.0	3.8	6.7	8.8	15.8
4096	1.0	1.2	1.7	2.0	2.7	4.2	1.0	2.0	3.8	6.6	11.6	16.1
8192	1.0	1.4	2.0	2.2	2.9	3.9	1.0	2.0	3.9	7.5	13.2	20.9
16384	1.0	1.3	2.2	2.0	3.0	3.2	1.0	2.0	3.9	7.6	13.9	22.9

Compare with Naive GEMM

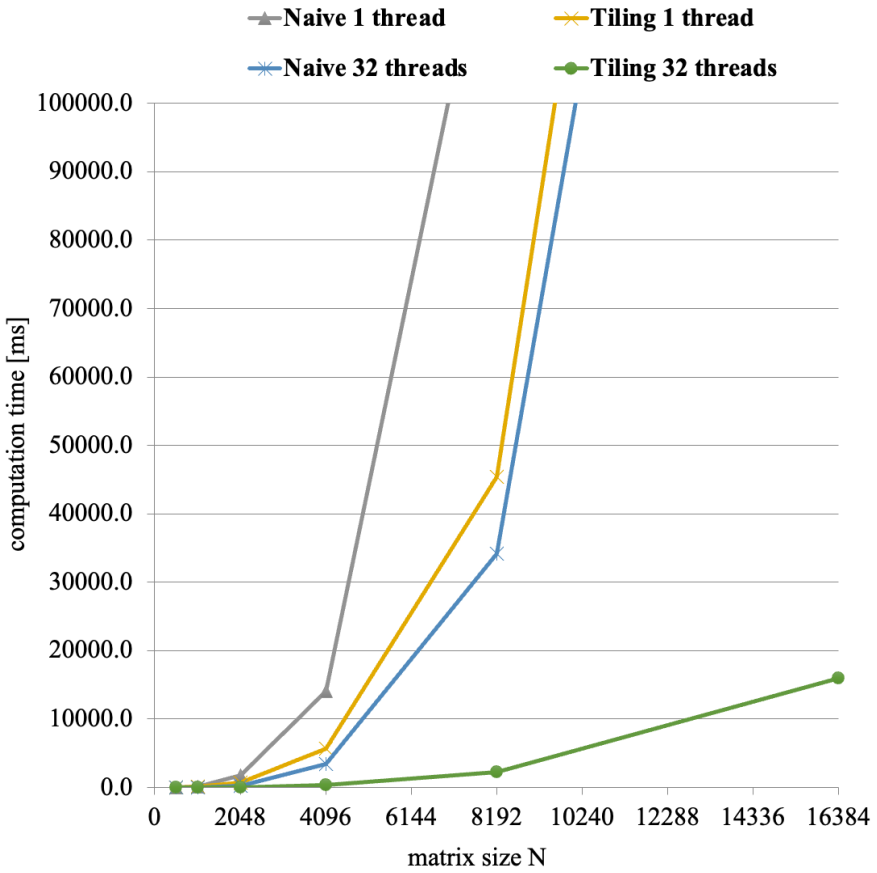
Throughput [GOP/s]													
N	Naive # thread(s)						Tiling # thread(s)						
	1	2	4	8	16	32	1	2	4	8	16	32
512	12	16	28	51	73	40	19	28	41	53	57	53
1024	15	27	48	79	114	123	22	40	66	99	118	126
2048	10	11	17	17	34	66	24	46	89	156	207	370
4096	10	12	17	19	26	40	24	48	93	161	282	392
8192	8	12	16	18	24	32	24	48	95	181	320	505
16384	8	11	19	17	25	27	24	48	95	183	333	549

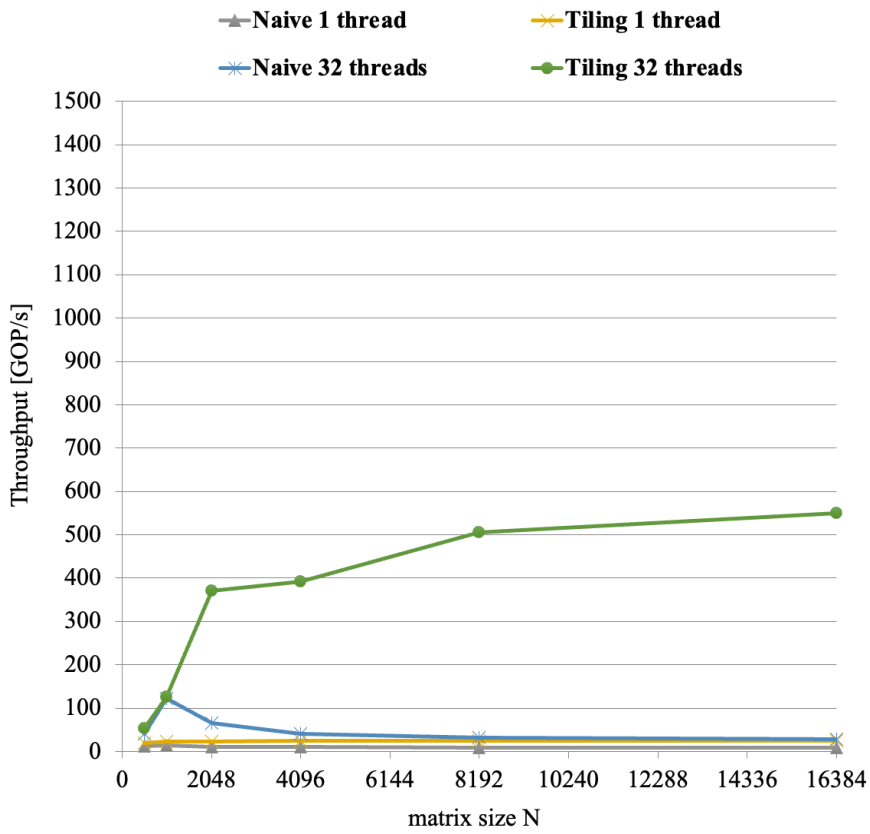
Compare with Naive GEMM





Compare with Naive GEMM





Compare with Naive GEMM

- Tiling can improve the performance of matrix multiplication program
- And the performance improvement is very huge

Leading Dimensions[1]

float GEMM times [ms]													
N	Tiling # thread(s) with LD						Tiling # thread(s) without LD						
	1	2	4	8	16	32	1	2	4	8	16	32
512	14.0	9.5	6.5	5.0	4.7	5.1	14.3	10.6	5.9	4.5	5.6	6.4
1024	96.5	54.2	32.4	21.6	18.2	17.1	89.8	52.3	31.5	23.2	16.5	16.3
2048	730.1	369.9	193.3	109.8	83.1	46.3	704.1	367.3	193.3	116.3	77.7	44.9
4096	5634.6	2871.5	1482.8	855.6	487.1	350.6	5665.1	2816.5	1467.5	815.5	523.1	505.1
8192	45421.6	22759.0	11592.5	6090.3	3433.1	2177.4	44480.3	22643.2	11514.2	6151.9	3650.4	2895.5
16384	366541.7	183905.7	93005.1	48172.2	26391.1	16009.8	359113.7	179784.7	91564.8	48216.2	26877.2	18780.8

[1] Compile with options: -mavx -mavx2 -mfma -funroll-loops, block size is 64

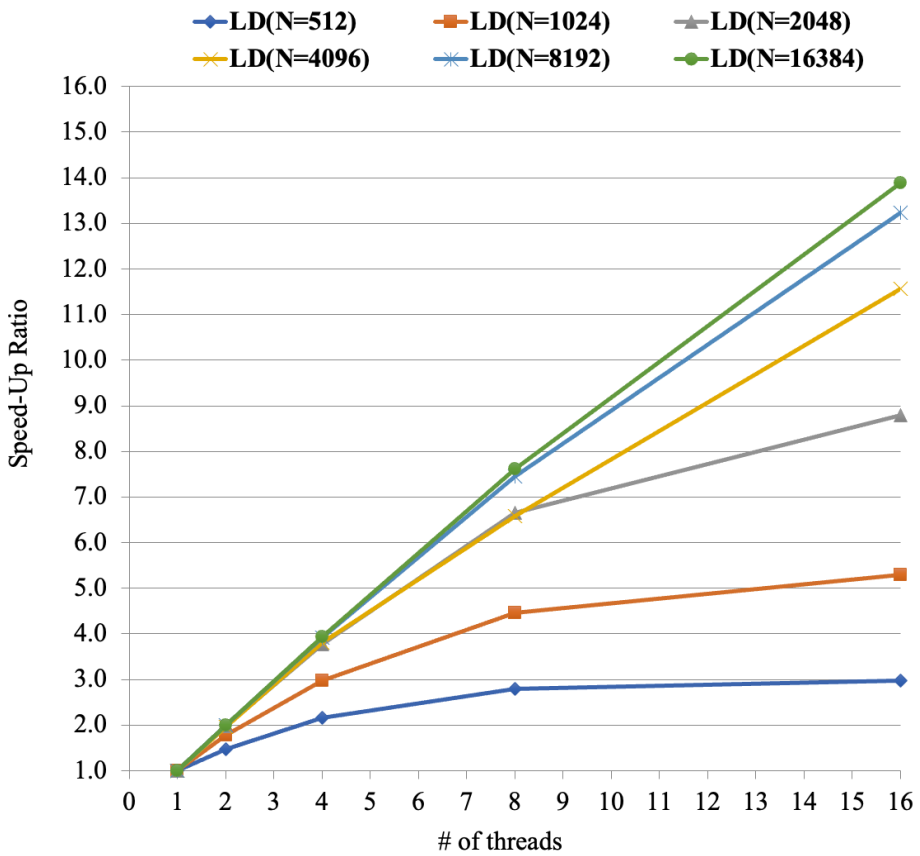
Leading Dimensions

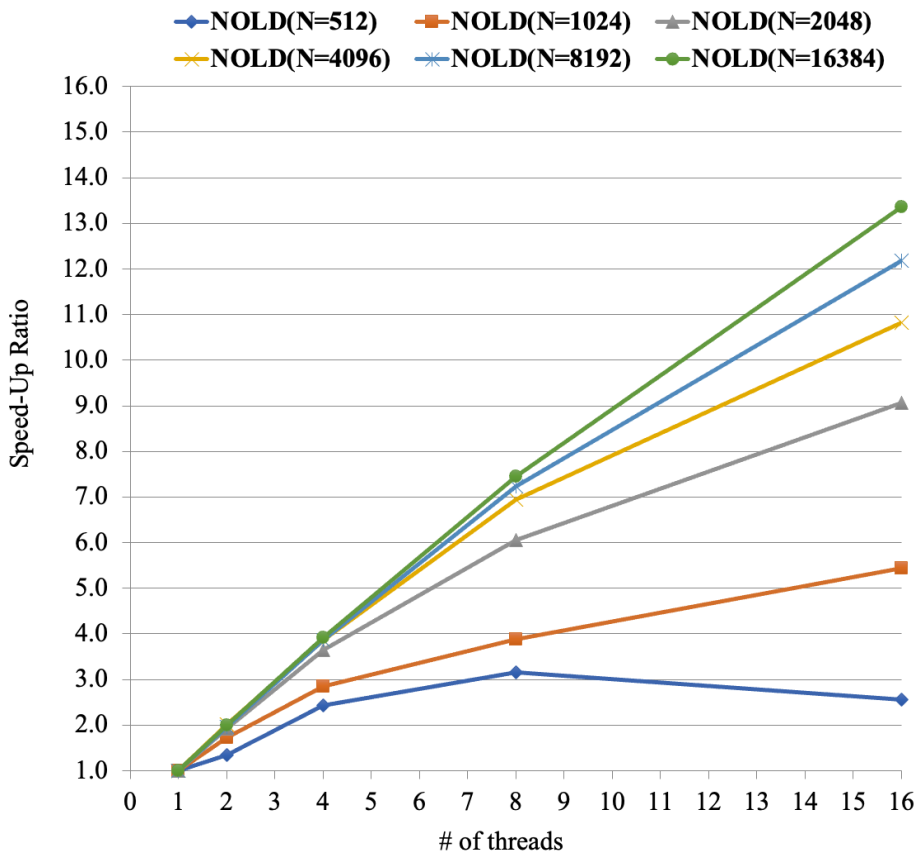
Speedup												
N	Tiling # thread(s) with LD						Tiling # thread(s) without LD					
	1	2	4	8	16	32	1	2	4	8	16	32
512	1.0	1.5	2.2	2.8	3.0	2.8	1.0	1.3	2.4	3.2	2.6	2.2
1024	1.0	1.8	3.0	4.5	5.3	5.7	1.0	1.7	2.9	3.9	5.4	5.5
2048	1.0	2.0	3.8	6.7	8.8	15.8	1.0	1.9	3.6	6.1	9.1	15.7
4096	1.0	2.0	3.8	6.6	11.6	16.1	1.0	2.0	3.9	6.9	10.8	11.2
8192	1.0	2.0	3.9	7.5	13.2	20.9	1.0	2.0	3.9	7.2	12.2	15.4
16384	1.0	2.0	3.9	7.6	13.9	22.9	1.0	2.0	3.9	7.4	13.4	19.1

Leading Dimensions

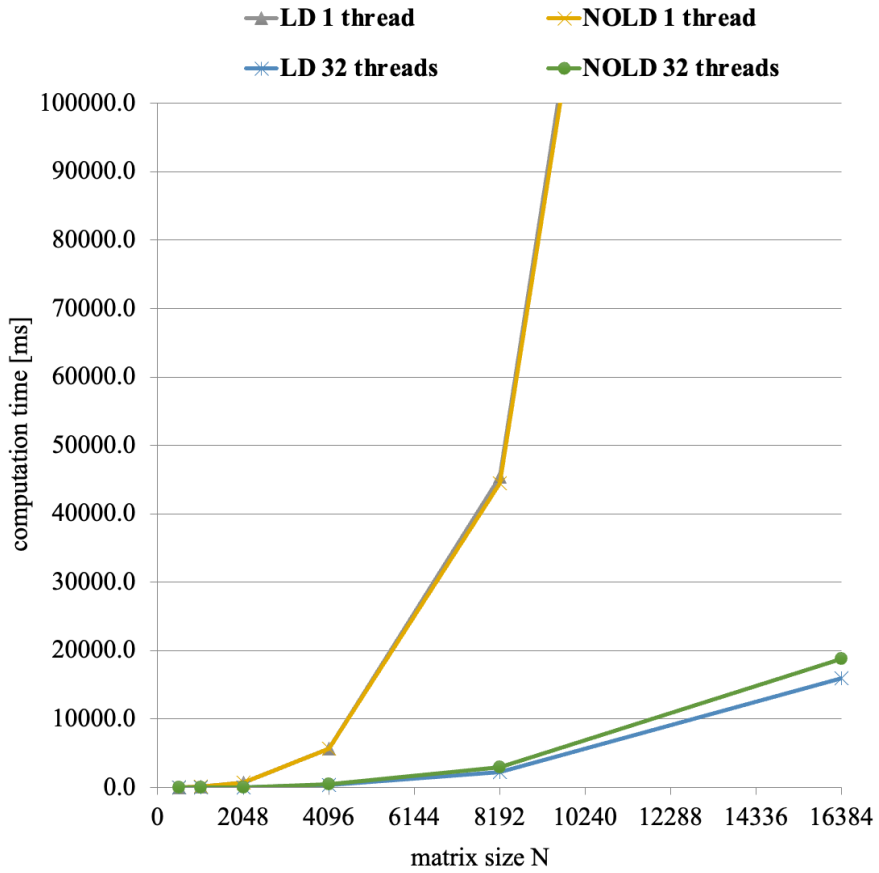
Throughput [GOP/s]													
N	Tiling # thread(s) with LD						Tiling # thread(s) without LD						
	1	2	4	8	16	32	1	2	4	8	16	32
512	19	28	41	53	57	53	19	25	45	59	48	42
1024	22	40	66	99	118	126	24	41	68	93	130	131
2048	24	46	89	156	207	370	24	47	89	148	221	382
4096	24	48	93	161	282	392	24	49	94	169	263	272
8192	24	48	95	181	320	505	25	49	95	179	301	380
16384	24	48	95	183	333	549	24	49	96	182	327	468

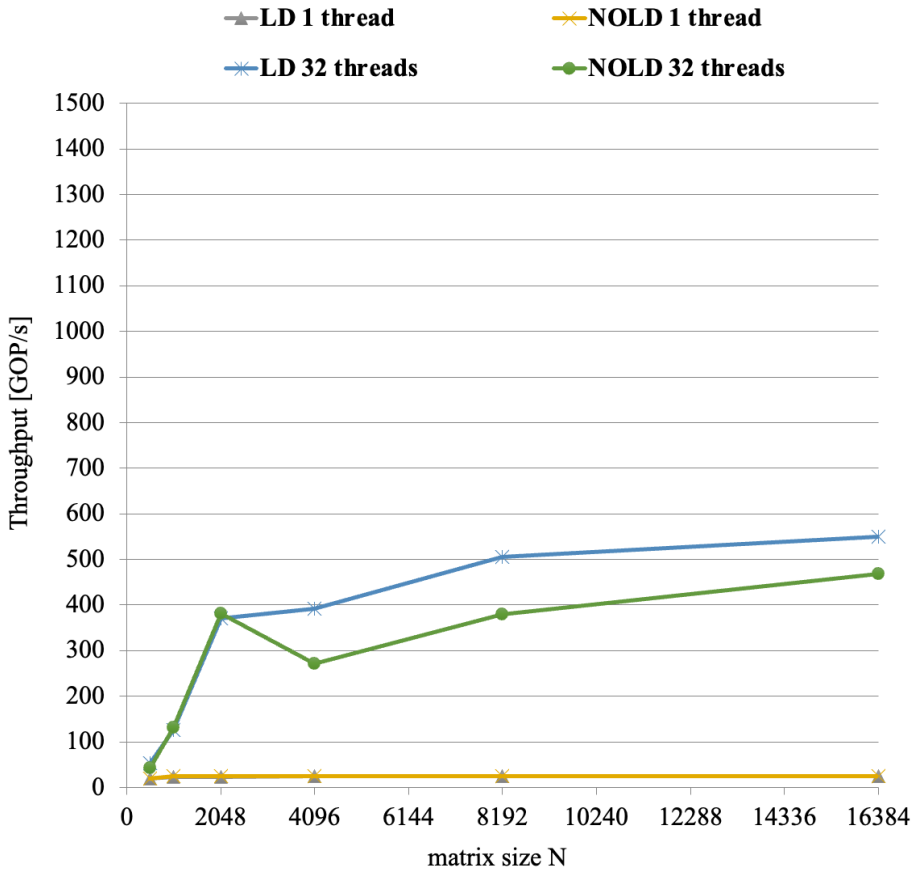
Leading Dimensions





Leading Dimensions





Leading Dimensions

- Leading dimensions can make a program performance improving when:
  - the size of matrix is big
  - the program has lots of threads

Compare with OpenBLAS[1]

- OpenBLAS is an optimized **B**asic **L**inear **A**lgebra **S**ubprograms library

float GEMM times [ms]													
N	Tiling # thread(s)[2]						OpenBLAS # thread(s)						
	1	2	4	8	16	32	1	2	4	8	16	32
512	14.0	9.5	6.5	5.0	4.7	5.1	14.5	17.3	6.6	8.2	9.2	11.8
1024	96.5	54.2	32.4	21.6	18.2	17.1	61.5	58.2	27.3	23.9	18.5	17.9
2048	730.1	369.9	193.3	109.8	83.1	46.3	362.0	192.8	110.7	59.5	51.2	47.6
4096	5634.6	2871.5	1482.8	855.6	487.1	350.6	2303.9	1235.4	673.2	361.6	203.8	125.4
8192	45421.6	22759.0	11592.5	6090.3	3433.1	2177.4	17951.3	9322.5	4835.7	2516.6	1439.7	810.6
16384	366541.7	183905.7	93005.1	48172.2	26391.1	16009.8	142728.5	73427.9	37553.3	19562.5	10657.5	6407.3

[1] Compile with options: -mavx -mavx2 -mfma -funroll-loops
[2] Block size is 64

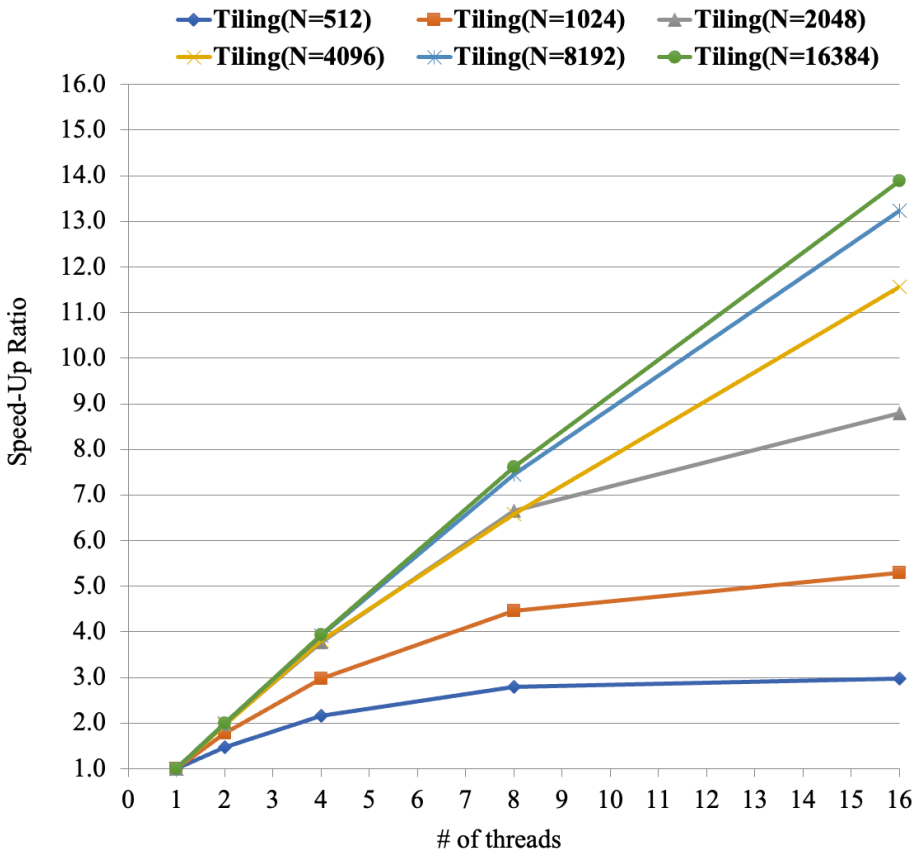
Compare with OpenBLAS

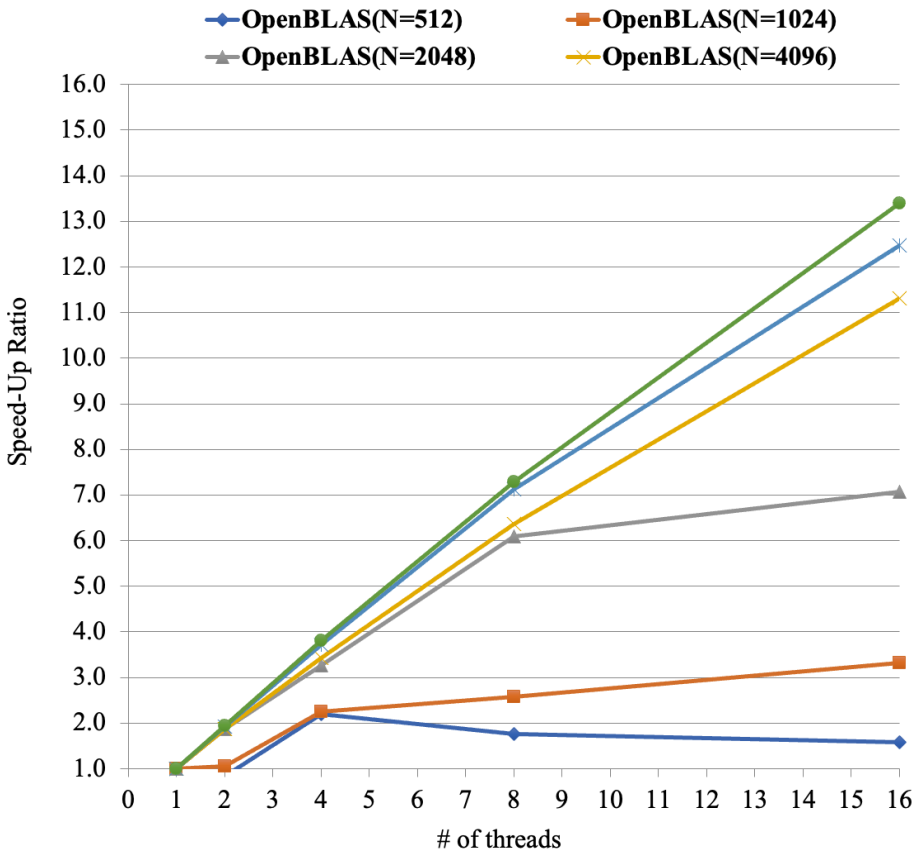
Speedup												
N	Tiling # thread(s)						OpenBLAS # thread(s)					
	1	2	4	8	16	32	1	2	4	8	16	32
512	1.0	1.5	2.2	2.8	3.0	2.8	1.0	0.8	2.2	1.8	1.6	1.2
1024	1.0	1.8	3.0	4.5	5.3	5.7	1.0	1.1	2.3	2.6	3.3	3.4
2048	1.0	2.0	3.8	6.7	8.8	15.8	1.0	1.9	3.3	6.1	7.1	7.6
4096	1.0	2.0	3.8	6.6	11.6	16.1	1.0	1.9	3.4	6.4	11.3	18.4
8192	1.0	2.0	3.9	7.5	13.2	20.9	1.0	1.9	3.7	7.1	12.5	22.1
16384	1.0	2.0	3.9	7.6	13.9	22.9	1.0	1.9	3.8	7.3	13.4	22.3

Compare with OpenBLAS

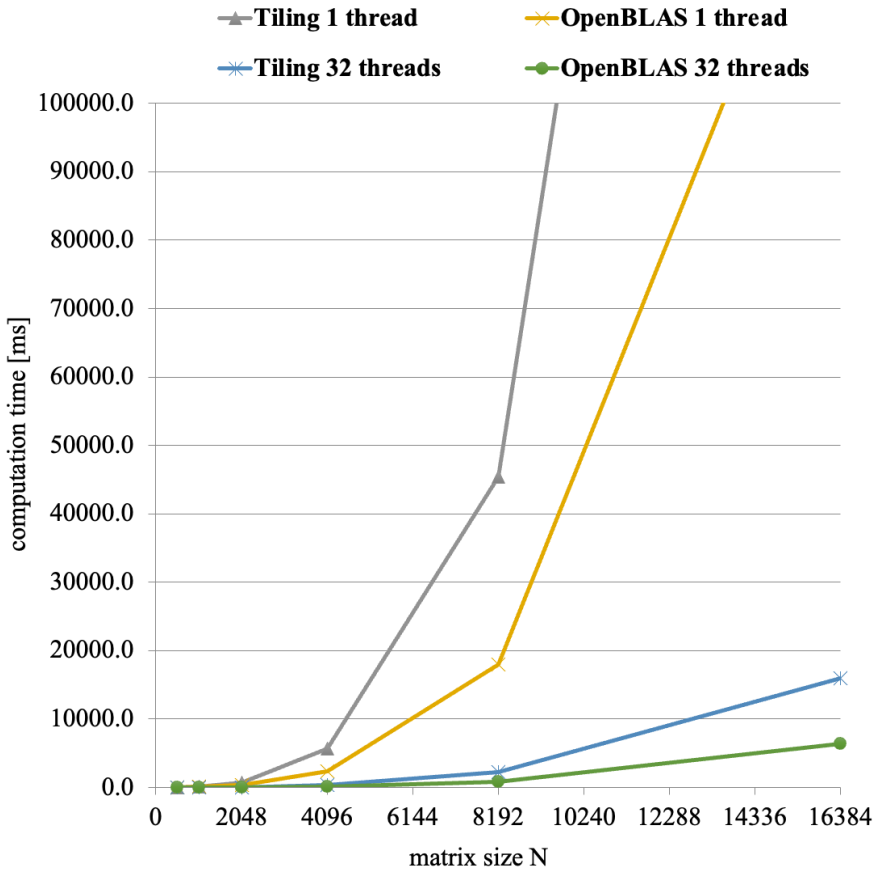
Throughput [GOP/s]												
N	Tiling # thread(s)						OpenBLAS # thread(s)					
	1	2	4	8	16	32	1	2	4	8	16	32
512	19	28	41	53	57	53	19	15	41	33	29	23
1024	22	40	66	99	118	126	35	37	79	90	116	120
2048	24	46	89	156	207	370	47	89	155	289	336	360
4096	24	48	93	161	282	392	60	111	204	380	674	1096
8192	24	48	95	181	320	505	61	118	227	437	764	1356
16384	24	48	95	183	333	549	62	120	234	450	825	1373

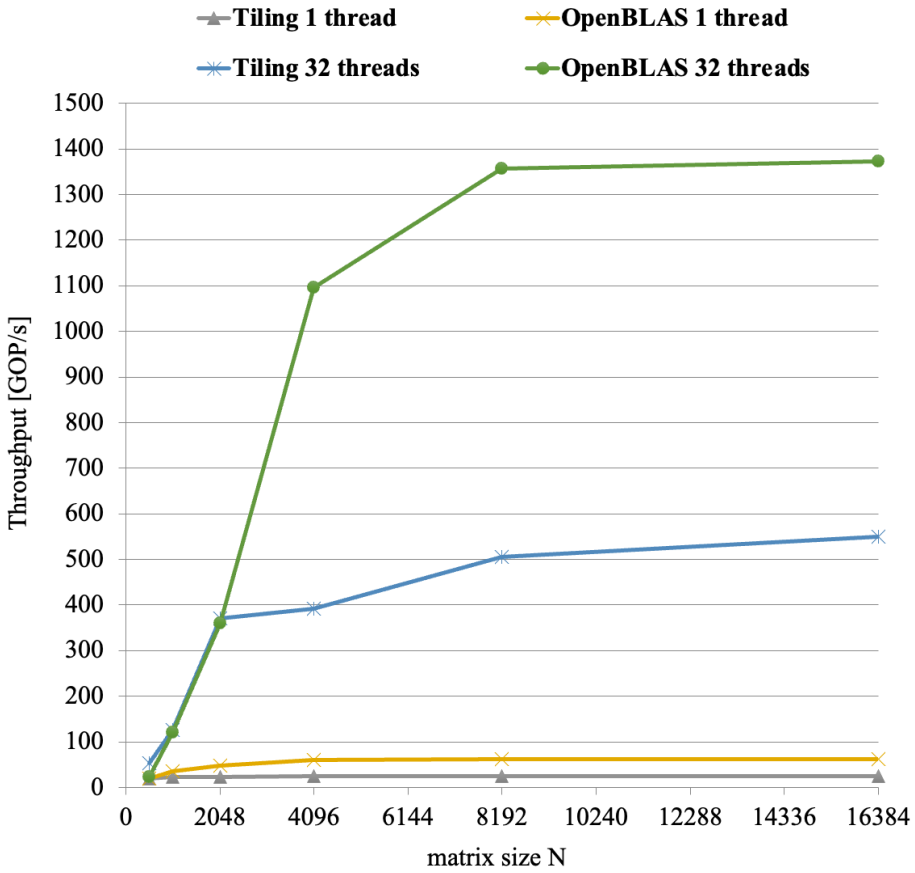
Compare with OpenBLAS





Compare with OpenBLAS





Compare with OpenBLAS

- The GEMM of OpenBLAS has a very high throughput
  - 31.9% of theoretical performance, when CPU clock is 4.2GHz
  - 44.7% of theoretical performance, when CPU clock is 3GHz
- But, using SIMD instructions will cause the CPU downclocks from the base clock
- SO, when using SIMD instructions, the clock frequency will never be the MAX value (4.2Ghz) and the base value (3GHz)
- SO, OpenBLAS's GEMM has a performance (maybe) up to 50% of theoretical performance (or more)

Other Performance Evaluation

- There are some necessary performance evaluations going on:
    1. Different SIMD instructions (SSE, AVX256, AVX512)
    2. Different block size
    3. When submatrix is not a square matrix
    4. Other way to improve the performance of GEMM

Research Plan

Research Plan

- Previous research plan:
  - Developing a parallel programming language for embedded devices
  - But may not be completed at graduate level
  - It might be a good idea to focus on one point, the parallel programming on embedded devices

Research Plan

- New research plan:
  - On Raspberry Pi 4b, there is a 48GFLOPS CPU and a 32GFLOPS QPU (VideoCore6, or VC6)
  - VC6 is a 4way-SIMD processor
  - So, focus on:
    1. Can the performance of OpenBLAS be improved for embedded devices, or
    2. Developing a C language GPGPU library for Raspberry Pi, like *py-videocore6* (a python GPGPU library for Raspberry Pi)

References

References

[1] Bryant, R., & O'Hallaron, D. Computer systems (pp. 679-683).

[2] Wilkinson, B. (2005). Parallel Programming. Prentice Hall.

[3] (2022). Retrieved 17 July 2022, from https://sites.cs.ucsb.edu/~tyang/class/240a17/slides/Cache3.pdf

[4] Sornet, G., Jubertie, S., Dupros, F., De Martin, F., & Limet, S. (2018, July). Performance analysis of SIMD vectorization of high-order finite-element kernels. In *2018 International Conference on High Performance Computing & Simulation (HPCS)* (pp. 423-430). IEEE.

References

[5] GitHub - xianyi/OpenBLAS: OpenBLAS is an optimized BLAS library based on GotoBLAS2 1.13 BSD version. (2022). Retrieved 18 July 2022, from https://github.com/xianyi/OpenBLAS

[6] Raspberry Pi4のGPGPUに挑戦（その１） - あざらしなので. (2022). Retrieved 18 July 2022, from https://taiki-azrs.hatenablog.com/entry/2021/05/18/190052

[7] GitHub - Idein/py-videocore6: Python library for GPGPU programming on Raspberry Pi 4. (2022). Retrieved 18 July 2022, from https://github.com/Idein/py-videocore6