Non-Negative Matrix Factorization

Daniele Coppola
Viktor Gsteiger
Maša Nešić
Matteo Oldani

ETH
Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

The Algorithm

$$V = W \times H$$

$$H_{[i,j]}^{n+1} = H_{[i,j]}^{n} \cdot \frac{((W^n)^T\, V)_{[i,j]}}{((W^n)^T\, W^n\, H^n)_{[i,j]}}$$

$$W_{[i,j]}^{n+1} = W_{[i,j]}^{n} \cdot \frac{(V\, (H^{n+1})^T)_{[i,j]}}{(W^n\, H^{n+1}\, (H^{n+1})^T)_{[i,j]}}$$

Cost analysis

- **Asymptotic runtime complexity :**

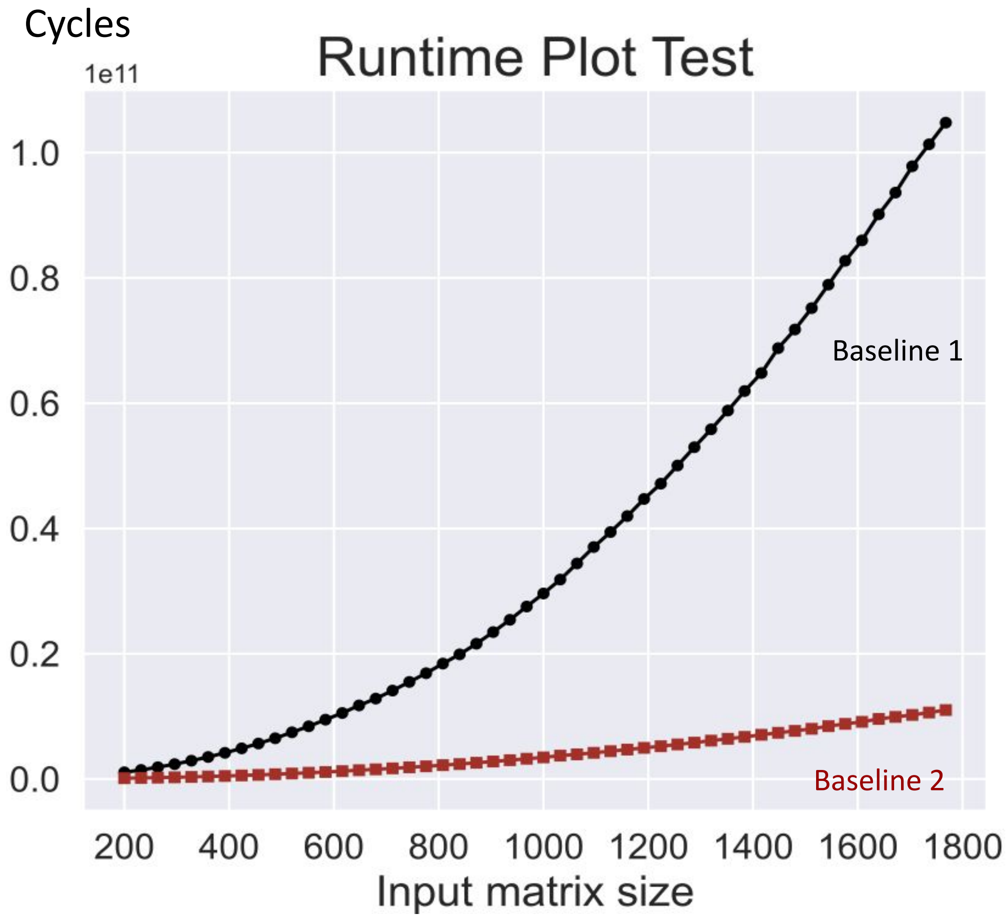
$$O(m \cdot n \cdot r + m \cdot r^2 + n \cdot r^2 + n^2 \cdot r + \ m \cdot n)$$

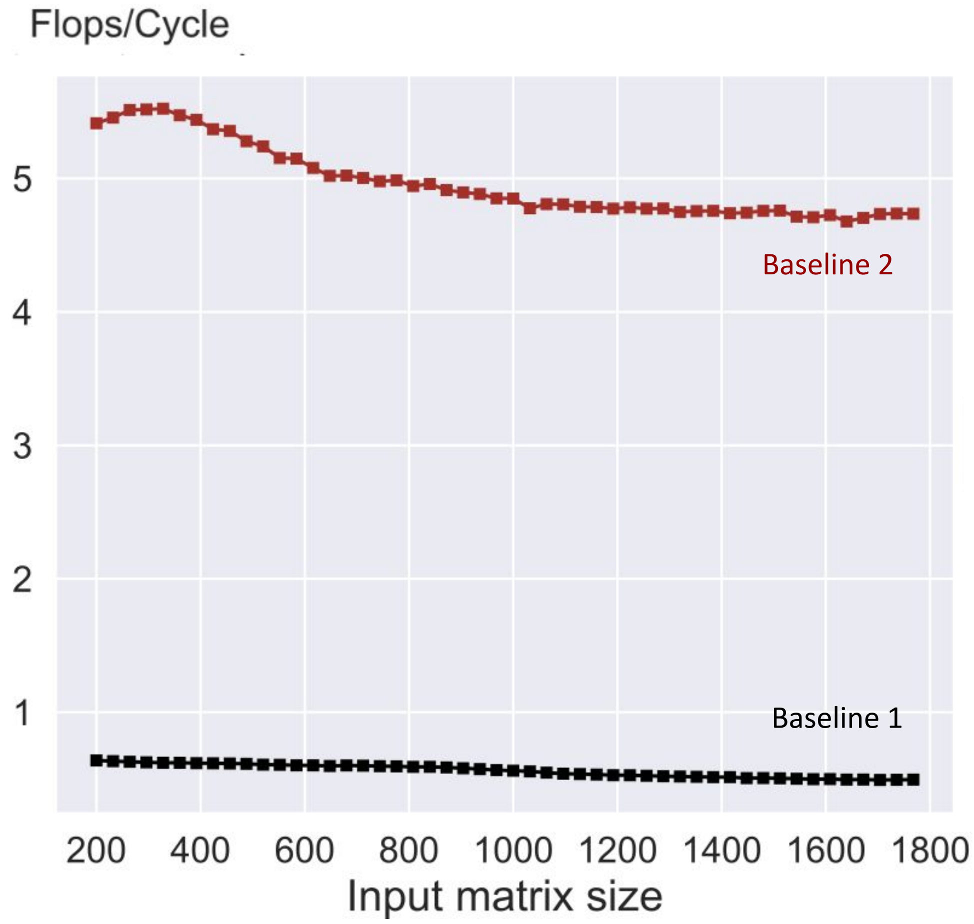
$$V \ : m \times n$$
$$W : m \times r$$
$$H : r \times n$$

- **Cost - measured by counting operations inside the code:**

```
2 * m * n * r + 5 * m * n + 3 +
number_of_iterations * (6 * m * n * r +
                        3 * m * r * r +
                        3 * n * r * r +
                        5 * m * n +
                        2 * m * r +
                        2 * n * r + 3)
```

Baseline Implementations

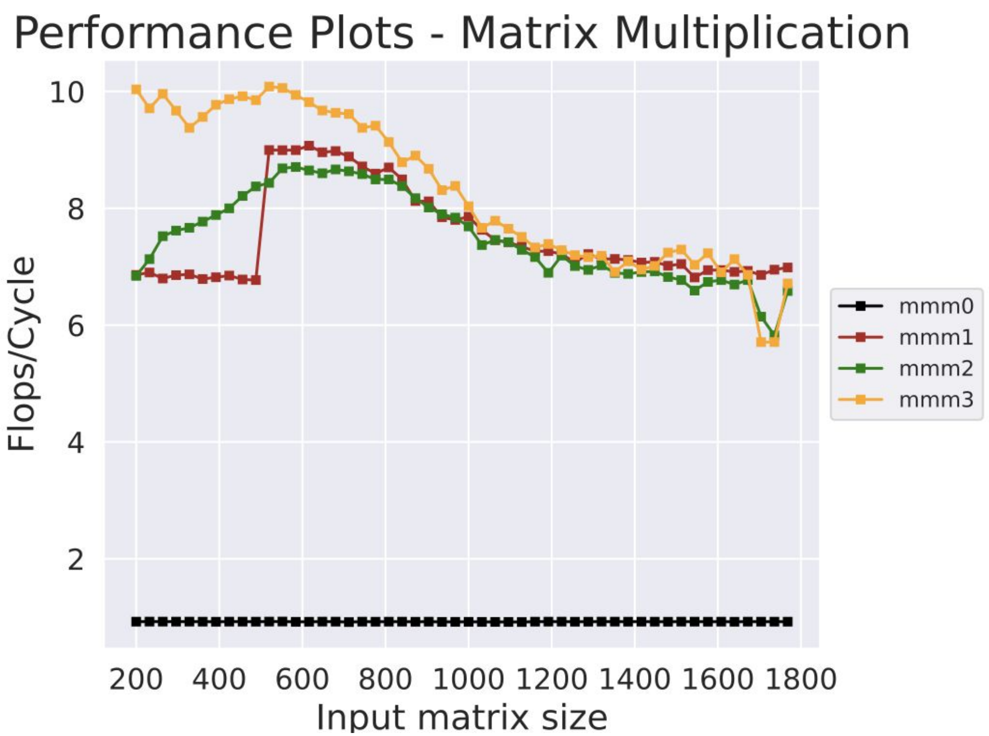




NNMF (double precision) on i5-6600K, 3.5 GHz, r = 16

Matrix Multiplication Optimizations

- Blocking (block size = 16)
- Loop unrolling to have 8 independent lines of computations
- Vectorization of the computation



NNMF (double precision) on i5-6600K, 3.5 GHz, r = 16

```
int idx_b = k * B_n_col + jj;
for (kk = k; kk < k + nB; kk++) {
    a0 = _mm256_set1_pd(A[Aii + kk]);
    a1 = _mm256_set1_pd(A[Aii + A_n_col + kk]);

    b0 = _mm256_load_pd((double *) &B[idx_b]);
    b1 = _mm256_load_pd((double *) &B[idx_b + 4]);
    b2 = _mm256_load_pd((double *) &B[idx_b + 8]);
    b3 = _mm256_load_pd((double *) &B[idx_b + 12]);

    r0 = _mm256_fmadd_pd(a0, b0, r0);
    r1 = _mm256_fmadd_pd(a0, b1, r1);
    r2 = _mm256_fmadd_pd(a0, b2, r2);
    r3 = _mm256_fmadd_pd(a0, b3, r3);

    r4 = _mm256_fmadd_pd(a1, b0, r4);
    r5 = _mm256_fmadd_pd(a1, b1, r5);
    r6 = _mm256_fmadd_pd(a1, b2, r6);
    r7 = _mm256_fmadd_pd(a1, b3, r7);

    idx_b += B_n_col;
}
```

Matrix Padding

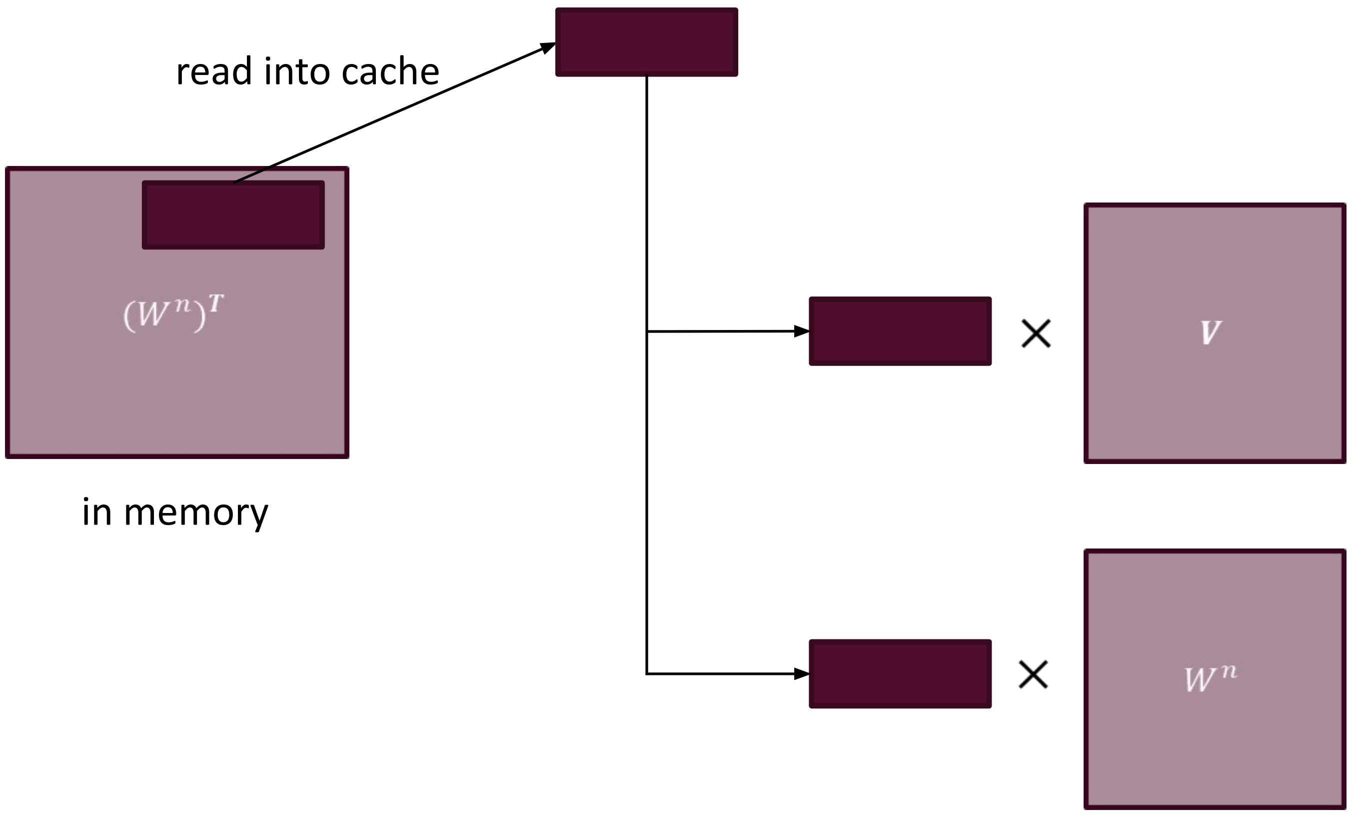
- **R multiple of block size**
  - Outperforming BLAS
- **R not multiple of block size**
  - Blas outperforming us
- **Matrix padding to the block size**
  - More computation but better microarchitecture usage
  - Outperforming BLAS
  - Padding with 0s

Algorithmic optimization 1 – Reuse entries of W

- **Reduces the number of read accesses to $(W^n)^T$ in the computation of H^{n+1}, and to $(H^{n+1})^T$ in the computation of W^{n+1} by half**

$$H^{n+1}_{[i,j]} = H^n_{[i,j]} \cdot \frac{((W^n)^T\, V)_{[i,j]}}{((W^n)^T\, W^n\, H^n)_{[i,j]}}$$

$$W^{n+1}_{[i,j]} = W^n_{[i,j]} \cdot \frac{(V\, (H^{n+1})^T)_{[i,j]}}{(W^n\, H^{n+1}\, (H^{n+1})^T)_{[i,j]}}$$



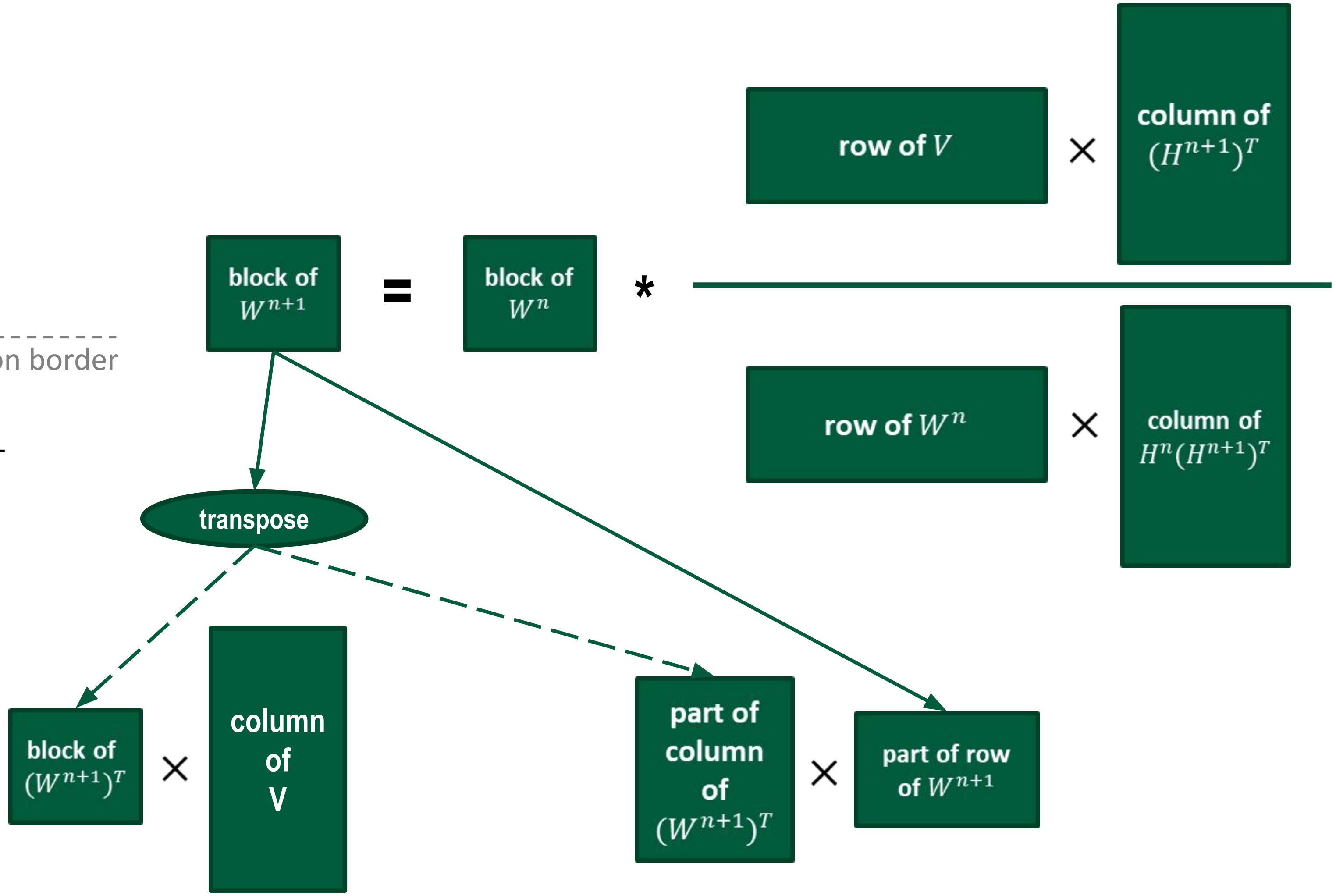
Algorithmic optimization 4 – Reuse block of W across 2 iterations

- **The calculated block of W^{n+1} is immediately used in the calculation of $(W^{n+1})^T V$ and $(W^{n+1})^T W^{n+1}$**

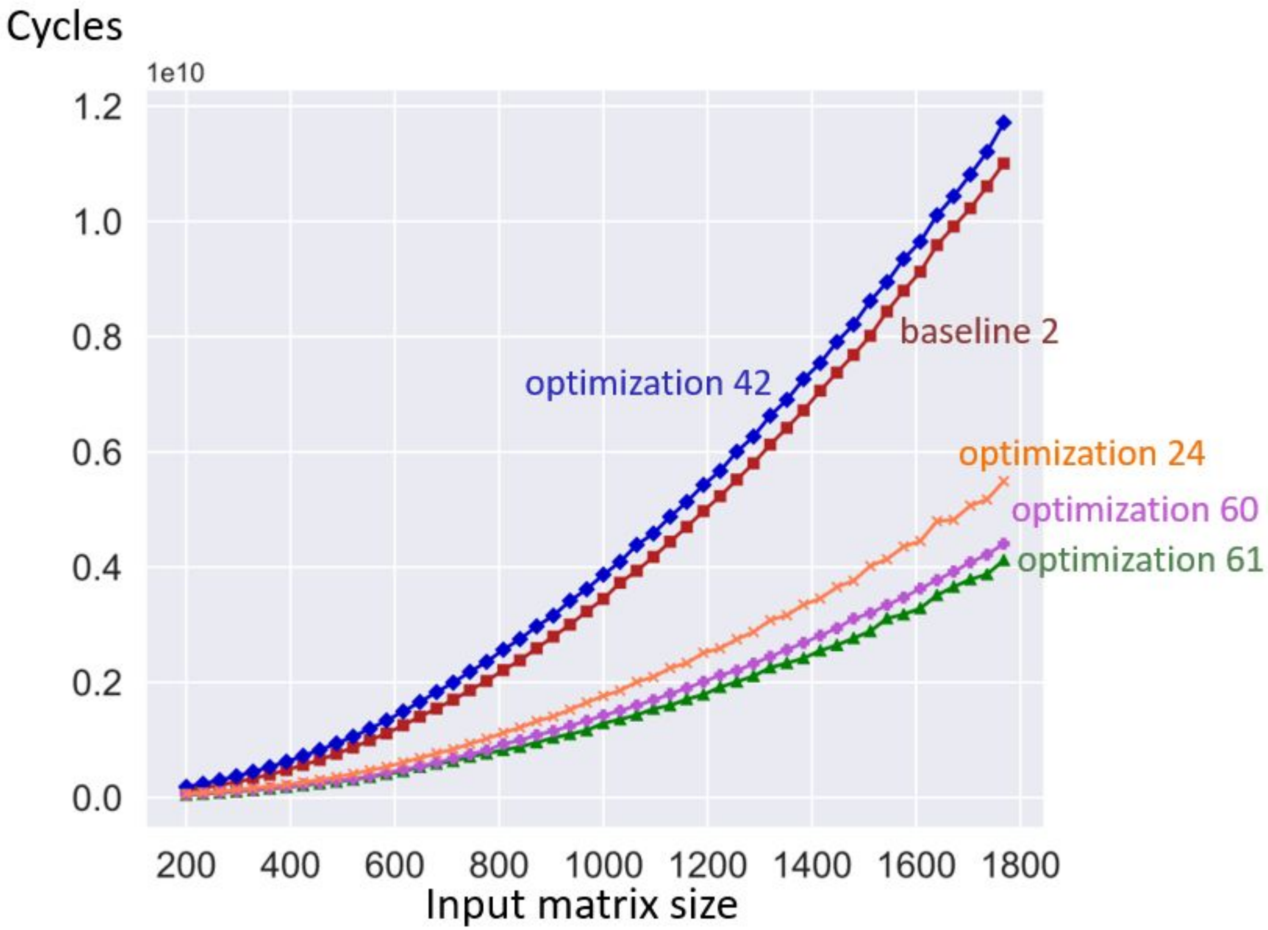
$$\boxed{W_{[i,j]}^{n+1}} = W_{[i,j]}^{n} \cdot \frac{(V\,(H^{n+1})^T)_{[i,j]}}{(W^n\,H^{n+1}\,(H^{n+1})^T)_{[i,j]}}$$

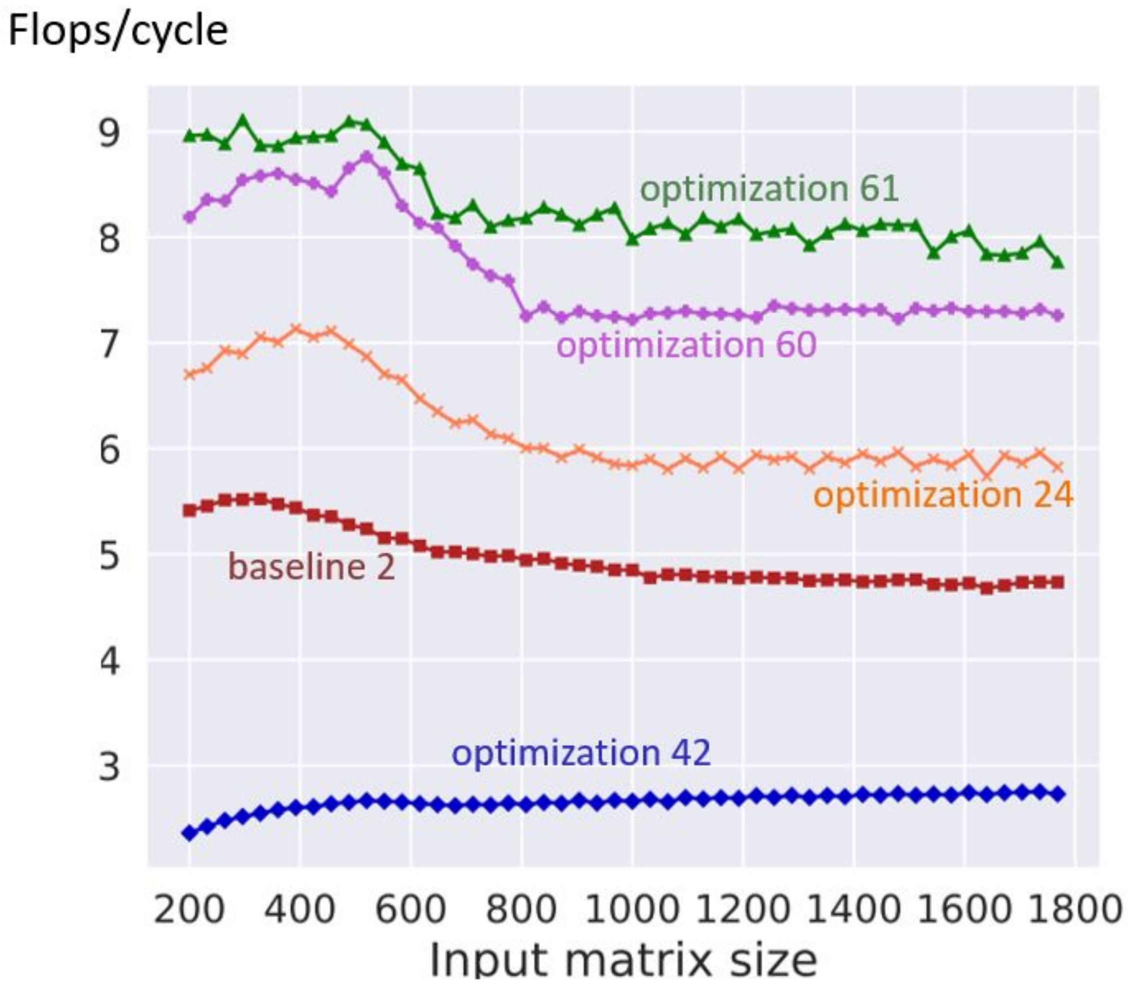
iteration border

$$H_{[i,j]}^{n+2} = H_{[i,j]}^{n+1} \cdot \frac{(\boxed{(W^{n+1})^T\,V})_{[i,j]}}{(\boxed{(W^{n+1})^T\,W^{n+1}}\,H^{n+1})_{[i,j]}}$$



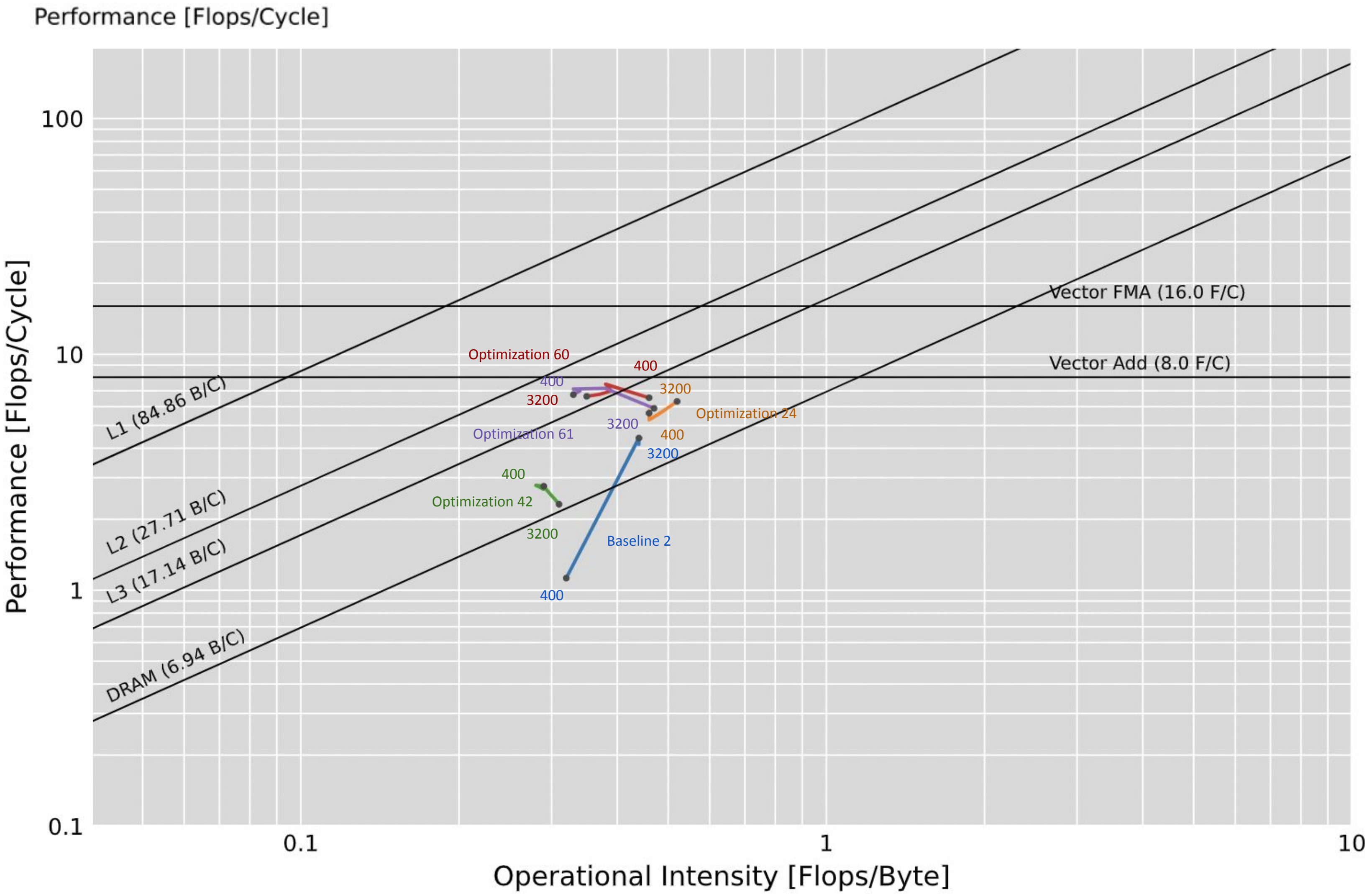
Runtime and Performance Plots





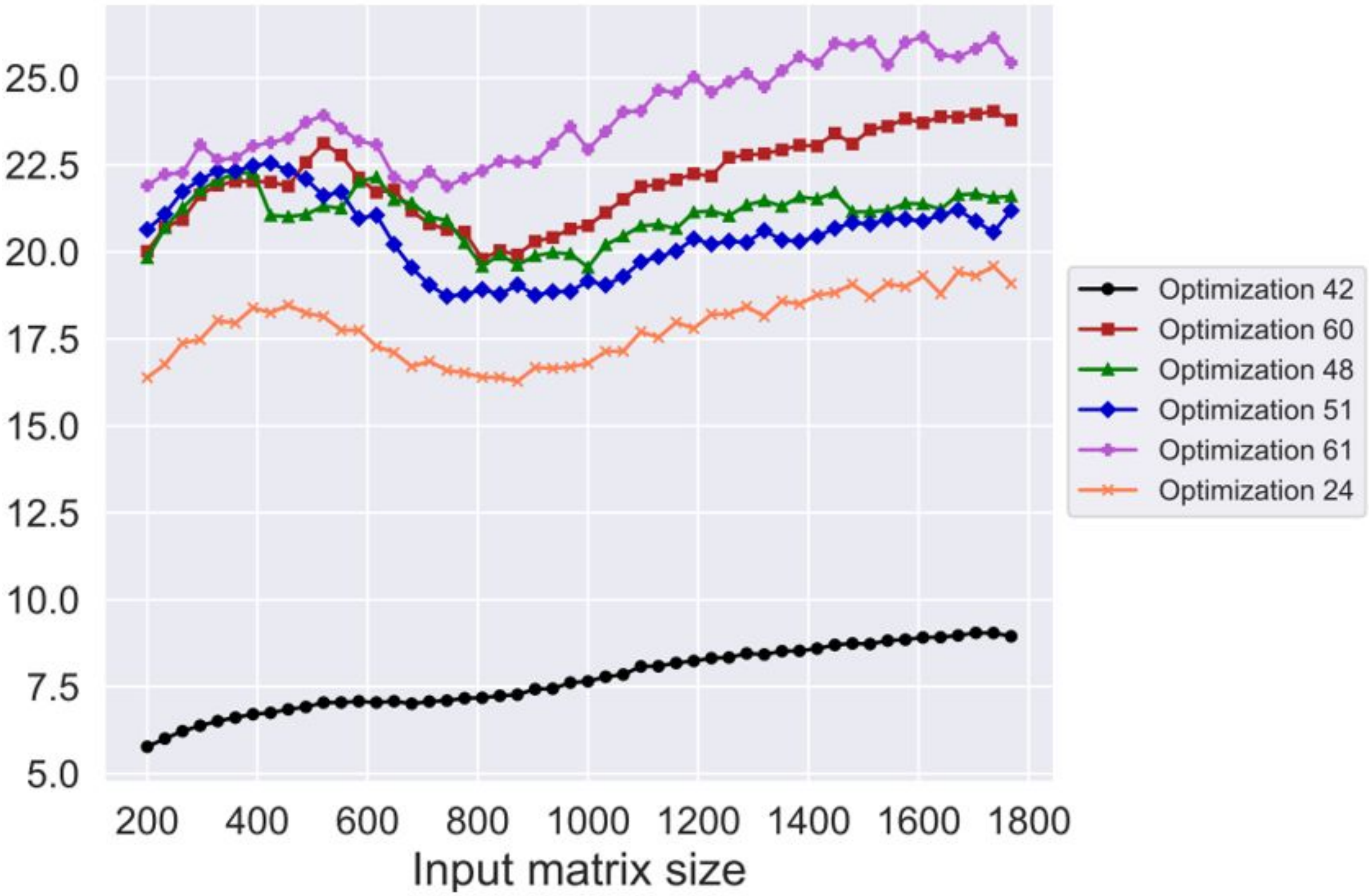
NNMF (double precision) on i5-6600K, 3.5 GHz, r = 16

Roofline Plot



NNMF (double precision) on i5-6600K, 3.5 GHz, r = 16

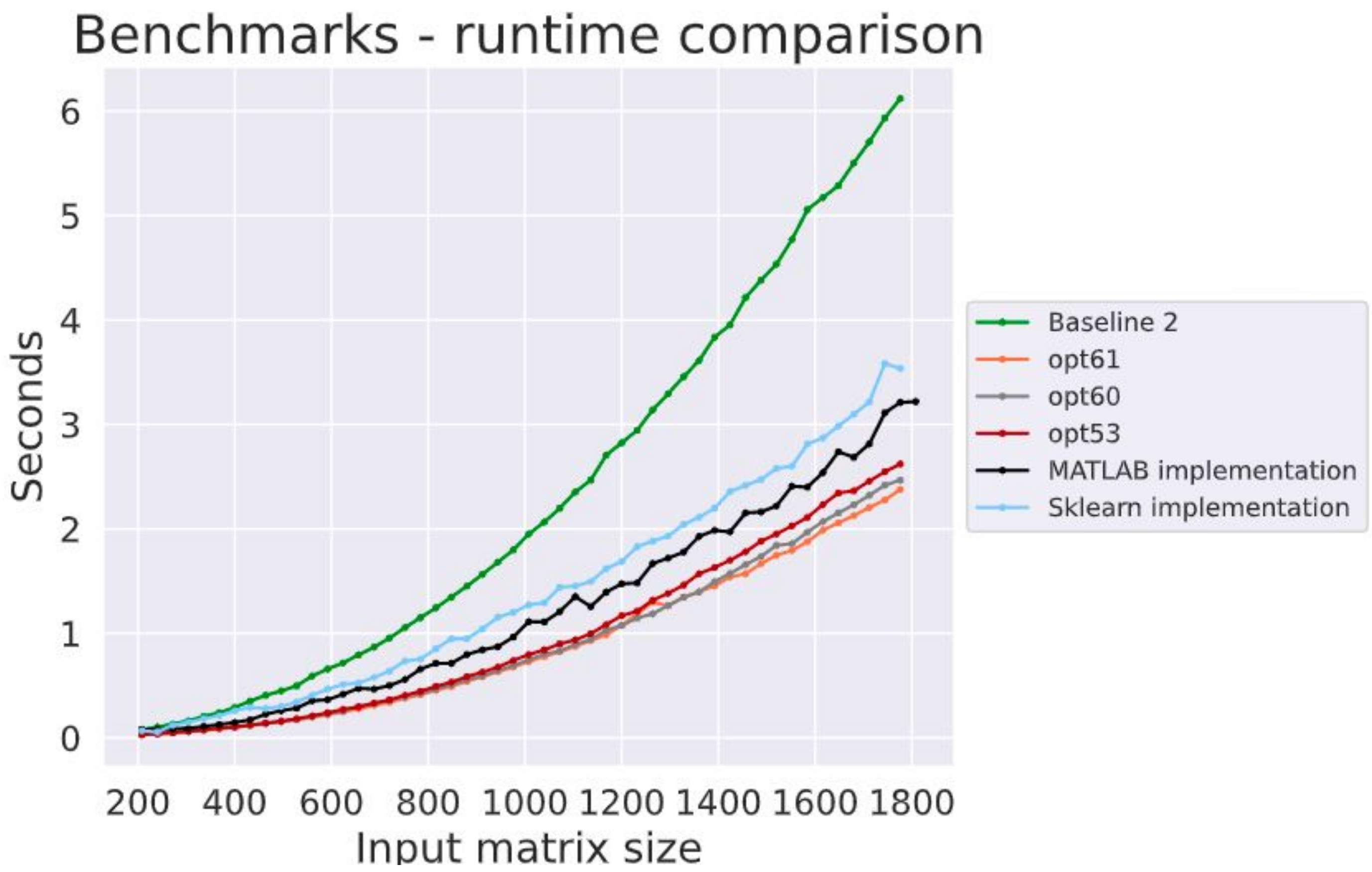
Speedup



NNMF (double precision) on i5-6600K, 3.5 GHz, r = 16

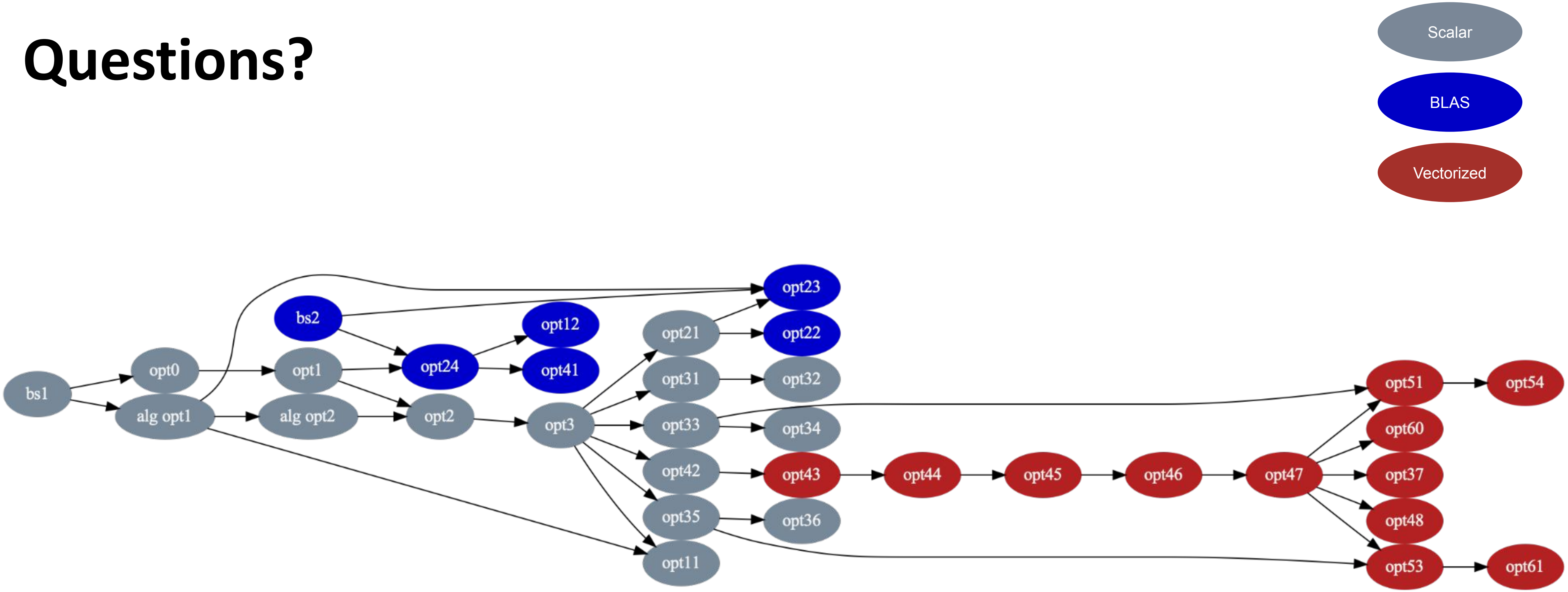
- **Average speedup:**
  - **Optimization 61 - 23.95**
  - Optimization 60 - 22.07
  - Optimization 48 - 21.05
  - Optimization 51 - 20.49
  - Optimization 54 - 19.24
  - Optimization 24 - 17.84
  - Optimization 42 - 7.71

Benchmarking plot



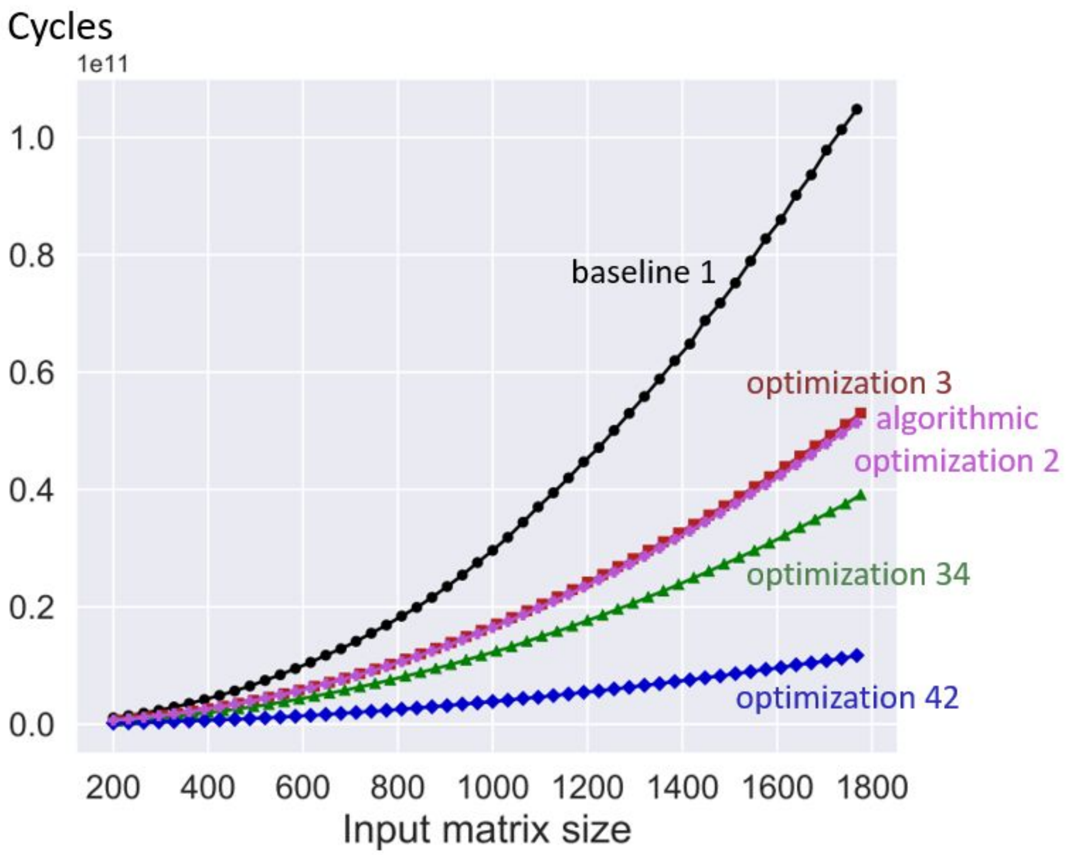
NNMF (double precision) on i7-8565U, 2 GHz, r = 16

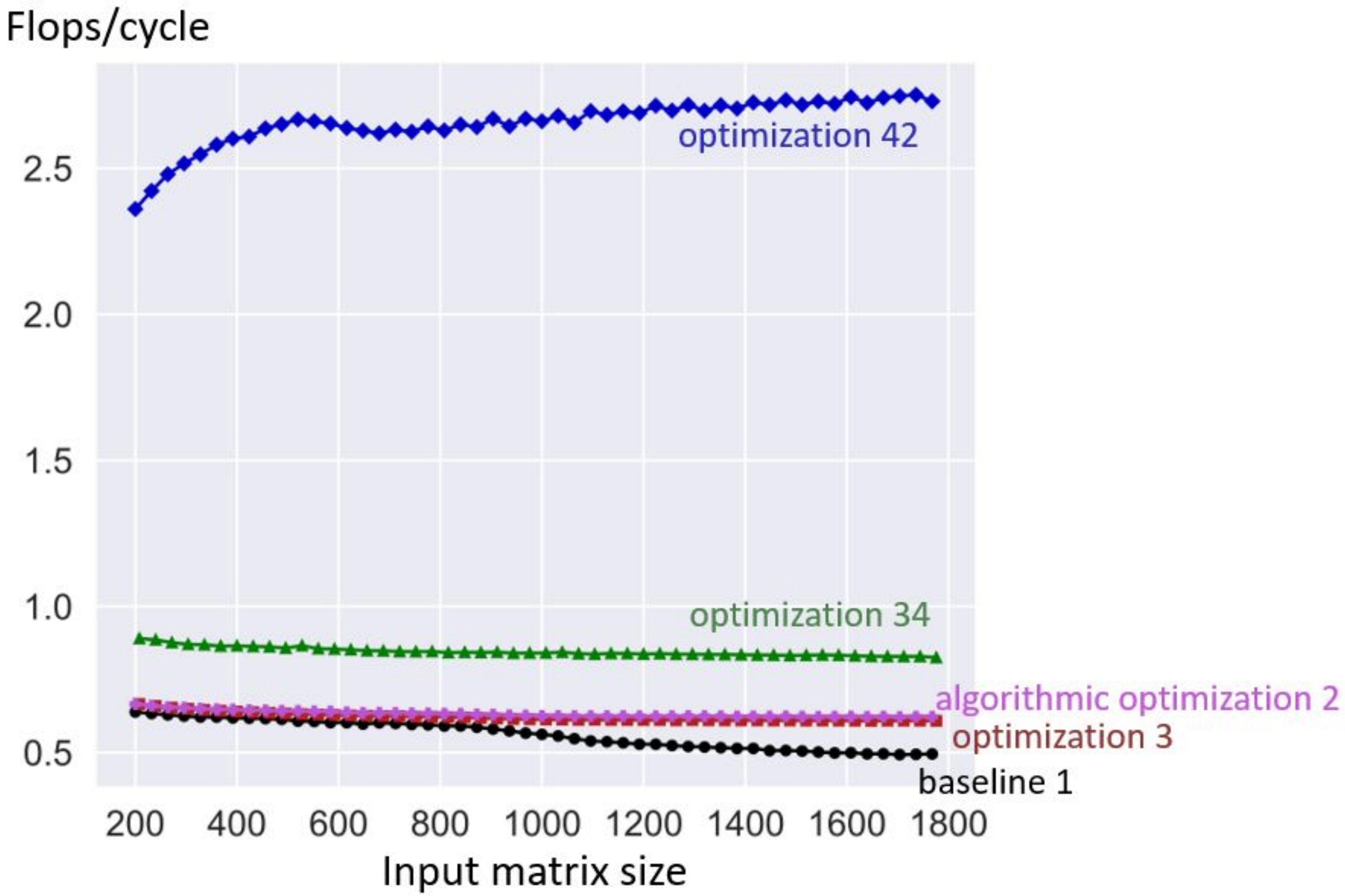
Questions?


Scalar
BLAS
Vectorized
bs1
opt0
alg opt1
bs2
opt1
alg opt2
opt24
opt2
opt12
opt41
opt3
opt21
opt31
opt33
opt42
opt35
opt11
opt23
opt22
opt32
opt34
opt43
opt36
opt44
opt45
opt46
opt47
opt51
opt60
opt37
opt48
opt53
opt54
opt61

Appendix

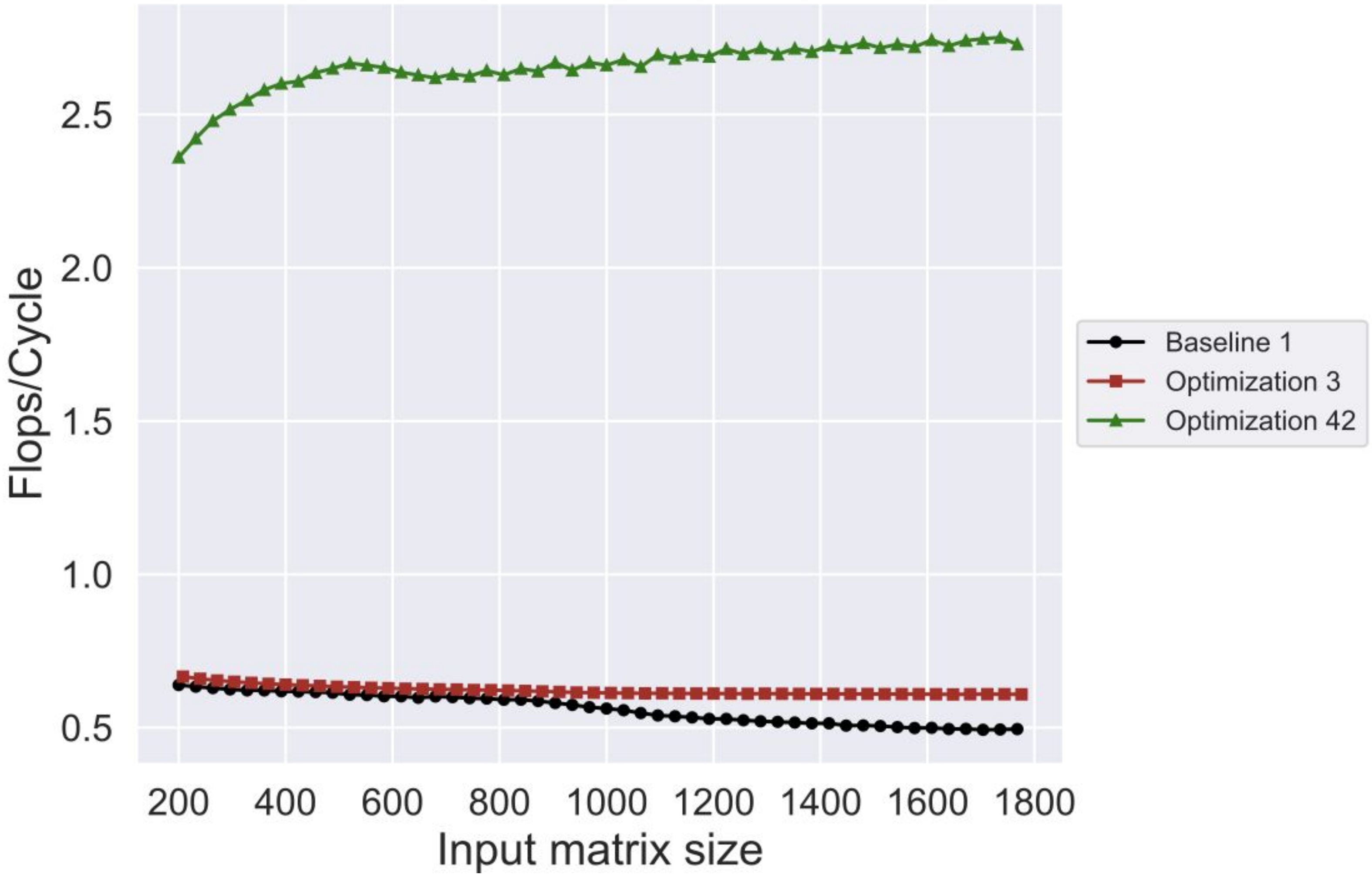
Runtime and Performance Plots - scalar





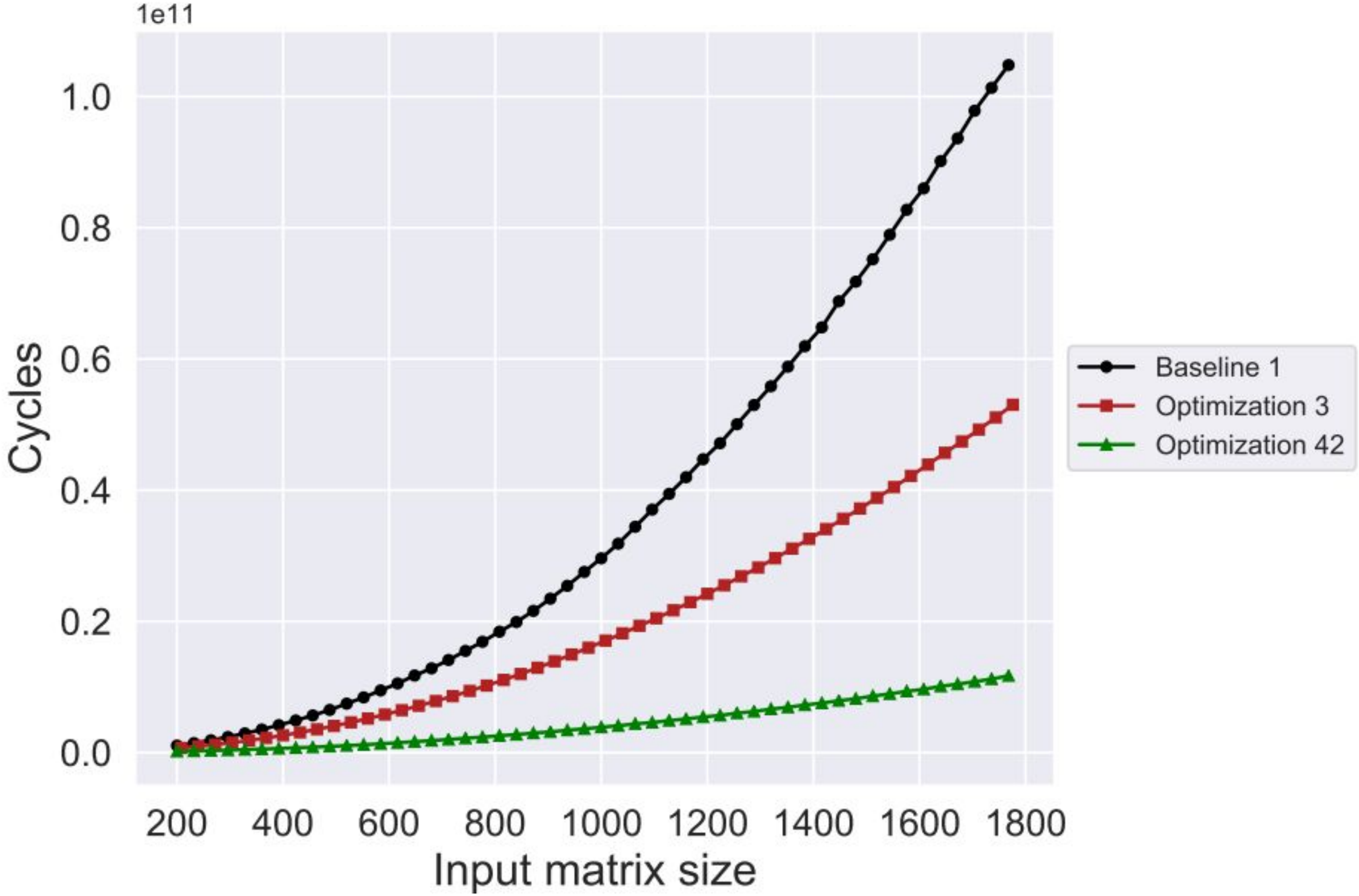
NNMF (double precision) on i5-6600K, 3.5 GHz, r = 16

Performance Scalar Optimizations



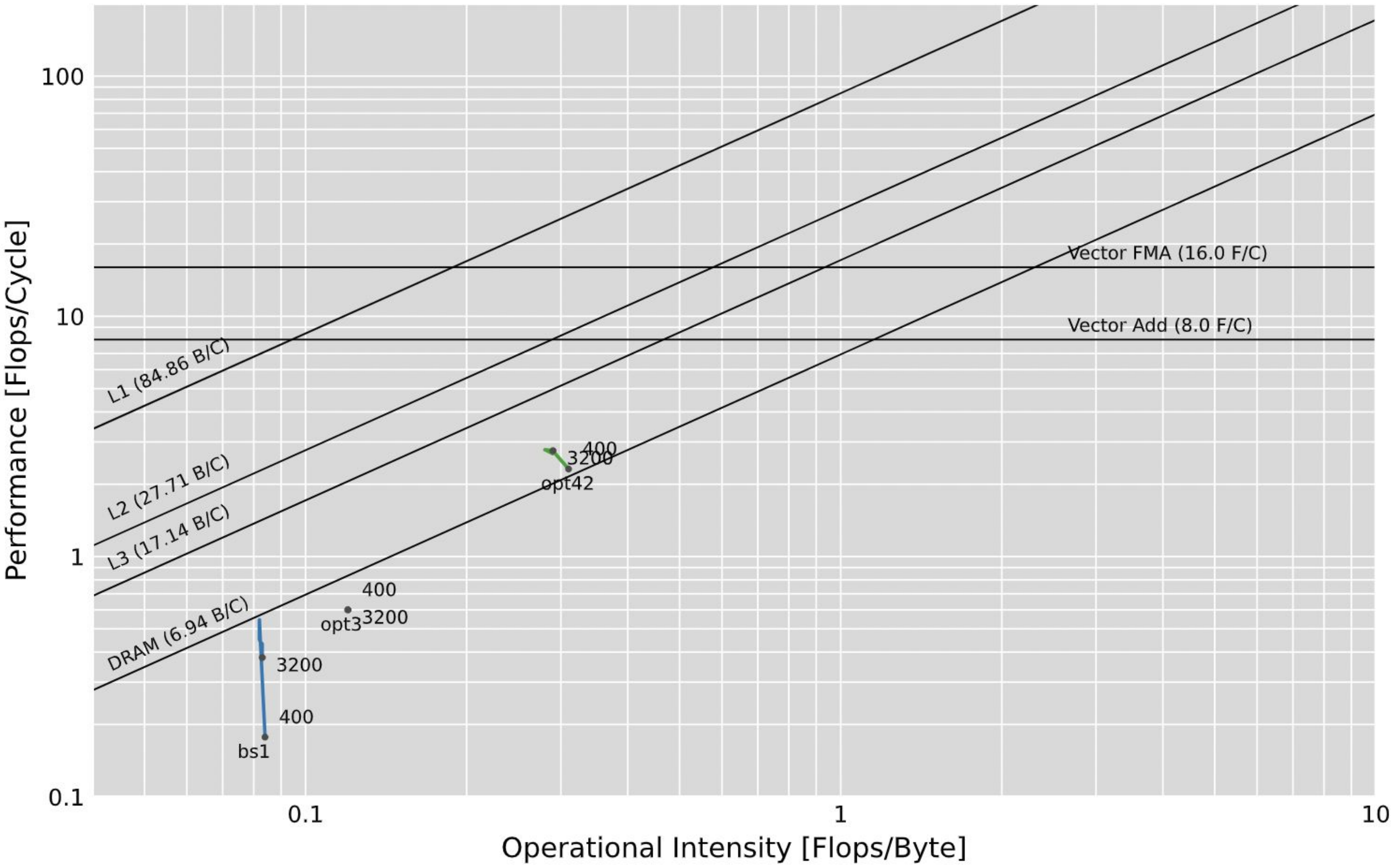
NNMF (double precision*)* on i5-6600K, 3.5 GHz, r = 16

Runtime Scalar Optimizations



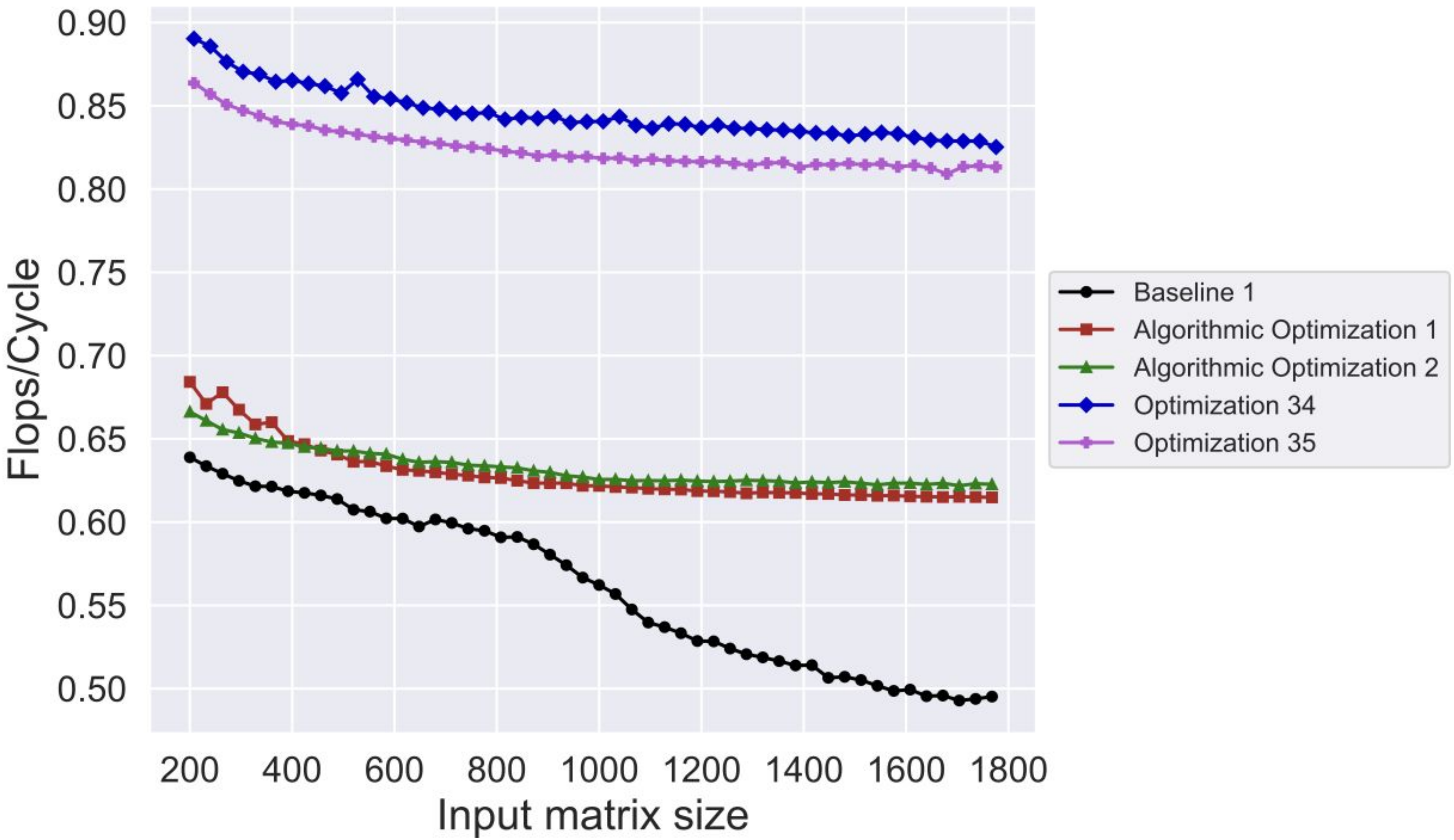
NNMF (double precision) on i5-6600K, 3.5 GHz, r = 16

Roofline Scalar Optimizations



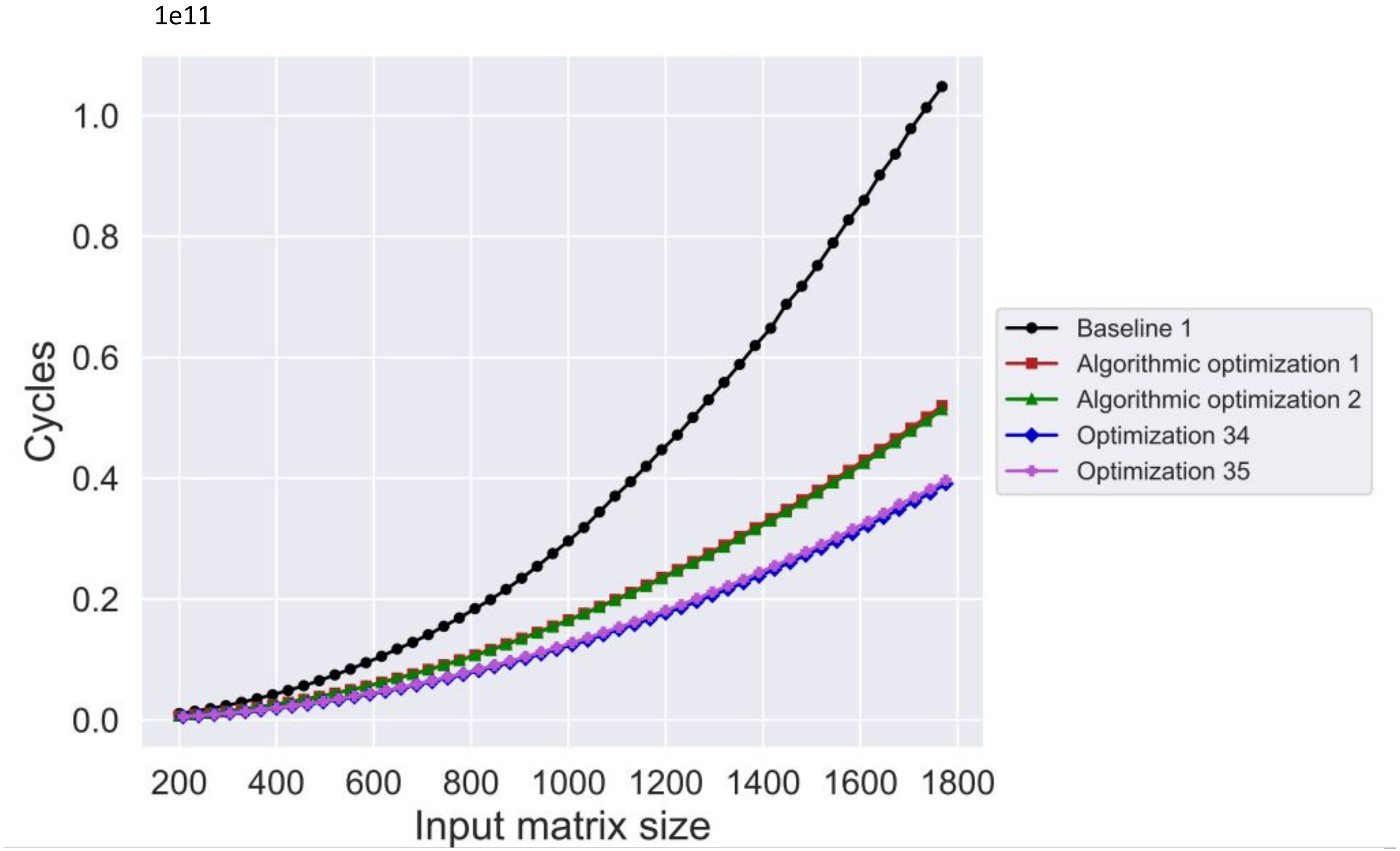
NNMF (double precision) on i5-6600K, 3.5 GHz, r = 16

Performance Scalar Algorithmic Optimizations



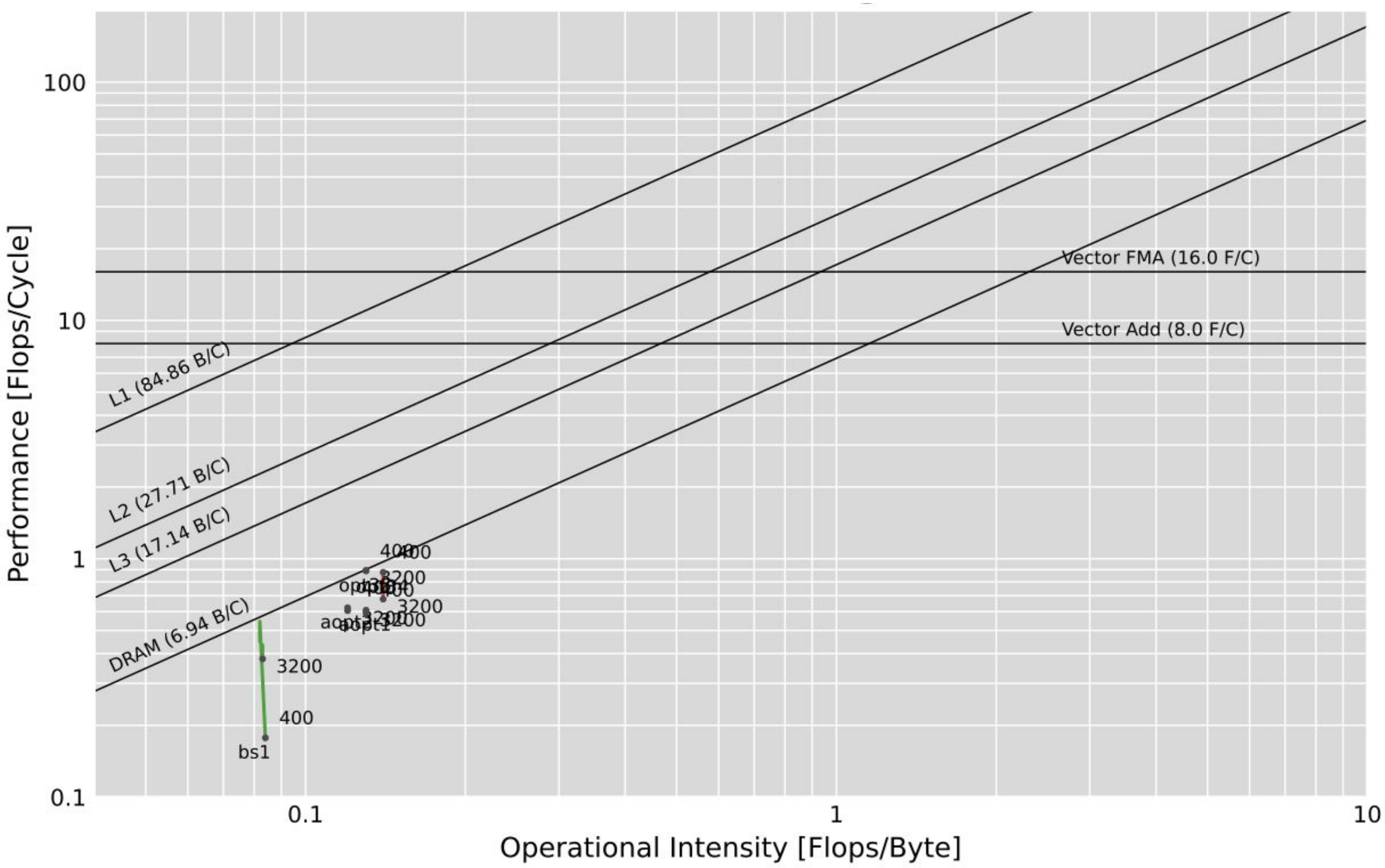
NNMF (double precision) on i5-6600K, 3.5 GHz, r = 16

Runtime Scalar Algorithmic Optimizations



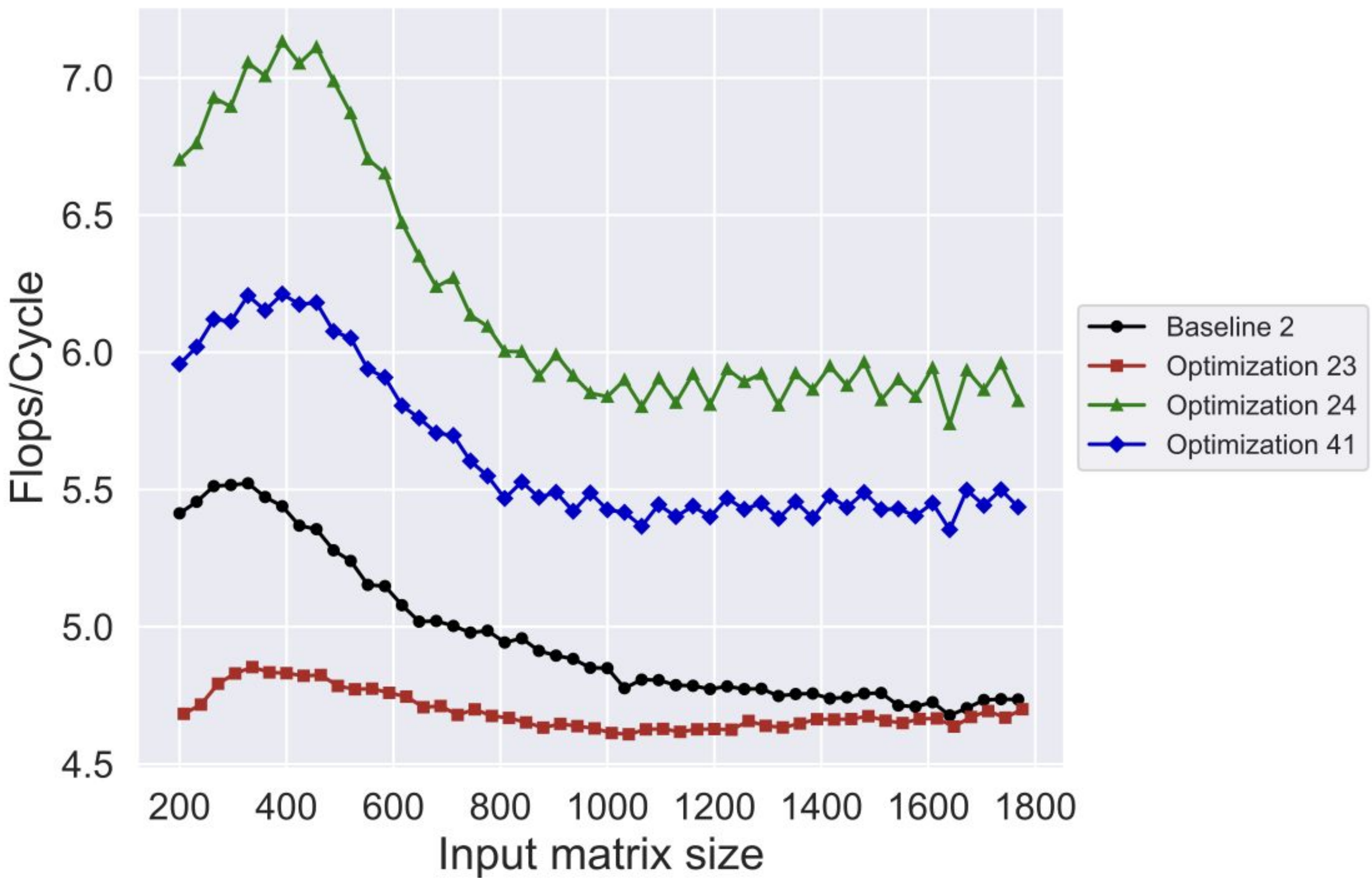
NNMF (double precision) on i5-6600K, 3.5 GHz, r = 16

Roofline Scalar Algorithmic Optimizations



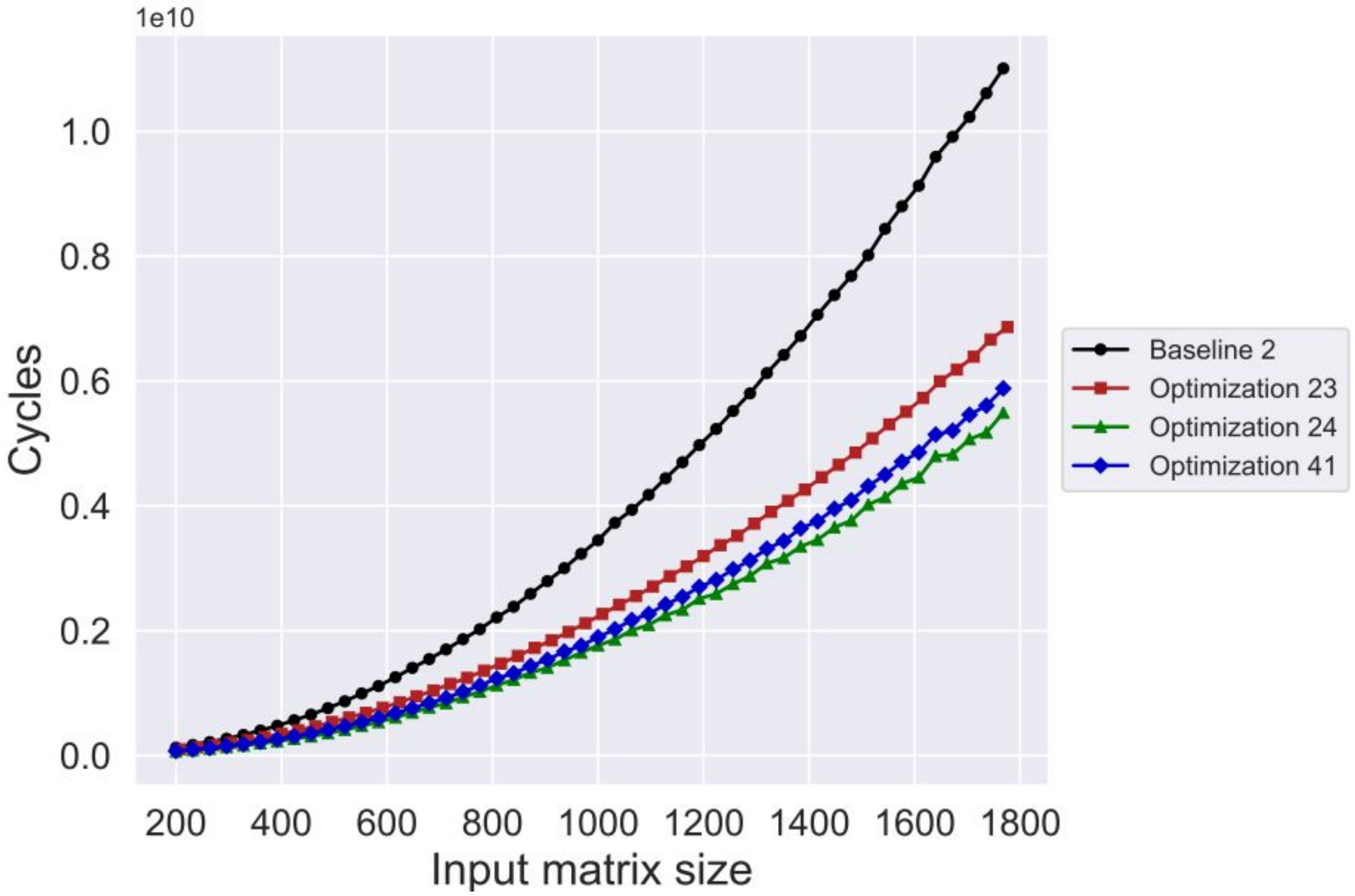
NNMF (double precision) on i5-6600K, 3.5 GHz, r = 16

Performance BLAS Optimizations



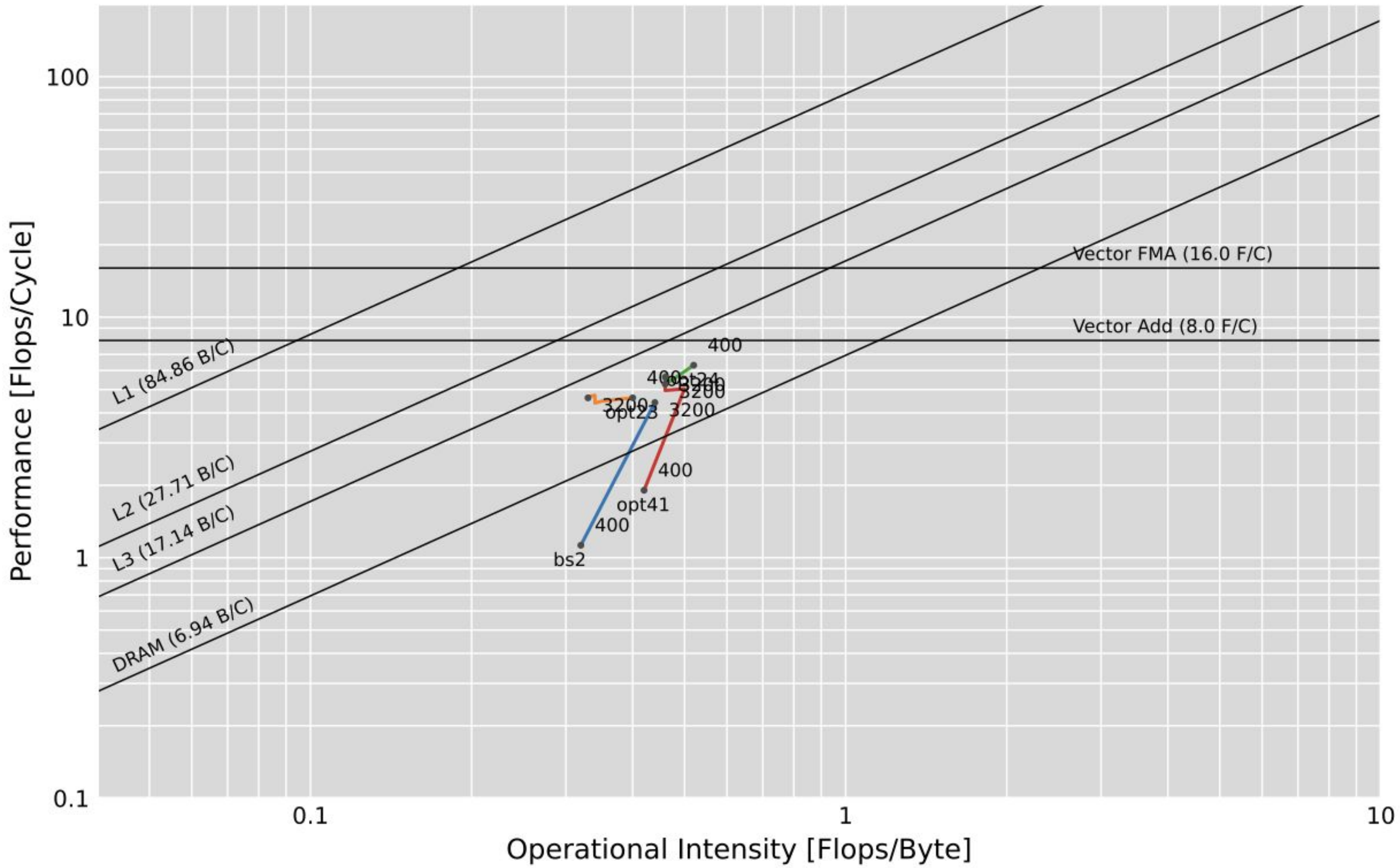
NNMF (double precision) on i5-6600K, 3.5 GHz, r = 16

Runtime BLAS Optimizations



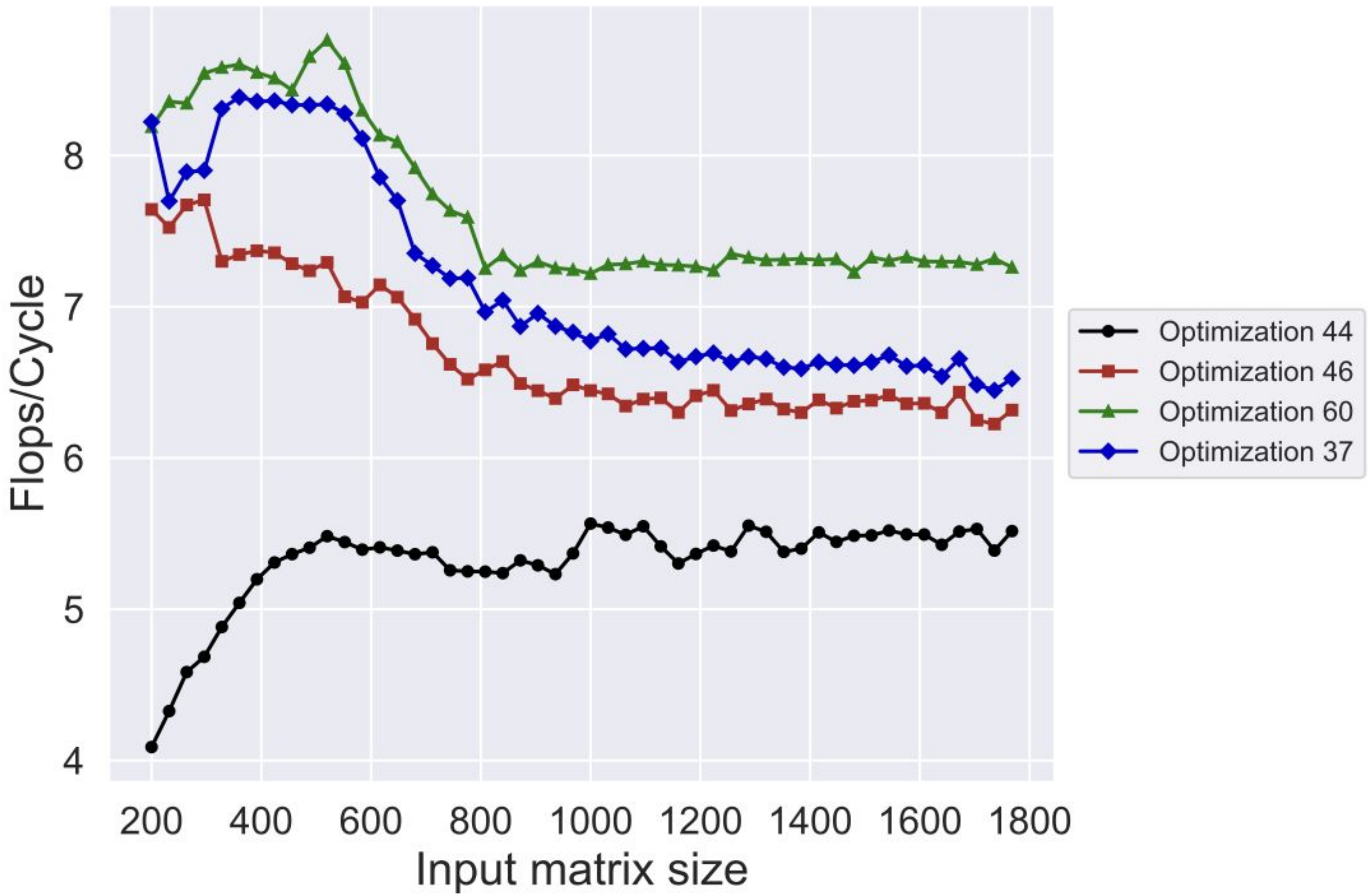
NNMF (double precision) on i5-6600K, 3.5 GHz, r = 16

Roofline BLAS Optimizations



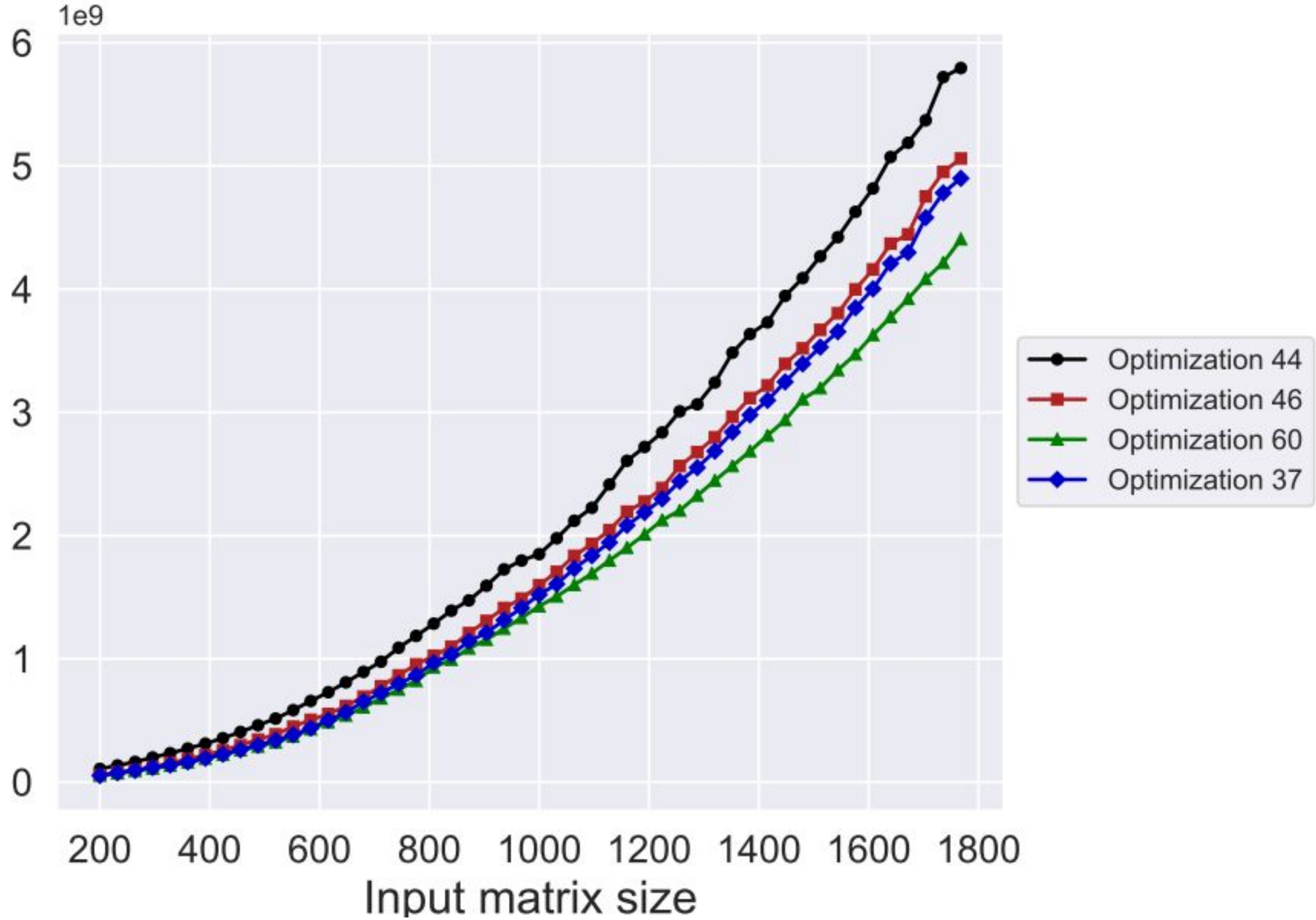
NNMF (double precision) on i5-6600K, 3.5 GHz, r = 16

Performance Vectorized Optimizations



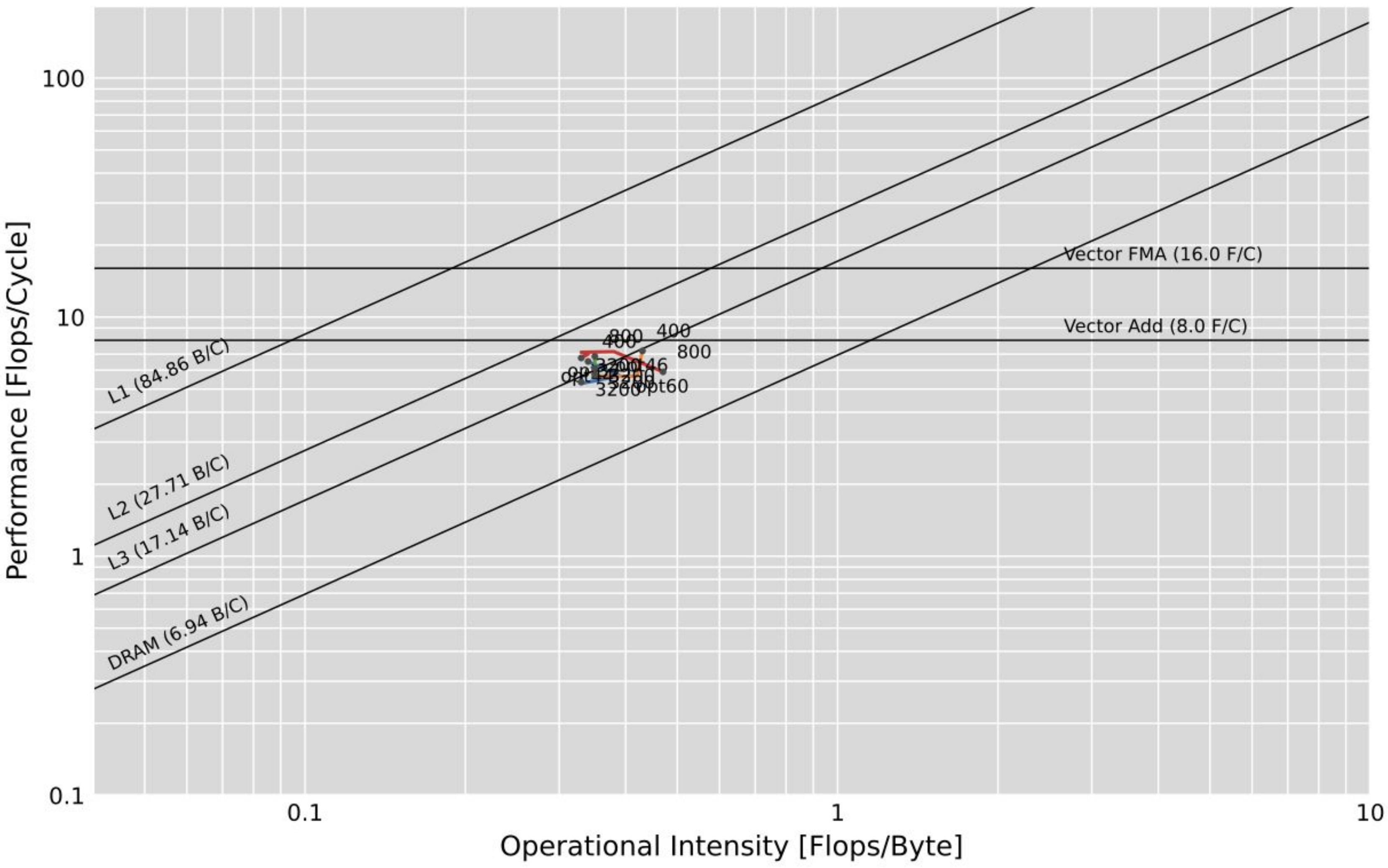
NNMF (double precision) on i5-6600K, 3.5 GHz, r = 16

Runtime Vectorized Optimizations



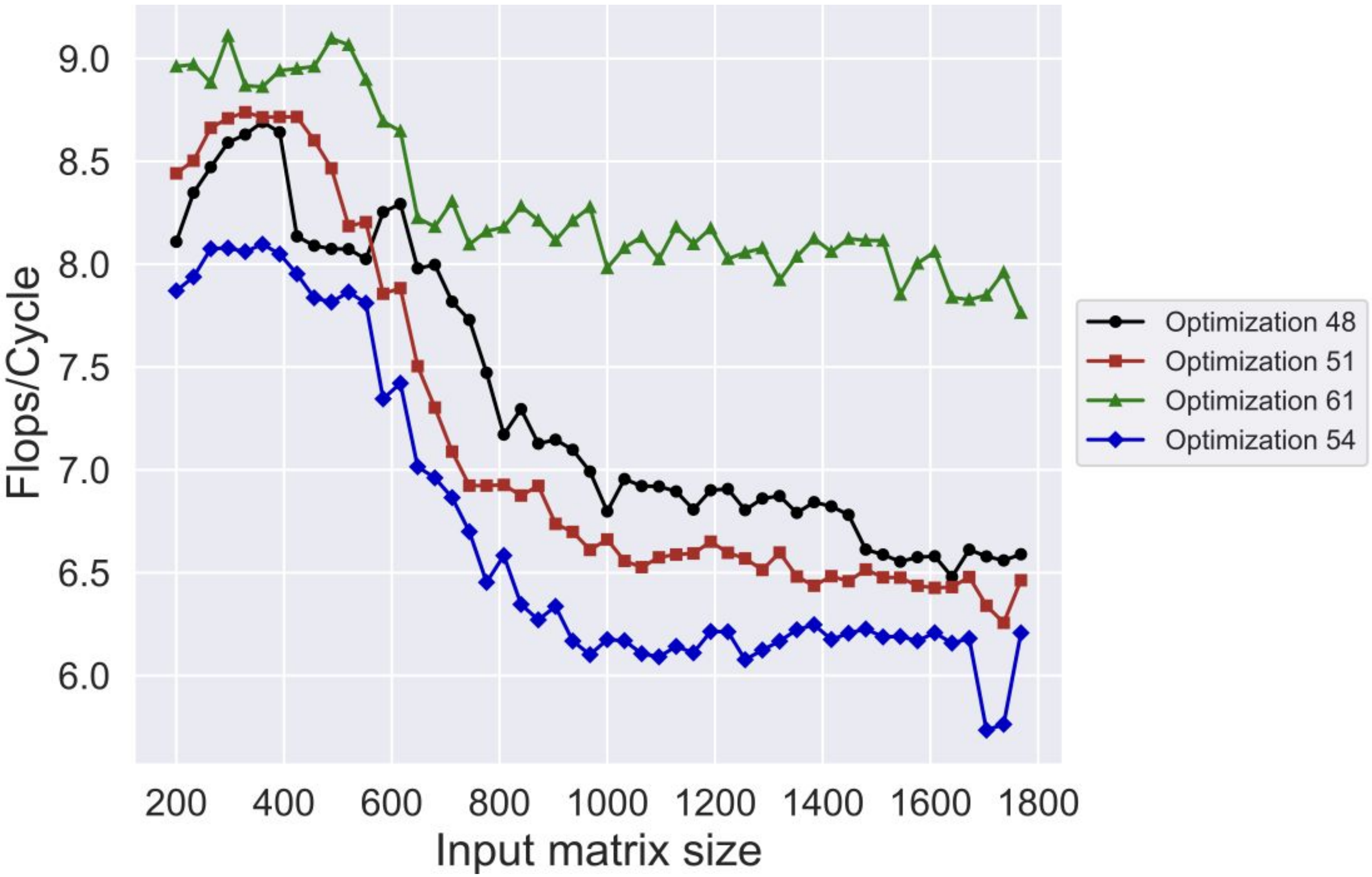
NNMF (double precision) on i5-6600K, 3.5 GHz, r = 16

Roofline Vectorized Optimizations



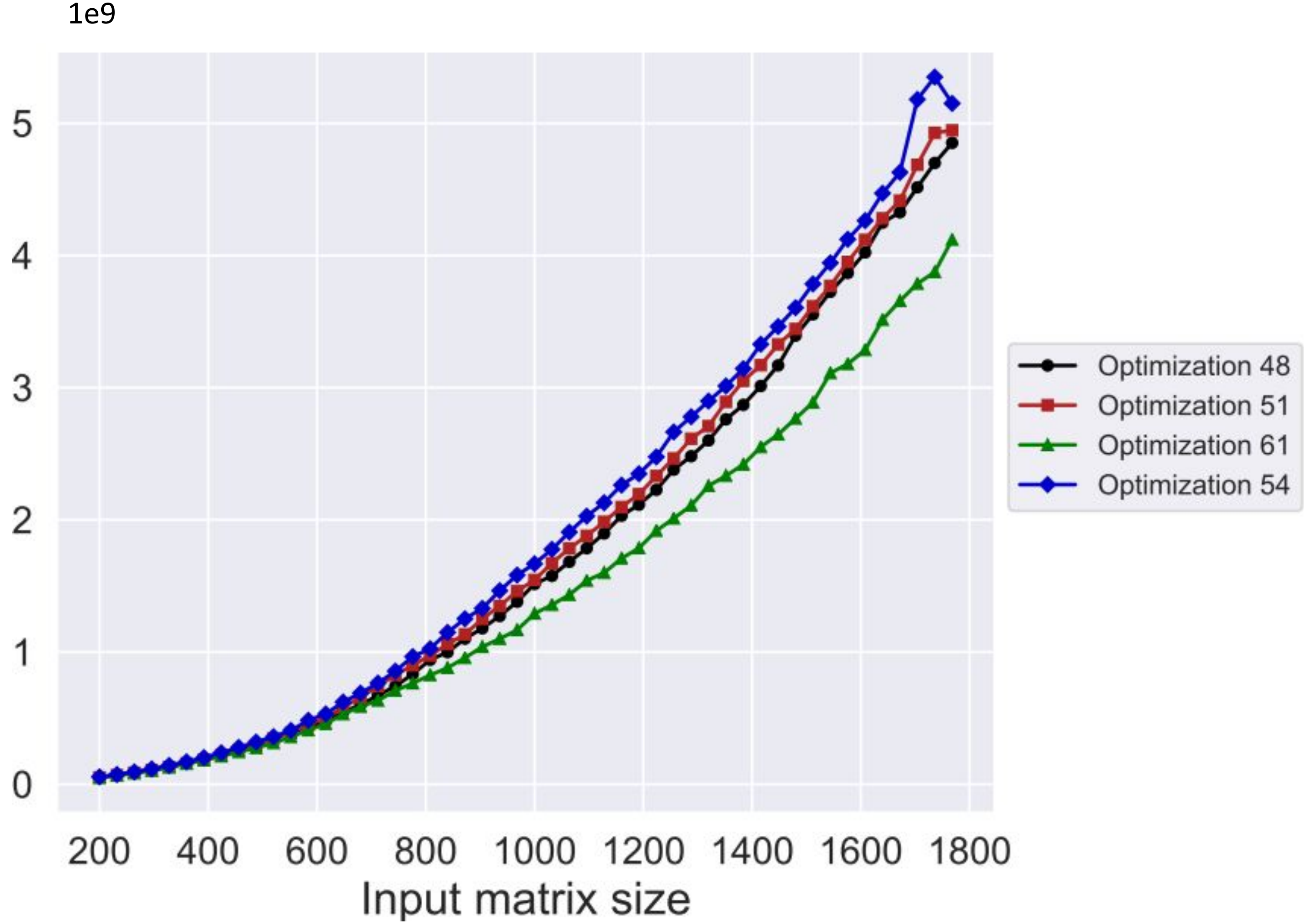
NNMF (double precision) on i5-6600K, 3.5 GHz, r = 16

Performance Vectorized Algorithmic Optimizations



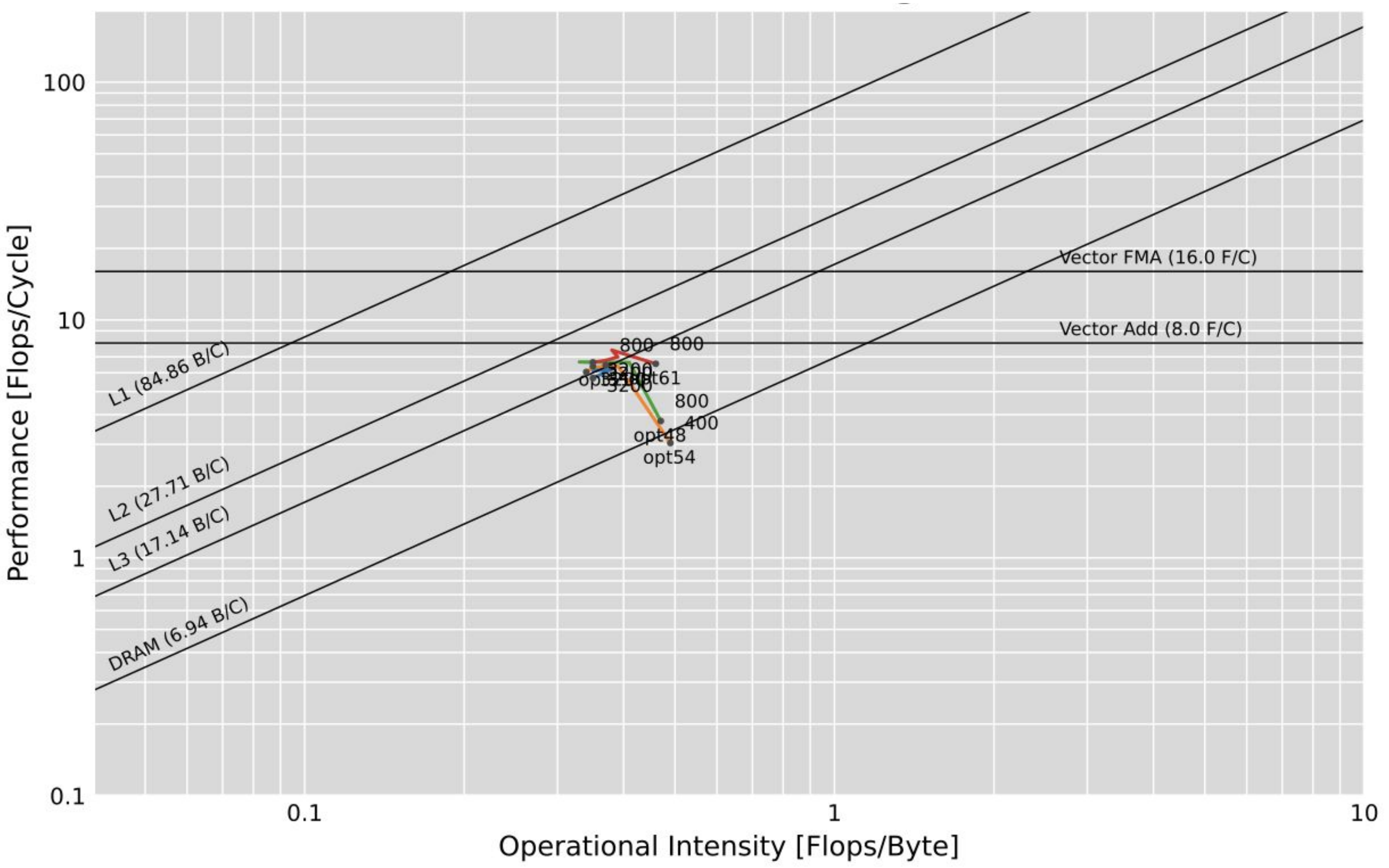
NNMF (double precision) on i5-6600K, 3.5 GHz, r = 16

Runtime Vectorized Algorithmic Optimizations

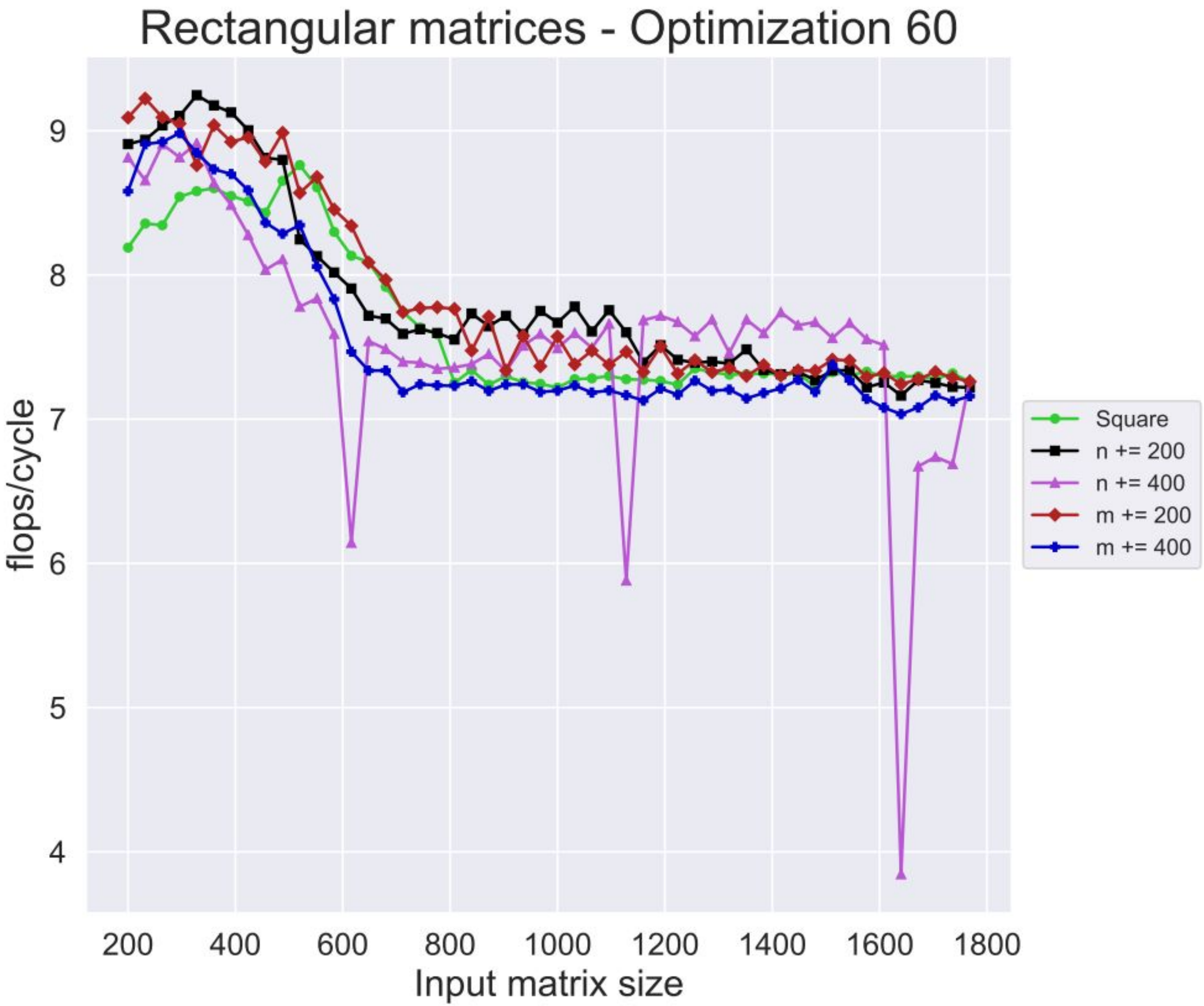


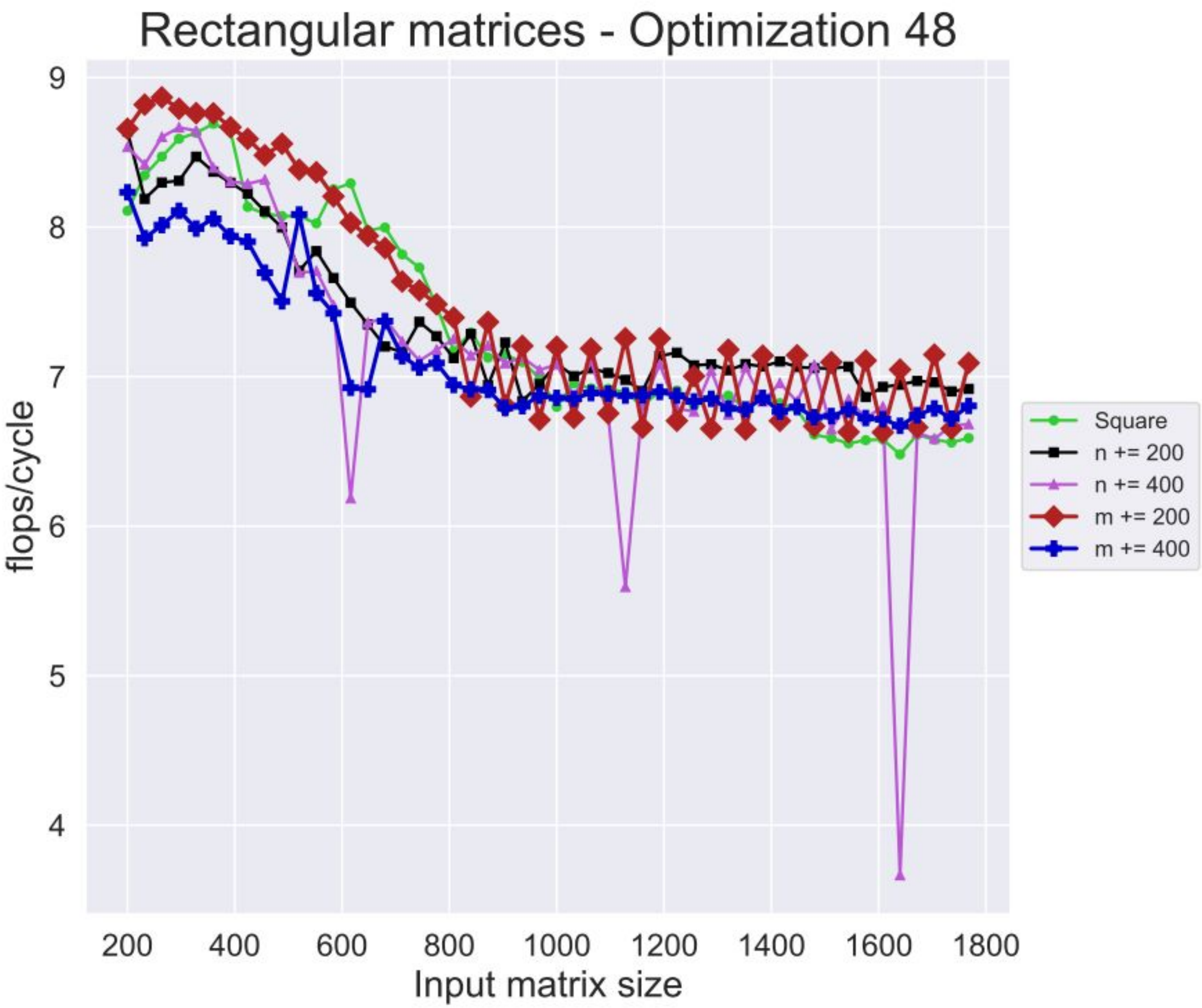
NNMF (double precision) on i5-6600K, 3.5 GHz, r = 16

Roofline Vectorized Algorithmic Optimizations

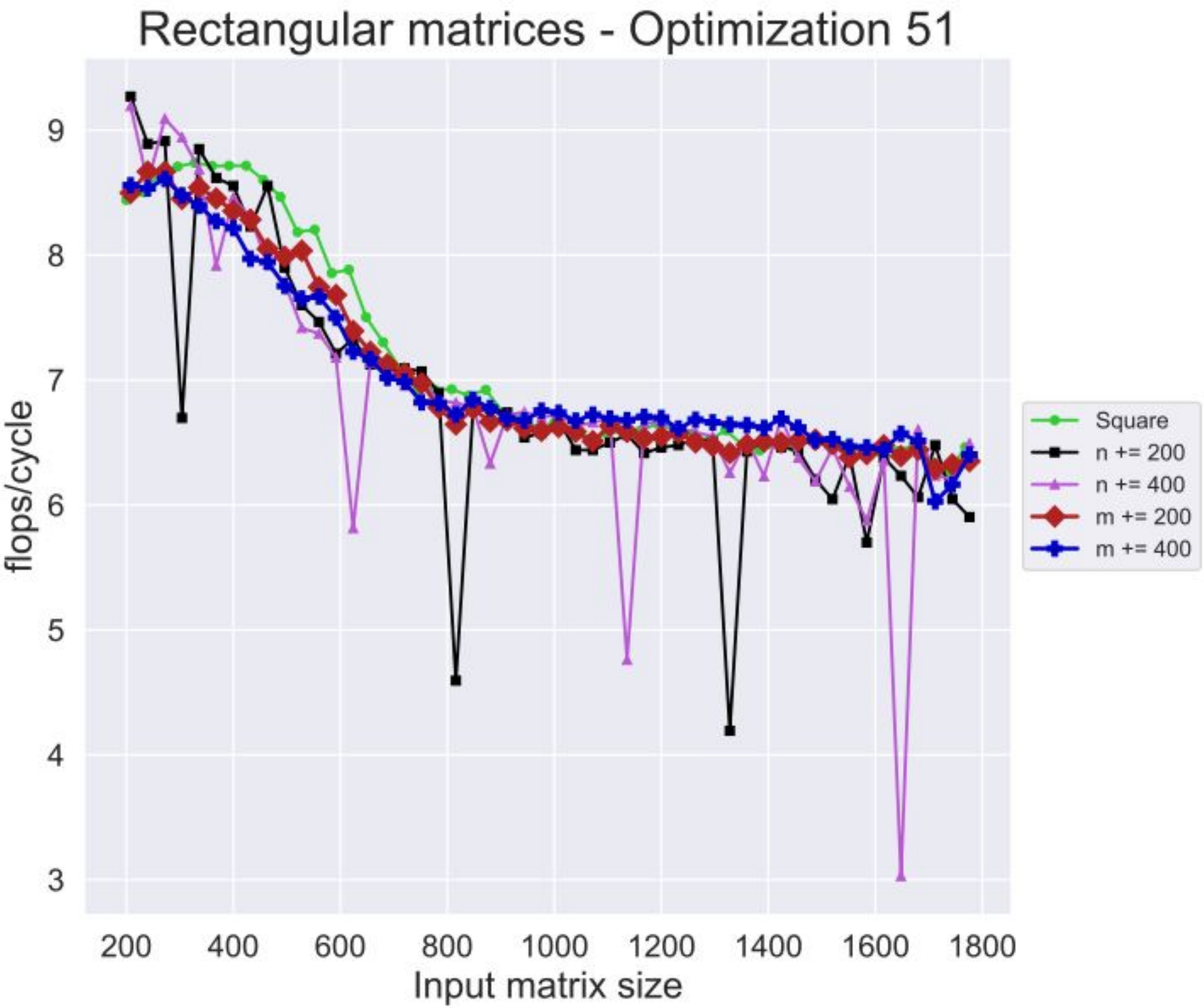


NNMF (double precision) on i5-6600K, 3.5 GHz, r = 16

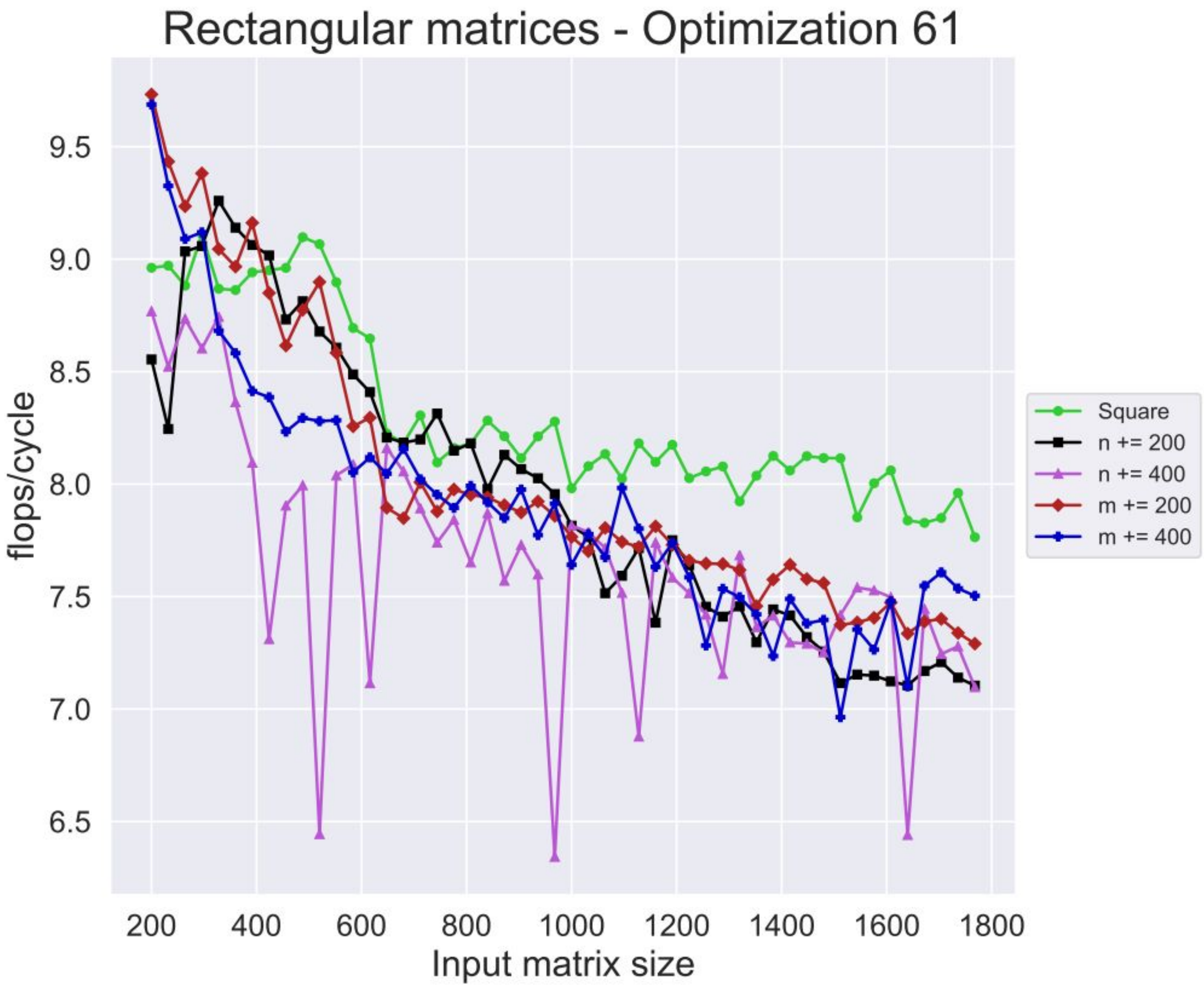

Rectangular matrices - Optimization 60
flops/cycle
Input matrix size
Square
n += 200
n += 400
m += 200
m += 400
9
8
7
6
5
4
200
400
600
800
1000
1200
1400
1600
1800

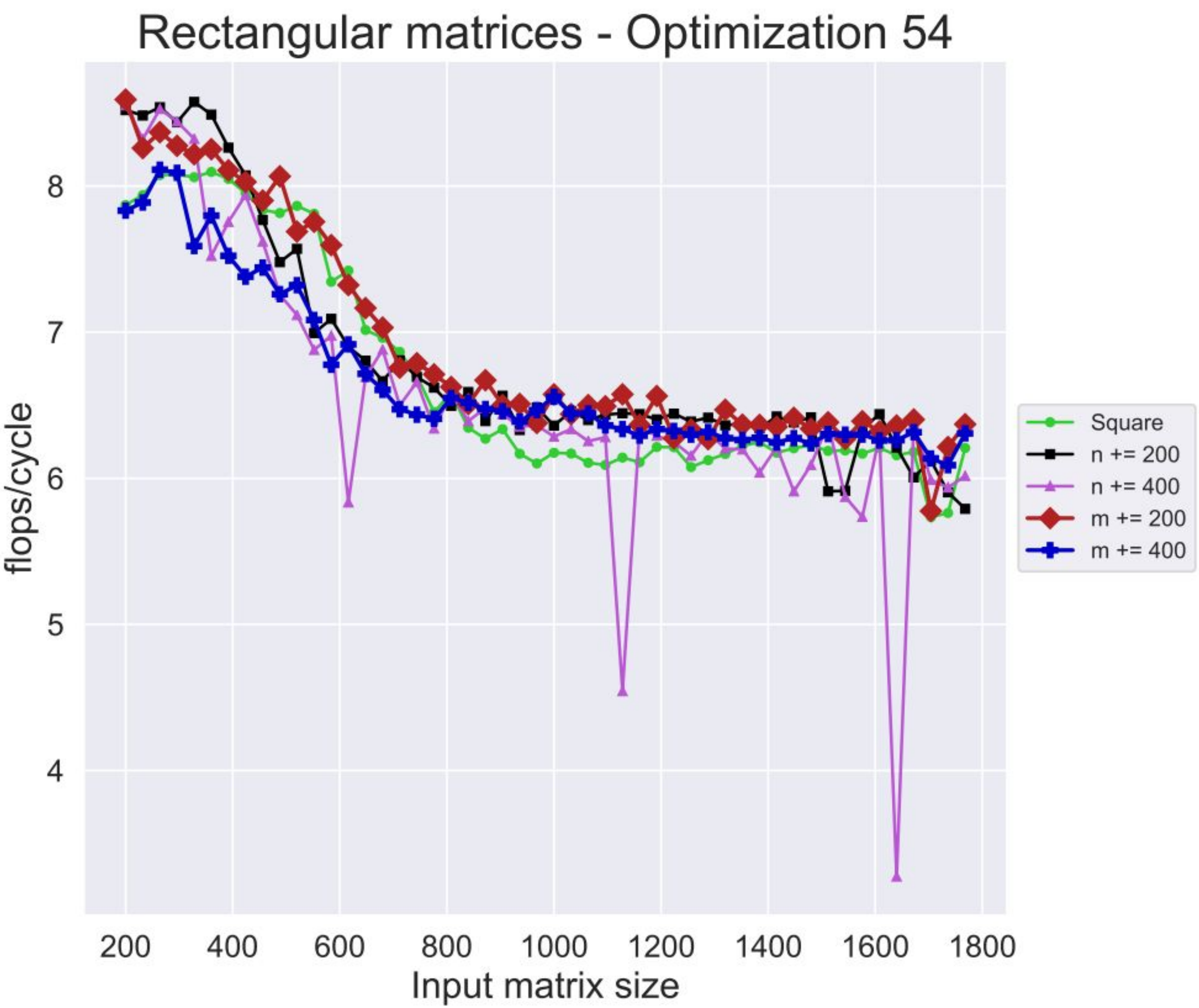


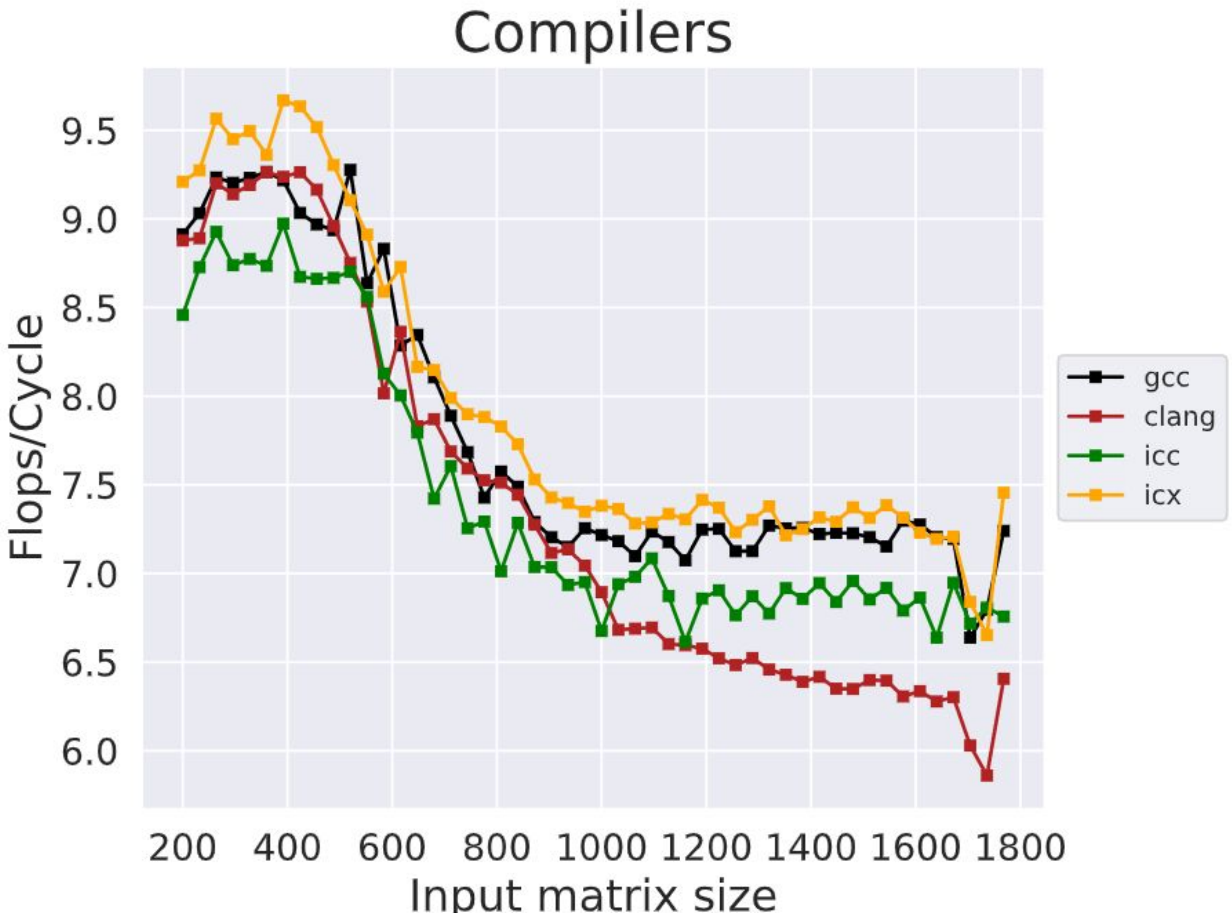
Rectangular matrices - Optimization 48
flops/cycle
Input matrix size
9
8
7
6
5
4
200
400
600
800
1000
1200
1400
1600
1800
Square
n += 200
n += 400
m += 200
m += 400

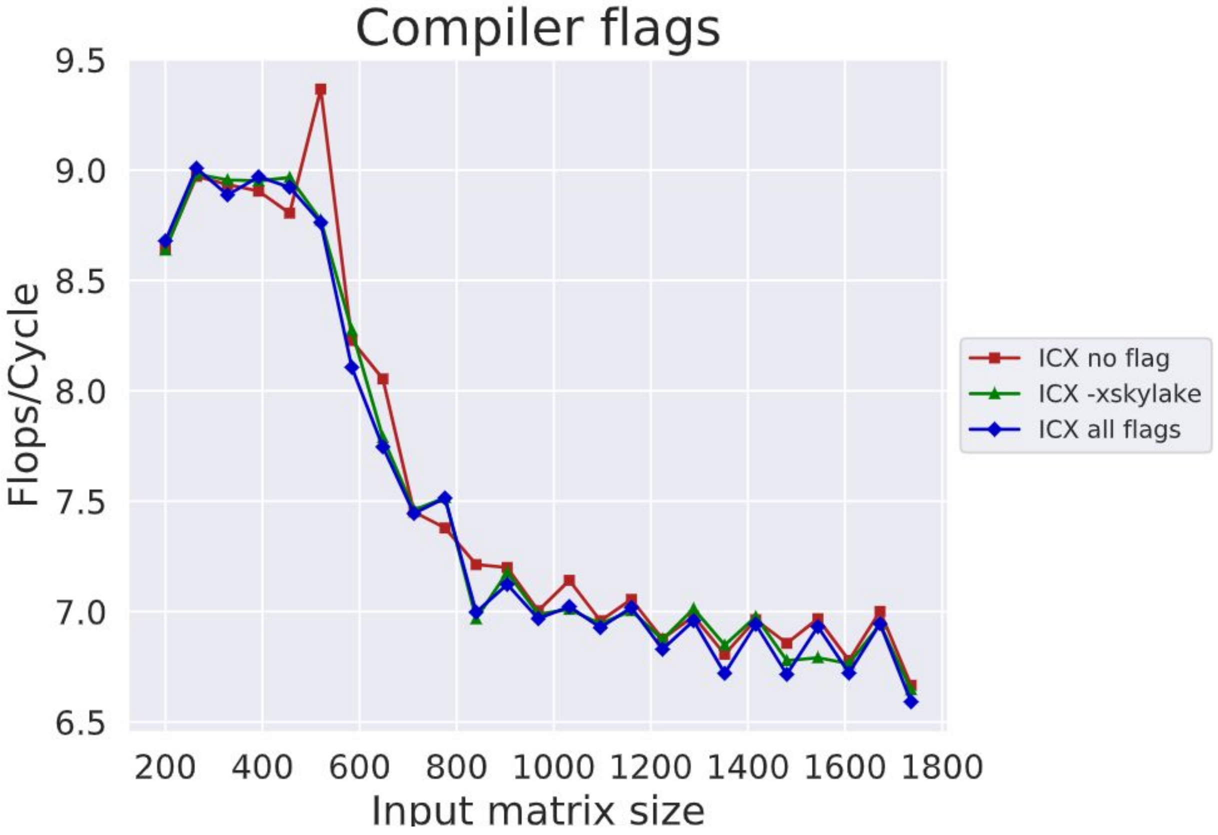


Rectangular matrices - Optimization 51
flops/cycle
Input matrix size
Square
n += 200
n += 400
m += 200
m += 400

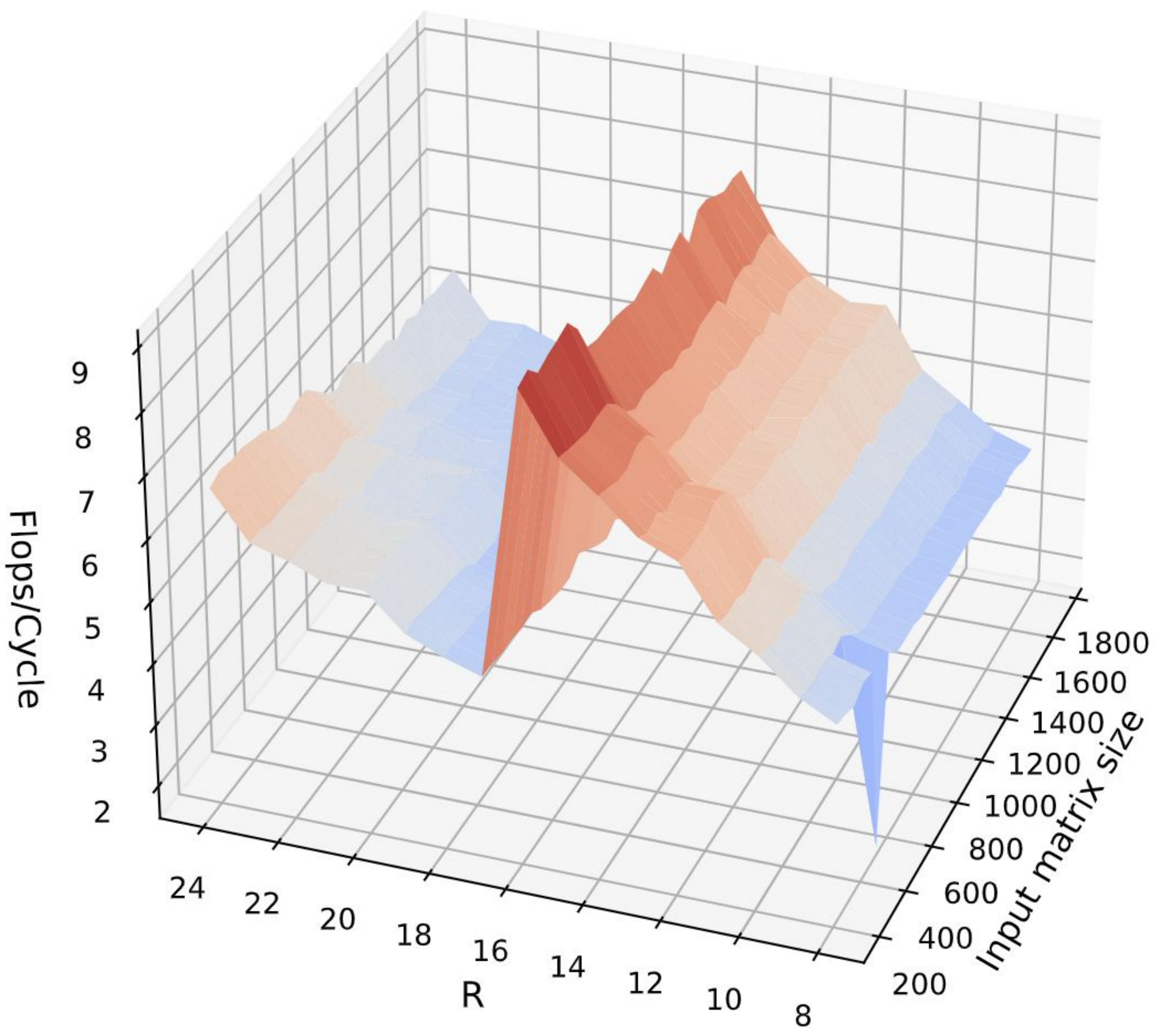

Rectangular matrices - Optimization 61
flops/cycle
Input matrix size
9.5
9.0
8.5
8.0
7.5
7.0
6.5
200
400
600
800
1000
1200
1400
1600
1800
Square
n += 200
n += 400
m += 200
m += 400

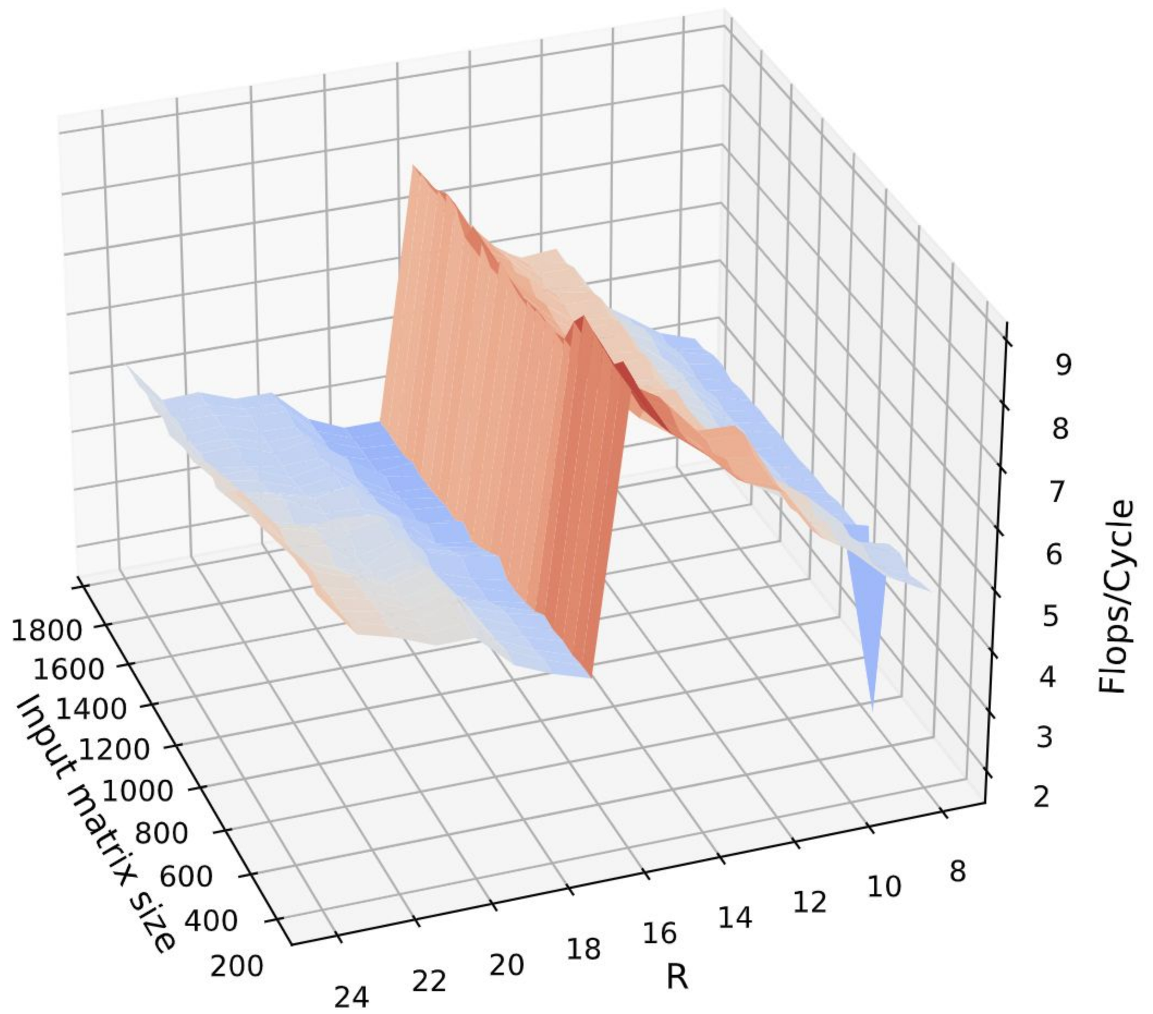

Rectangular matrices - Optimization 54
flops/cycle
Input matrix size
8
7
6
5
4
200
400
600
800
1000
1200
1400
1600
1800
Square
n += 200
n += 400
m += 200
m += 400


Compilers
9.5
9.0
8.5
8.0
7.5
7.0
6.5
6.0
Flops/Cycle
200
400
600
800
1000
1200
1400
1600
1800
Input matrix size
gcc
clang
icc
icx

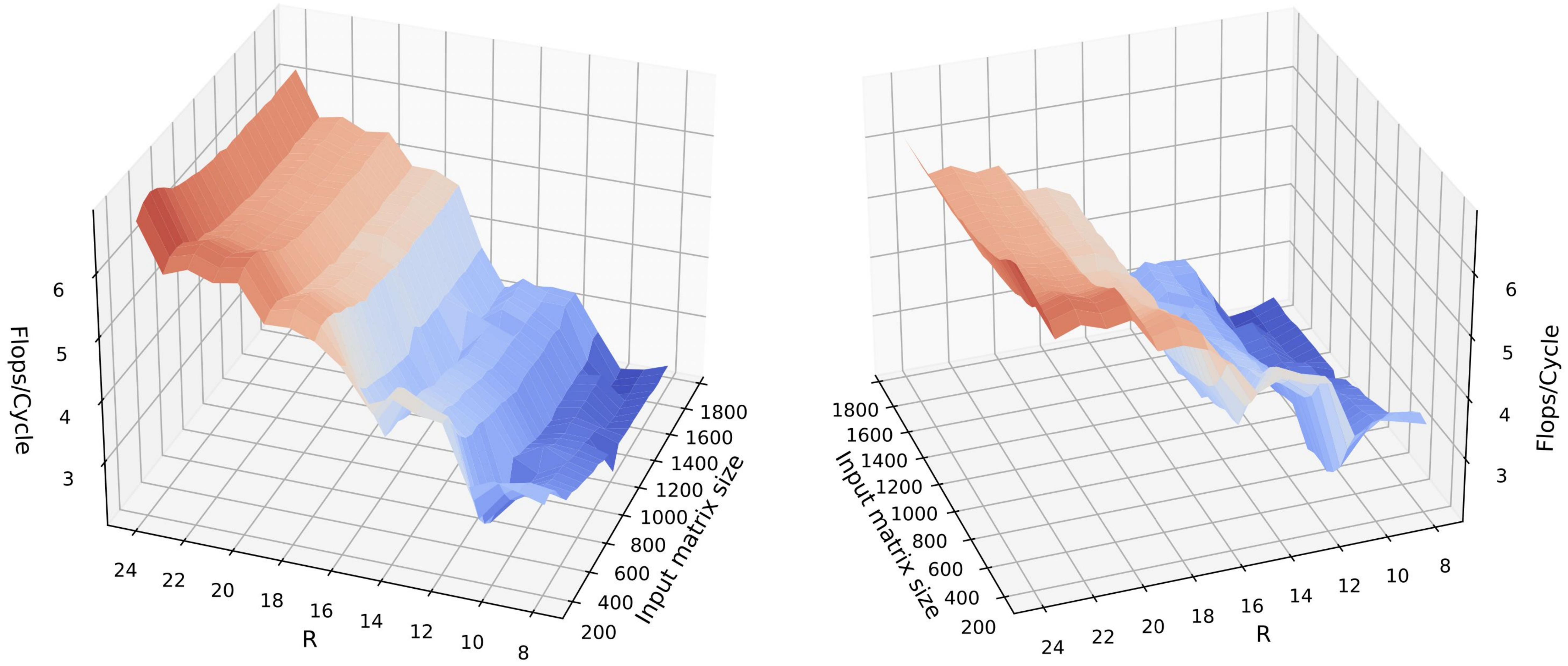

Compiler flags
9.5
9.0
8.5
8.0
7.5
7.0
6.5
Flops/Cycle
200
400
600
800
1000
1200
1400
1600
1800
Input matrix size
ICX no flag
ICX -xskylake
ICX all flags

3D Performance Plot Optimization 61





3D Performance Plot Baseline 2



Algorithmic optimization 2 – Interleave matrix multiplications

- **Avoids having to store and reread the numerator and the denominator**
- **The same approach is used in the calculation of W^{n+1} as well**

$$H^{n+1}_{[i,j]} = H^{n}_{[i,j]} \cdot \frac{\overbrace{((W^n)^T V)}^{\text{numerator - N}}{}_{[i,j]}}{\underbrace{\underbrace{((W^n)^T W^n}_{\text{denominator\_left - Dl}} H^n)}_{\text{denominator - D}}{}_{[i,j]}}$$

$$V : m \times n$$
$$W : m \times r$$
$$H : r \times n$$
$$Dl : r \times r$$
$$N : r \times n$$
$$D : r \times n$$

```
N = matrix_mul(W^T, V)
D = matrix_mul(Dl, H)
for(i=0; i<r; i++)
  for(j=0; j<n; j++)
    H[i][j] = H[i][j] * N[i][j] / D[i][j]
```

contain triple loops

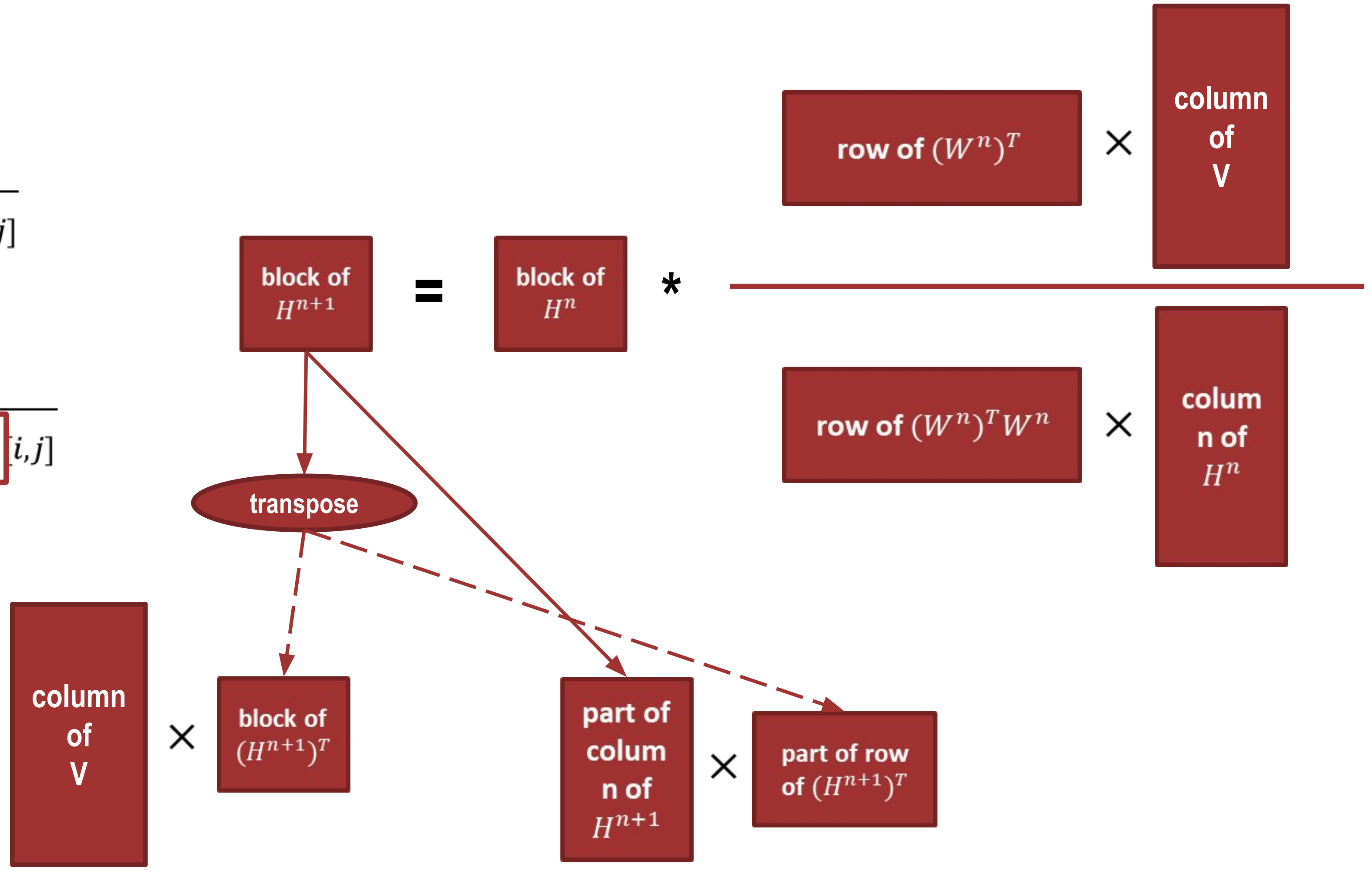
```
for(i=0; i<r; i++)
  for(j=0; j<n; j++)
    accumulator_N = 0
    accumulator_D = 0
    for(k=0; k<m; k++)
      accumulator_N += W^T[i][k] * V[k][j]
      if(k<r)
        accumulator_D += Dl[i][k] * H[k][j]
    H[i][j] = H[i][j] * accumulator_N / accumulator_D
```

Algorithmic optimization 3 – Reuse block of H

- The calculated block of H^{n+1} is immediately used in the calculation of $V(H^{n+1})^T$ and $H^{n+1}(H^{n+1})^T$

$$\boxed{H^{n+1}_{[i,j]}} = H^n_{[i,j]} \cdot \frac{((W^n)^T\, V)_{[i,j]}}{((W^n)^T\, W^n\, H^n)_{[i,j]}}$$

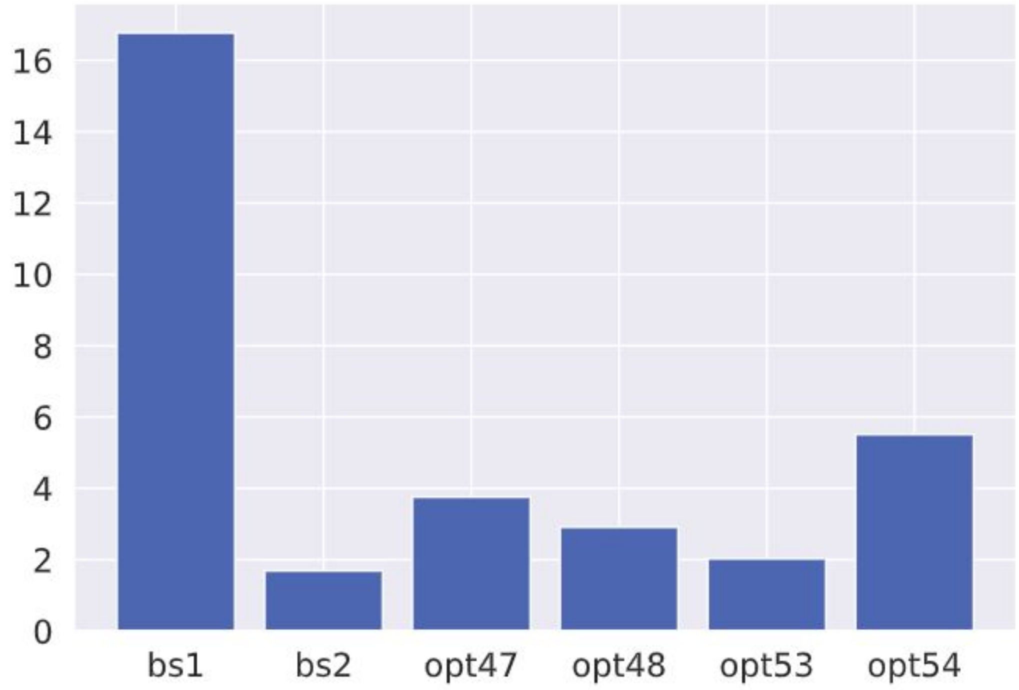
$$W^{n+1}_{[i,j]} = W^n_{[i,j]} \cdot \frac{\boxed{(V\,(H^{n+1})^T)}_{[i,j]}}{(W^n\,\boxed{H^{n+1}\,(H^{n+1})^T)}_{[i,j]}}$$



Additional analysis

- opt47 reduces cache misses by blocking
- opt53 reduces cache misses reusing W
- opt60 computes the approximation matrix WH one block at the time
- opt61 decrease in cache misses is higher than 60 and 53 combined

LLC MISSES(G)



V : 800x800
W: 800x16
H: 800x16