Matrix Exponential Optimization

Advanced Systems Lab

Team 28

ETH

Eidgenössische Technische Hochschule Zürich

Swiss Federal Institute of Technology Zurich

Algorithm

- **Goal**
  - Algorithm 5.1 proposes a new way to approximate e^A, where *A* is a square matrix.
- **Motivation**
  - Previous implementation had poor precision with big matrices (overscaling)
- **Extensive use of matrix multiplication**
  - Wipes out cache
  - *Blas* allowed for MMM
  - Main focus in optimization code fragments between MMM

Infrastructure

- **Validation**
  - Existing implementation: *scipy expm* in scipy.linalg module (Python)
  - The C++ infrastructure interacts with a Python script to get the a valid computation
- **Timing**
  - Processor used: i5-6400, Skylake microarchitecture @2700 Mhz
  - Cycles computed using *TSC* counter

Cost analysis

- **Cost measure**
  - Total number of FLOPS
  - All floating-point operations have the same weight
  - Code instructed to count FLOPS through macros

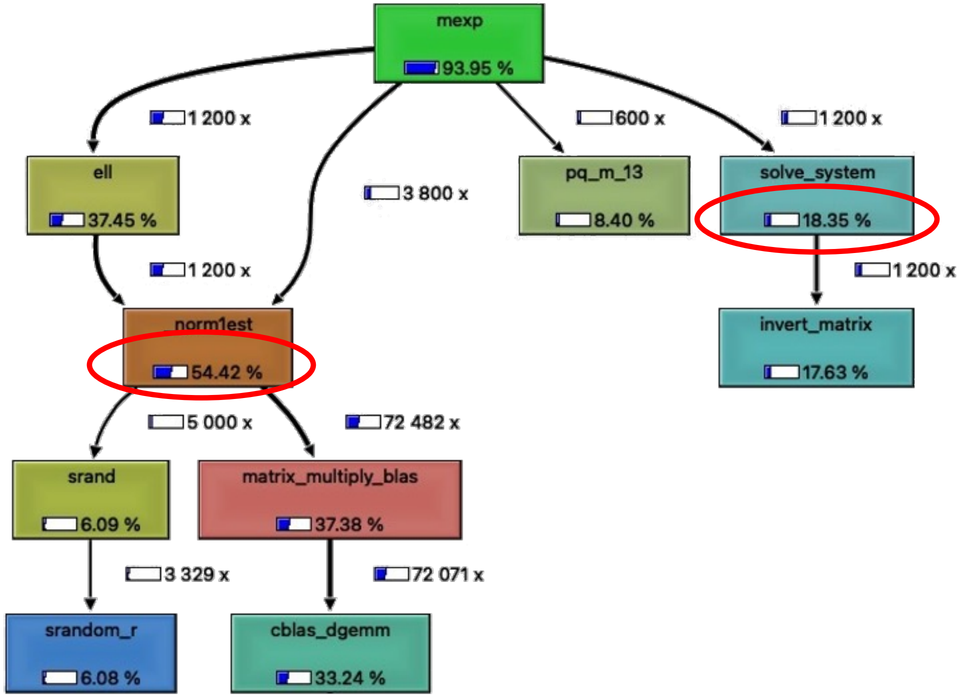
```
#ifdef FLOPS
#define FLOPS_RESET
#define FLOPS_ADD(x)
#define FLOPS_ADD_N(x)
#define FLOPS_ADD_N2(x)
#define FLOPS_ADD_N3(x)
#define FLOPS_PRINT
#endif
```

Straightforward C Implementation

- **Codebase**
  - **mexp_basic_.c**: performs the Matrix Exponential (Memexp), utilizing functions from utils_.c and queries *normest* from norms_.c to get an estimation of the norm from a matrix
  - **utils_.c** and **utils_.h**: utility functions needed in mexp_basic_.c, promoting modularity
  - **norms_.c**: houses *normest* for norm estimation, used by mexp_basic_.c for matrix calculations
- **BLAS is used to perform MMMs**
- **Matrices are represented as array in row-major order**

Callgrind analysis

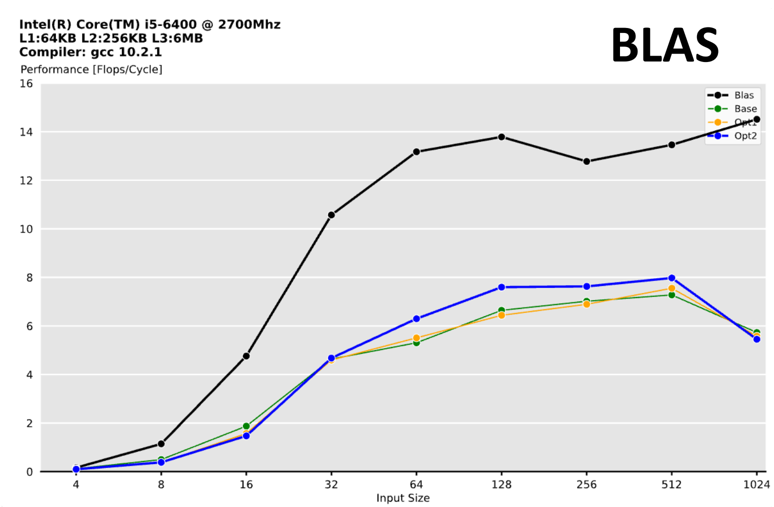
- **Bottlenecks**
  - solve-system
  - normest

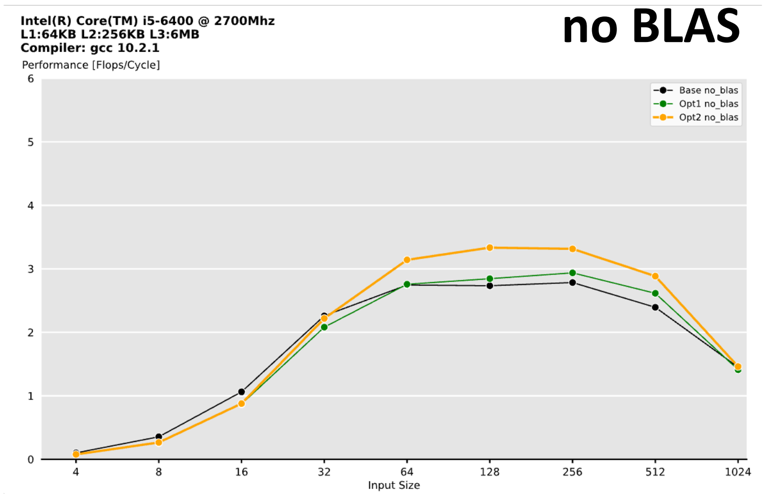


Basic optimizations

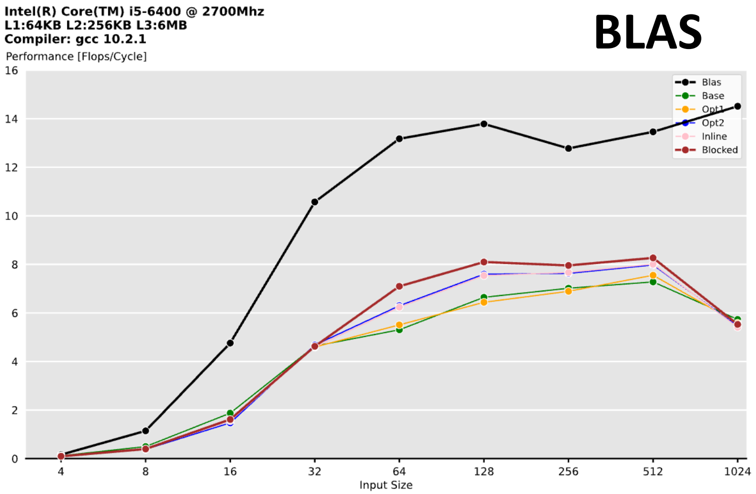
- **OPT-1 and OPT-2 modifications involved the following scalar optimizations:**
  - Ensure the constant use of arrays
  - Save precomputed data and reused after
  - Removed calls to abs(), ceil() or divisions
  - ILP improvements, unrolling and scalar replacement
  - Function inlining

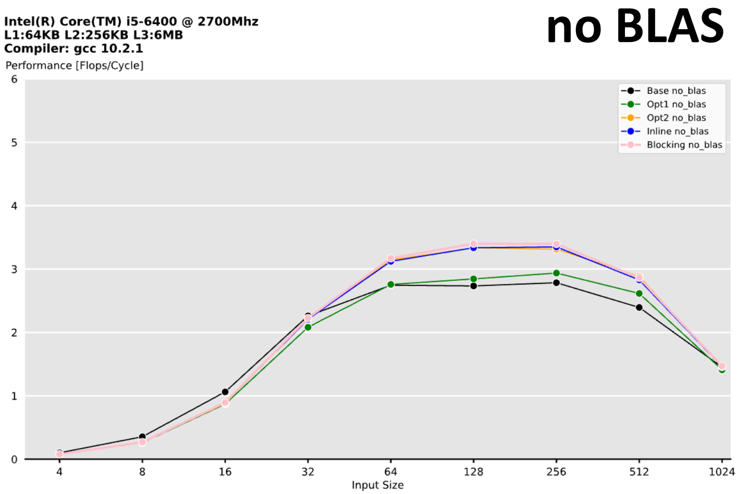
Basic optimizations



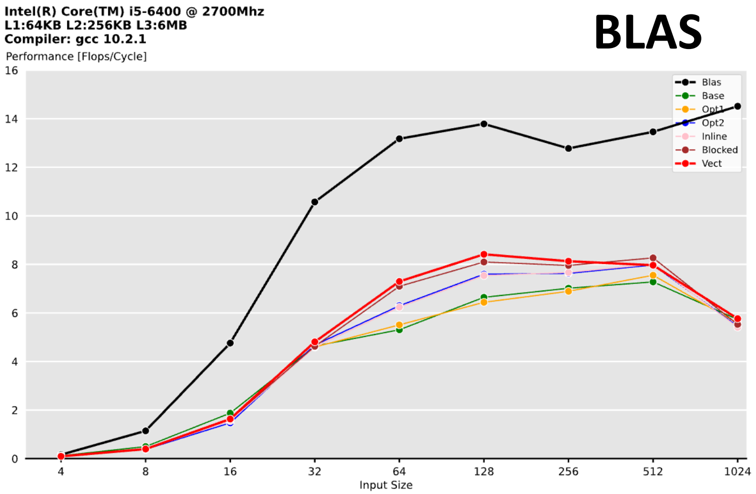


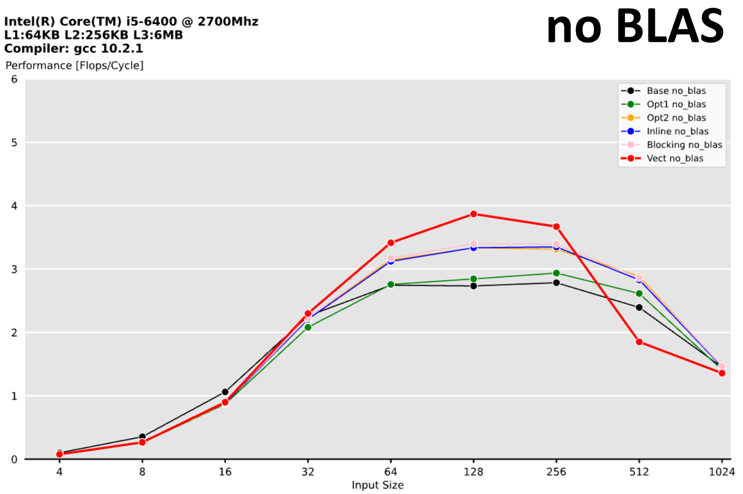
Blocking Plot



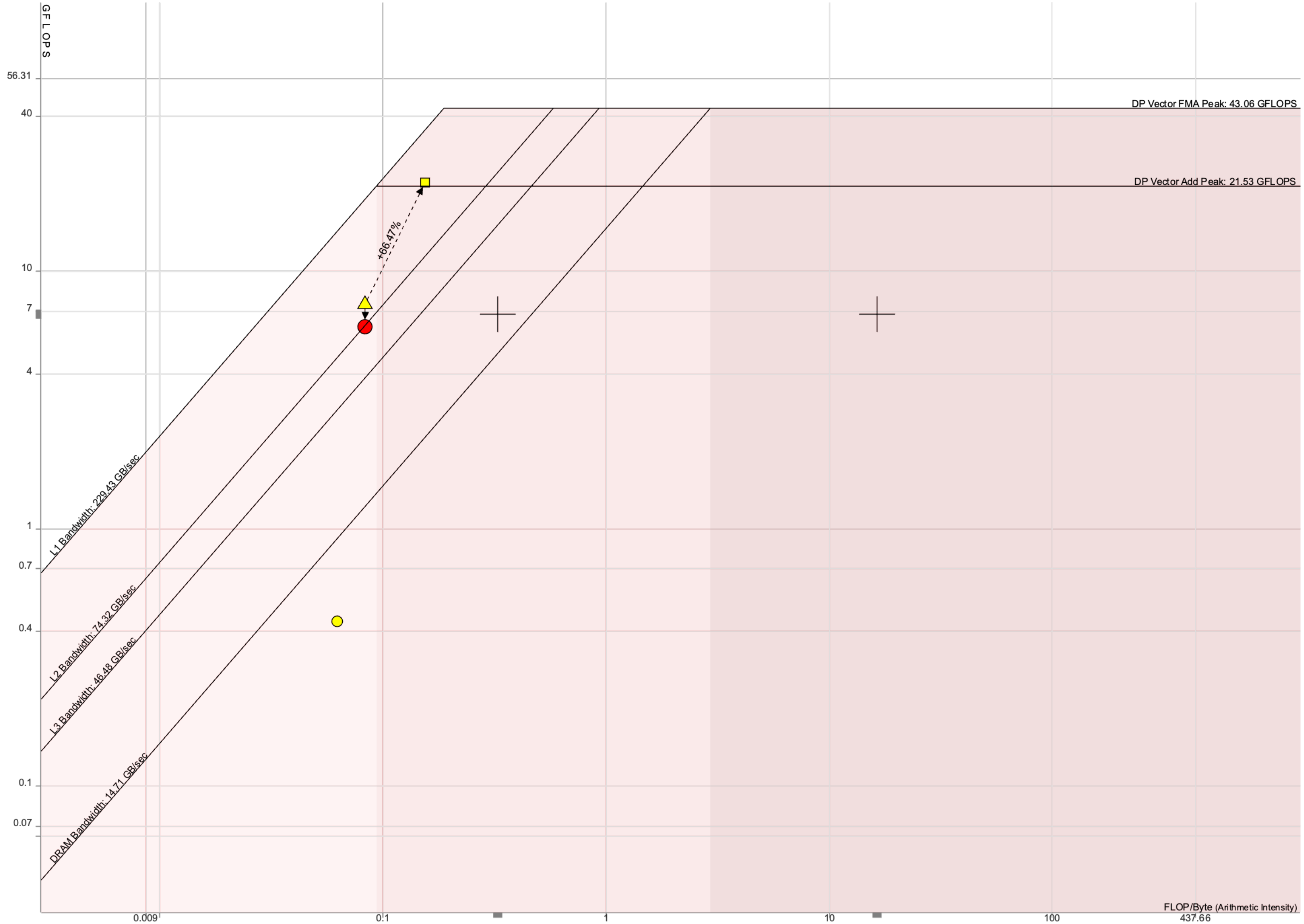


Vectorization Plot





Roofline plot *inverse (Base, BB and BV)*



Gaussian Elimination

- **The straightforward implementation used Gaussian Elimination to solve the system QX=P, where all matrices are NxN.**
- **Limitation**
  - No possible blocking in Gaussian Elimination
  - LU decomposition is the matrix form of Gaussian Elimination
- **In LU decomposition computations can be reused by backward and forward substitution applied to every column of P.**

Comparison

- **Gaussian Elimination optimizations**
  - Basic optimizations
    - *Inline swap_rows and avoid repeated calculations*
    - *Strength reduction*
  - Vectorization
- **LU decomposition optimizations**
  - Basic optimizations
    - *Loop reordering provided a major improvement (x10)*
  - Blocking
  - Vectorization

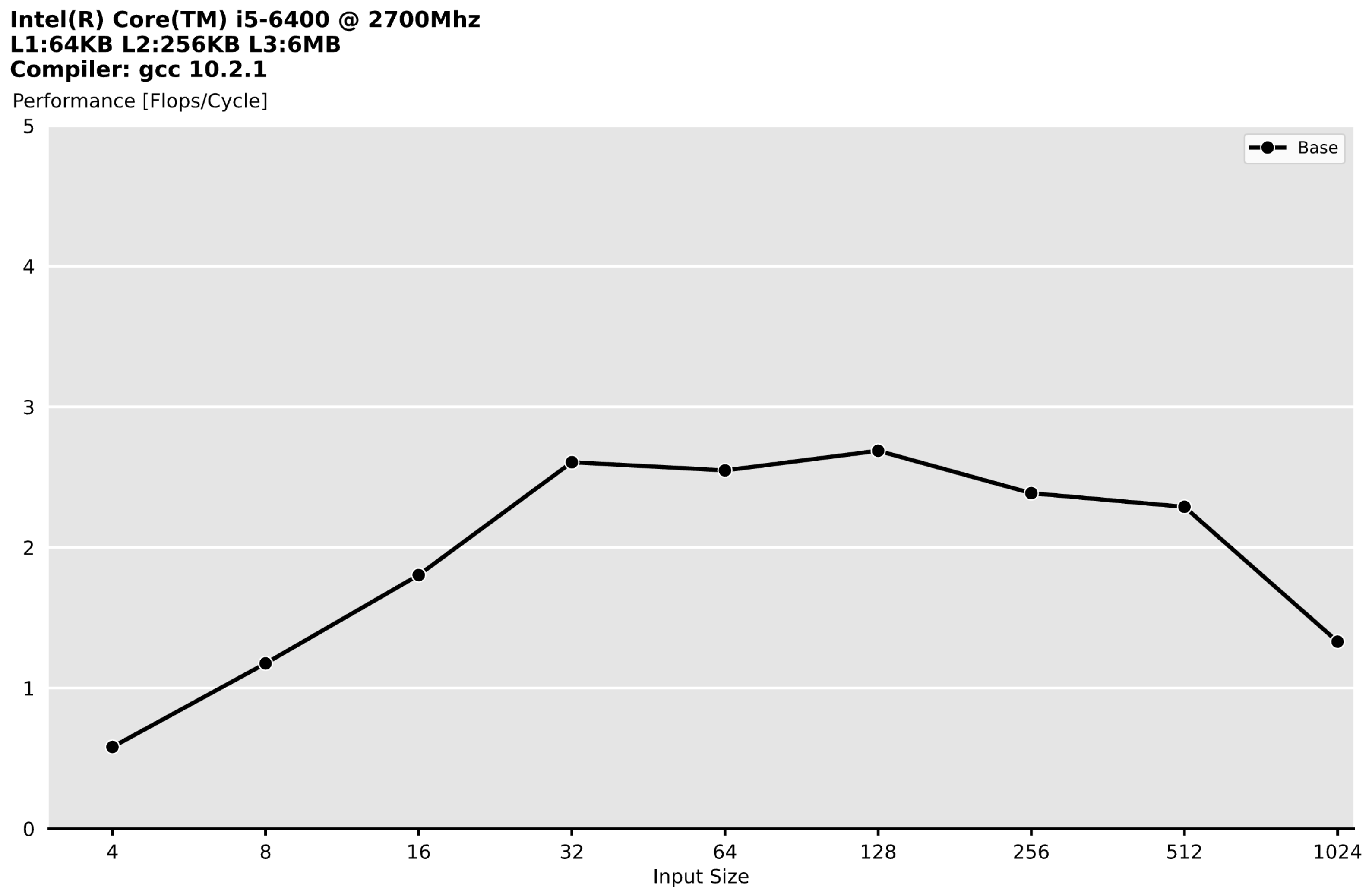
Loop ordering

- **Improved spatial and temporal locality**

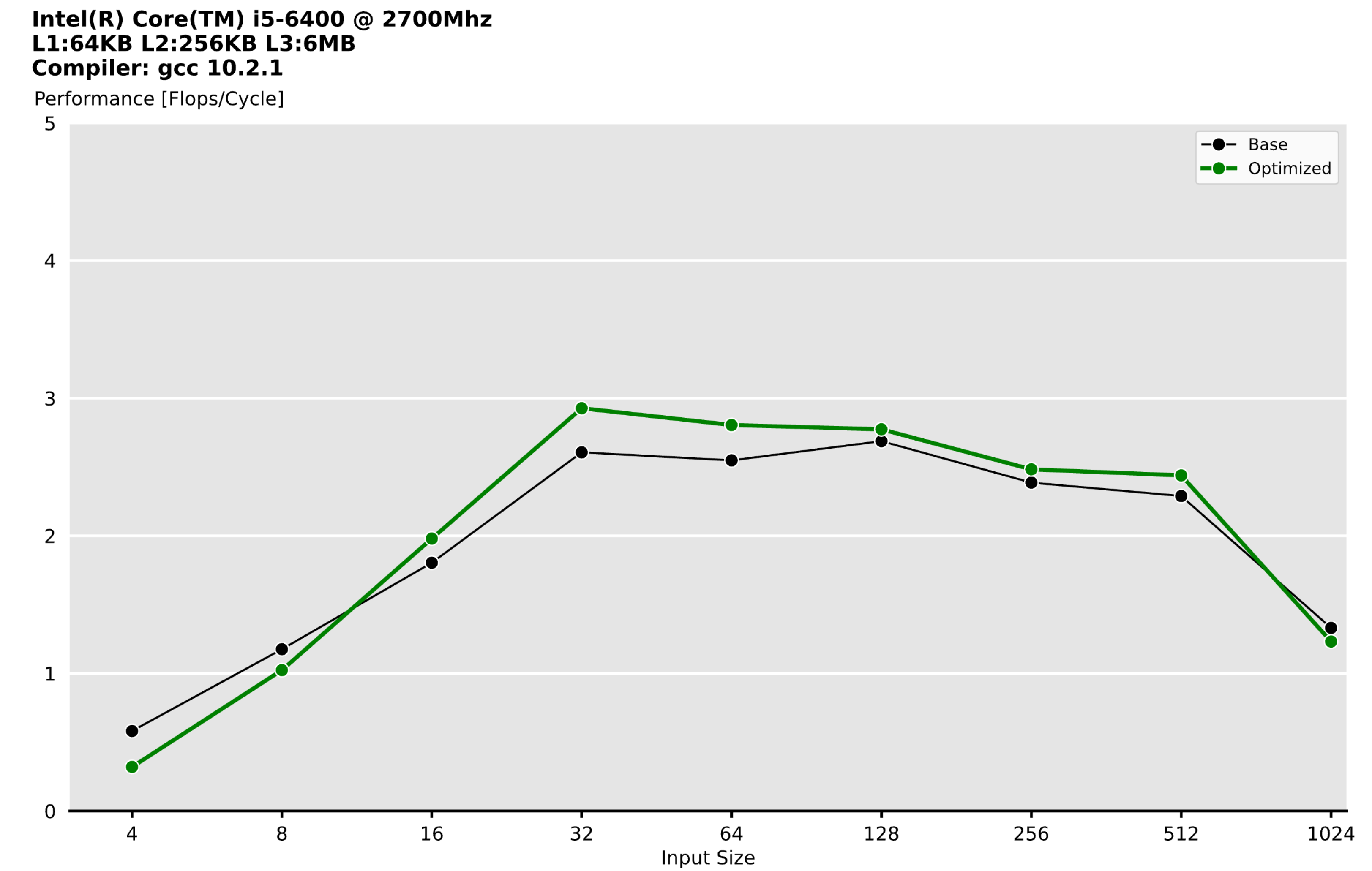
```
// Forward substitution
for (int i = 0; i < n; i++){
    for (int j = 0; j < n; j++){
        for (int k = 0; k < j; k++){
            P[j*n + i] -= P[k*n + i] * Q[j*n + k];
        }
    }
}

// Backward substitution
for (int i = 0; i < n; i++){
    for (int j = n - 1; j >= 0; j--){
        double temp = P[j*n + i];
        for (int k = j + 1; k < n; k++){
            temp -= P[k*n + i] * Q[j*n + k];
        }
        P[j*n + i] = temp / Q[j*n + j];
    }
}
```

```
// Forward substitution
for (int j = 0; j < n; j++){
    for (int k = 0; k < j; k++){
        for (int i = 0; i < n; i++){
            P[j*n + i] -= P[k*n + i] * Q[j*n + k];
        }
    }
}

// Backward substitution
for (int i = n - 1; i >= 0; i--) {
    for (int k = n - 1; k > i; k--) {
        for (int j = 0; j < n; j++) {
            P[i*n + j] -= Q[i*n + k] * P[k*n + j];
        }
    }
    for (int j = 0; j < n; j++) {
        P[i*n + j] /= Q[i*n + i];
    }
}
```

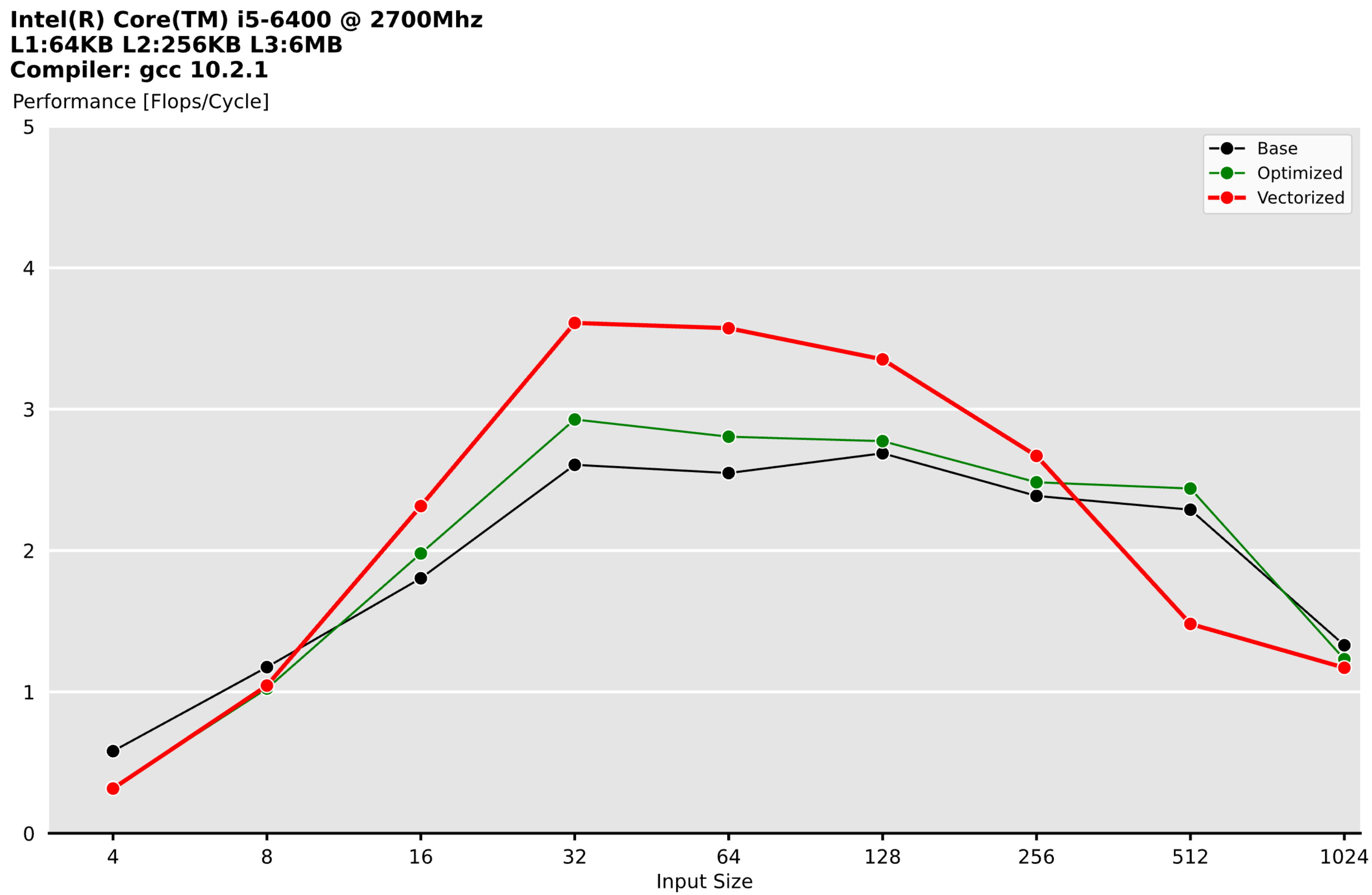
Gaussian Elimination Performance Plot



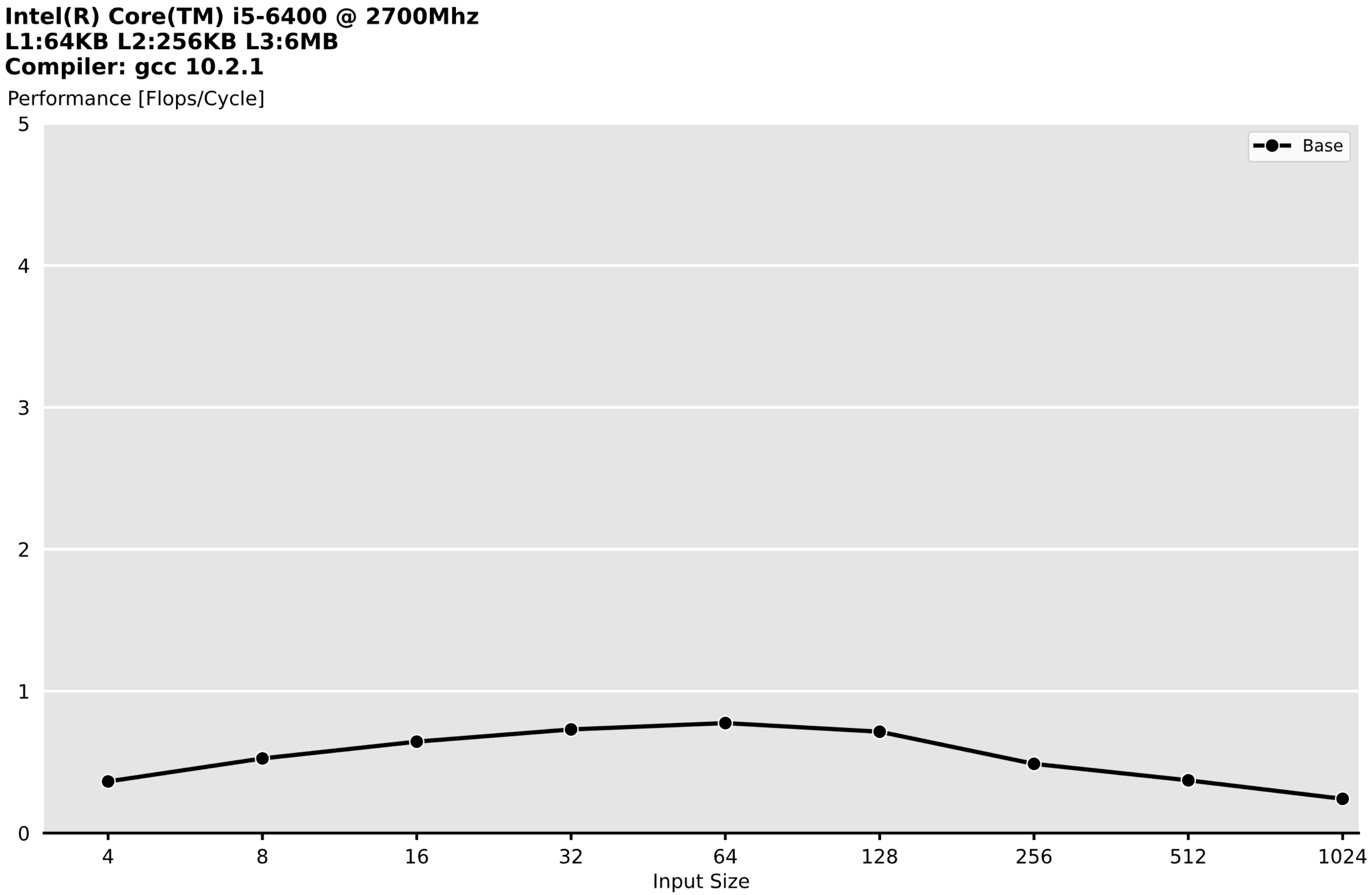
Gaussian Elimination Performance Plot



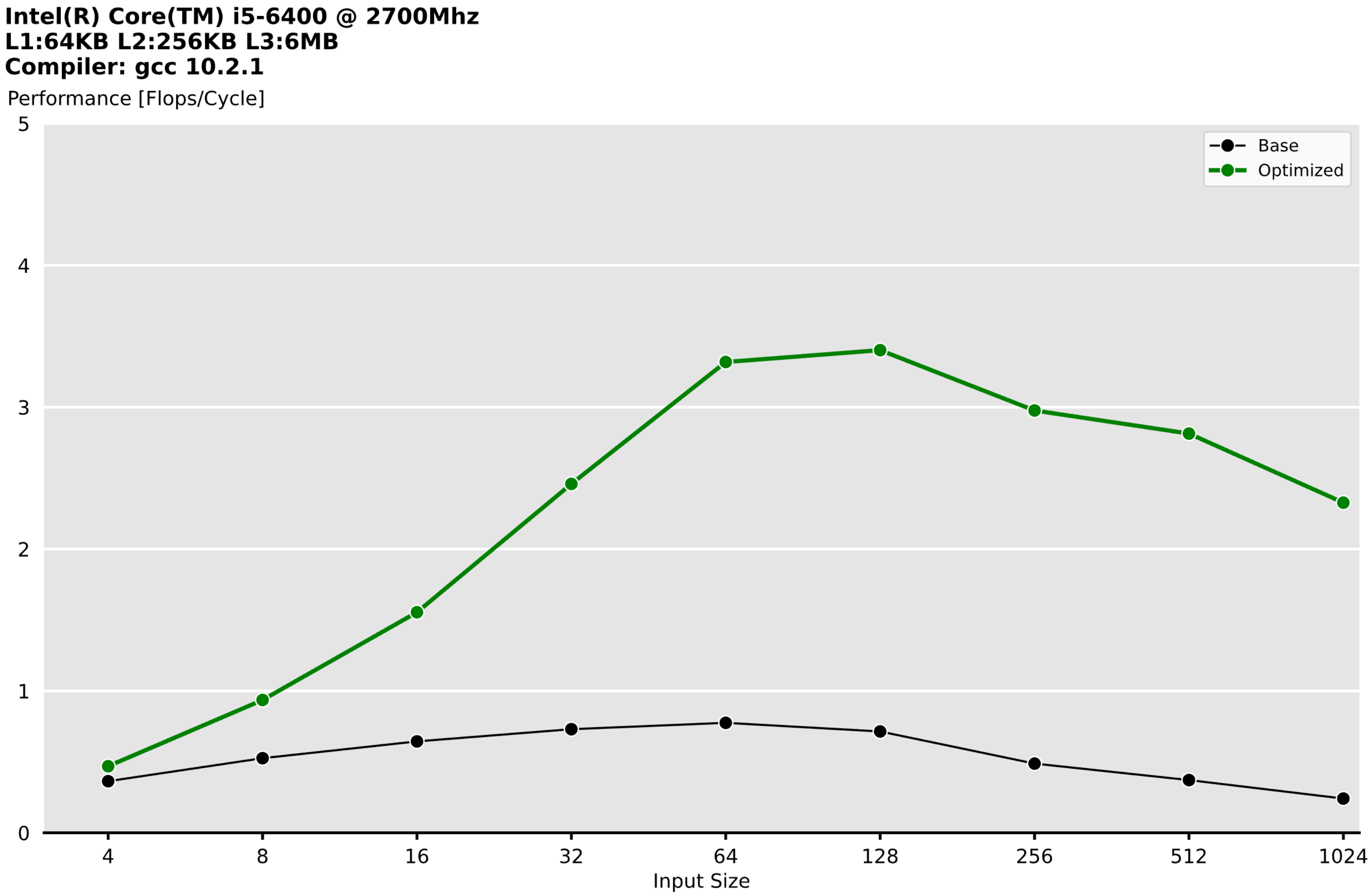
Gaussian Elimination Performance Plot



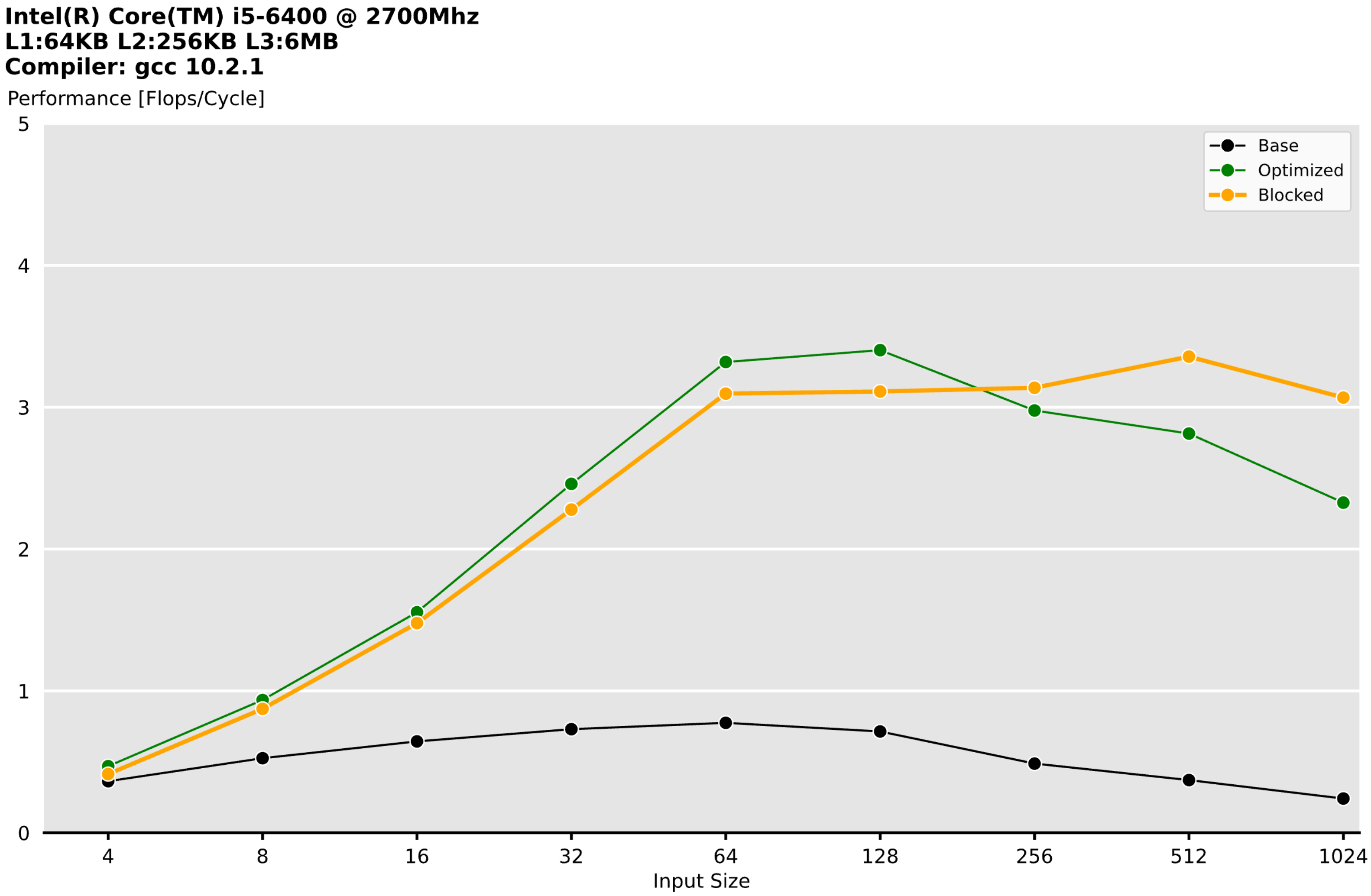
LU Decomposition Performance Plot



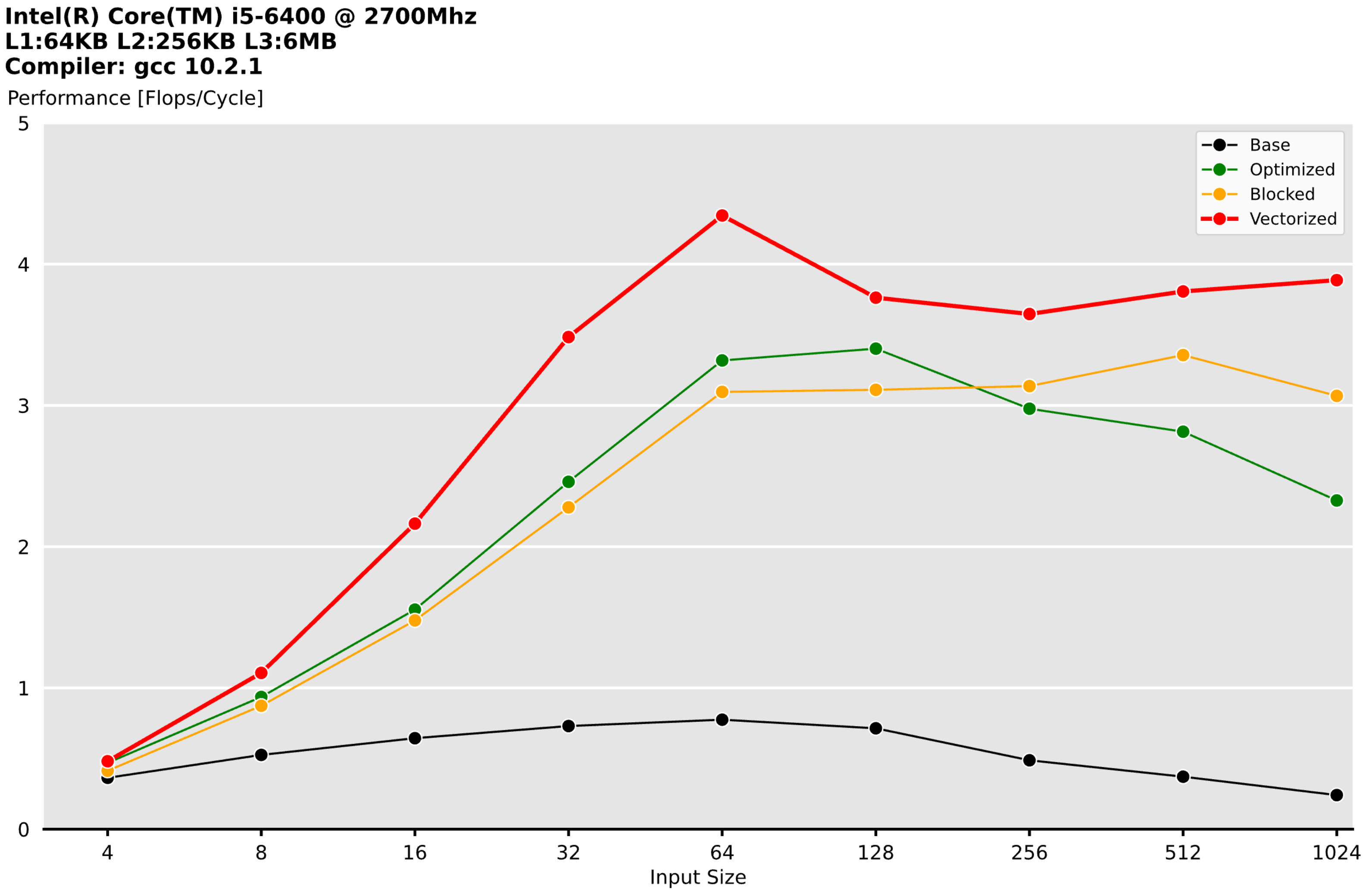
LU Decomposition Performance Plot



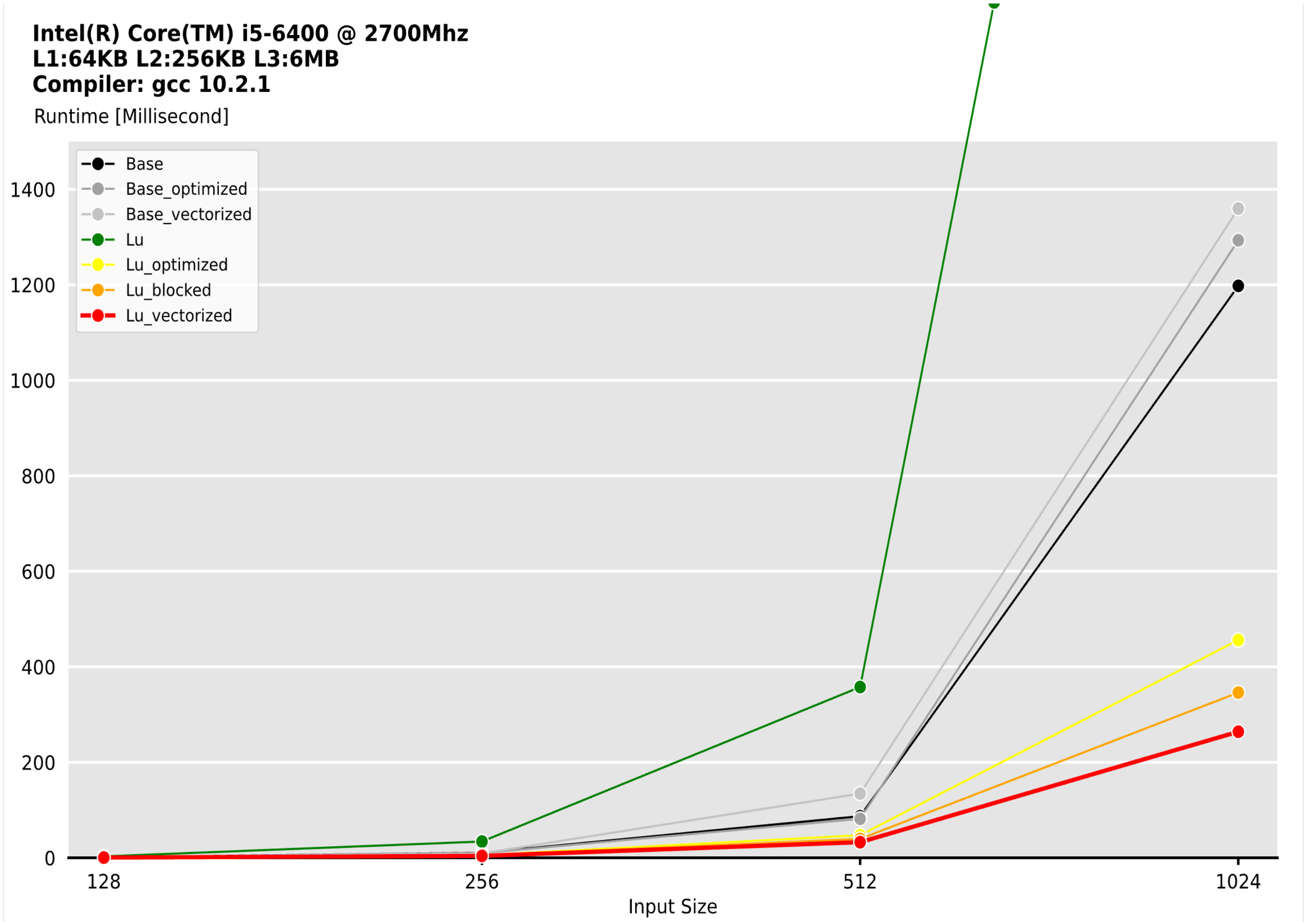
LU Decomposition Performance Plot



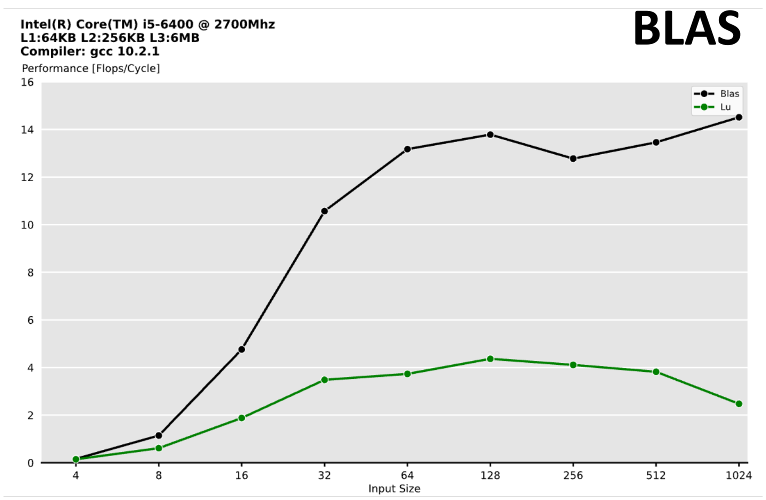
LU Decomposition Performance Plot



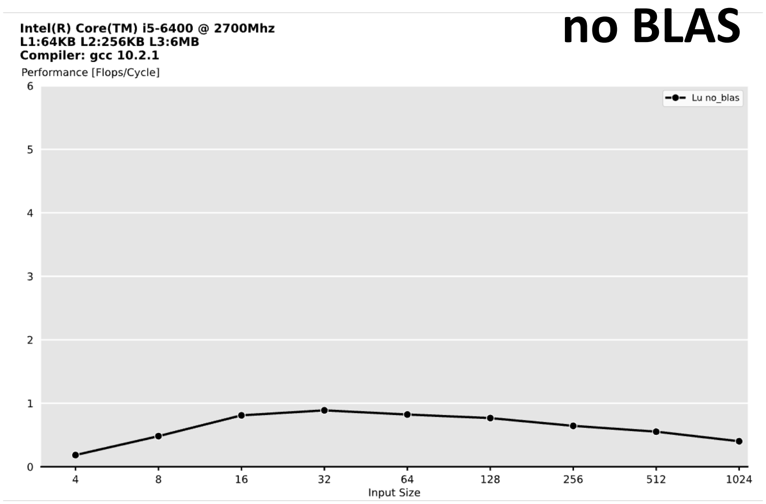
Runtime LU vs GE



Mexp-LU - Base

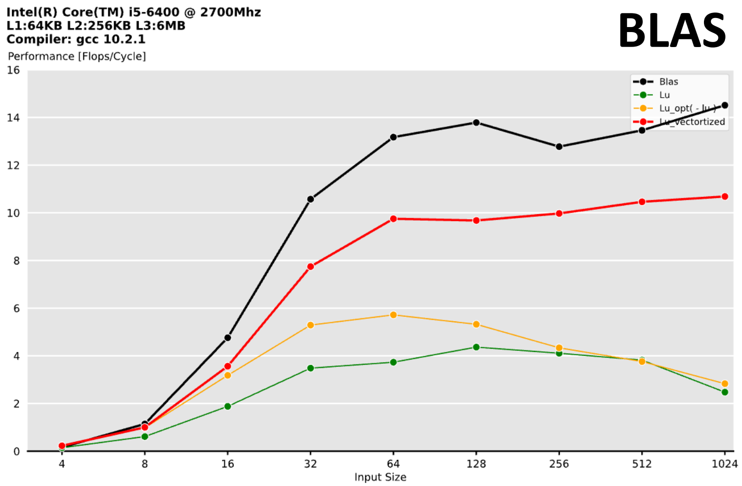


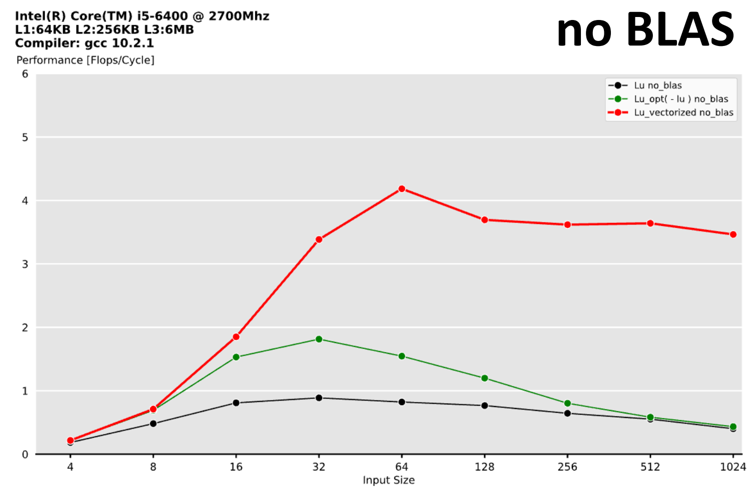
BLAS
Intel(R) Core(TM) i5-6400 @ 2700Mhz
L1:64KB L2:256KB L3:6MB
Compiler: gcc 10.2.1
Performance [Flops/Cycle]
Blas
Lu
Input Size



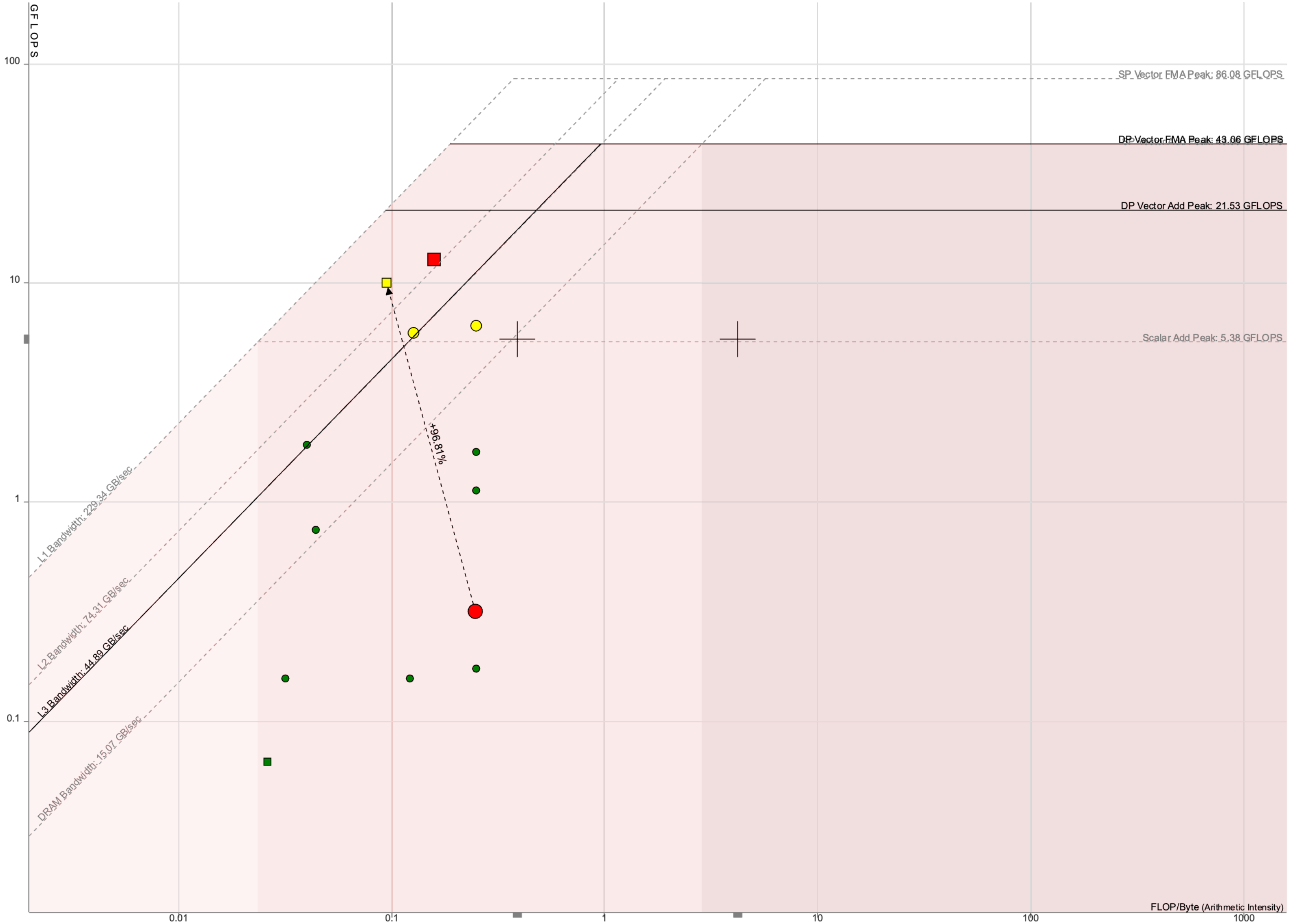
no BLAS
Intel(R) Core(TM) i5-6400 @ 2700Mhz
L1:64KB L2:256KB L3:6MB
Compiler: gcc 10.2.1
Performance [Flops/Cycle]
Lu no_blas
Input Size

Mexp-LU - Vectorised

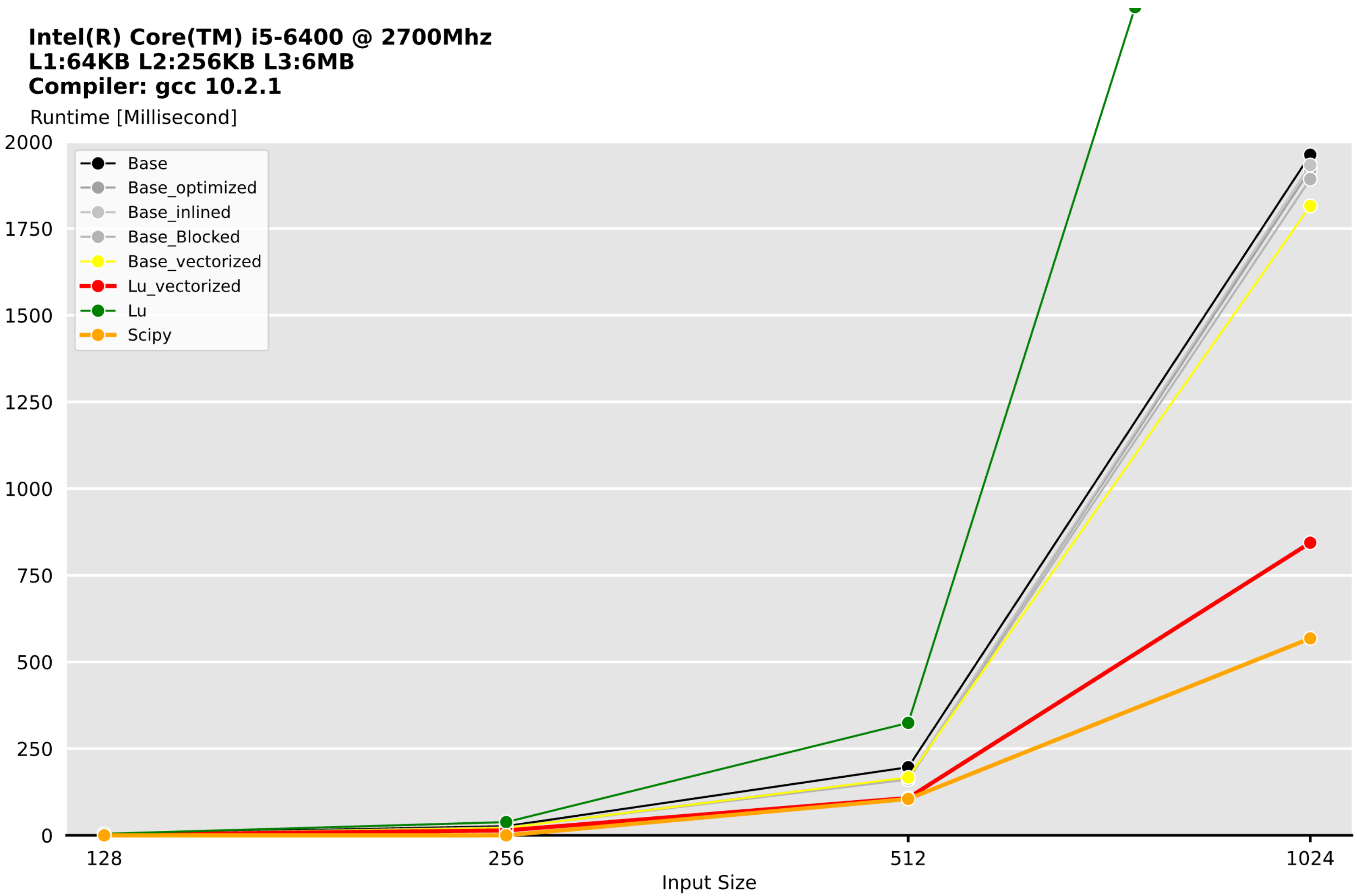




Roofline plot LU

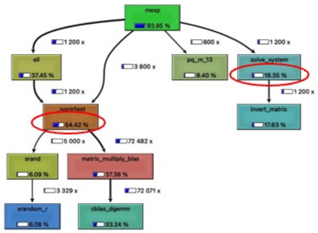


Benchmark runtime comparison



Callgrind analysis

- **Bottlenecks**
  - solve-system
  - normest



Loop ordering

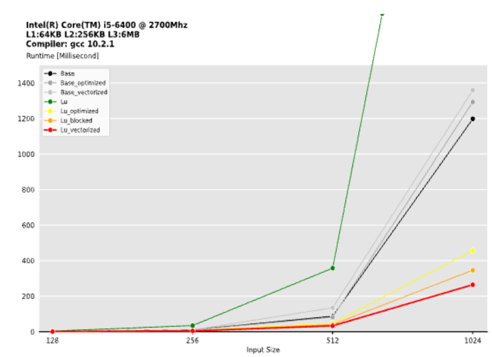
- **Improved spatial and temporal locality**

```
// Forward substitution
for (int i = 0; i < n; i++){
    for (int j = 0; j < n; j++){
        for (int k = 0; k < j; k++){
            P[j*n + i] -= P[k*n + i] * Q[j*n + k];
        }
    }
}

// Backward substitution
for (int i = 0; i < n; i++){
    for (int j = n - 1; j >= 0; j--){
        double temp = P[j*n + i];
        for (int k = j + 1; k < n; k++){
            temp -= P[k*n + i] * Q[j*n + k];
        }
        P[j*n + i] = temp / Q[j*n + j];
    }
}
```

```
// Forward substitution
for (int j = 0; j < n; j++){
    for (int k = 0; k < j; k++){
        for (int i = 0; i < n; i++){
            P[j*n + i] -= P[k*n + i] * Q[j*n + k];
        }
    }
}

// Backward substitution
for (int i = n - 1; i >= 0; i--) {
    for (int k = n - 1; k > i; k--) {
        for (int j = 0; j < n; j++) {
            P[i*n + j] -= Q[i*n + k] * P[k*n + j];
        }
    }
    for (int j = 0; j < n; j++) {
        P[i*n + j] /= Q[i*n + i];
    }
}
```

Q & A

Runtime LU vs GE



Roofline plot LU