# BLAS

Christoph Ortner
Stef Salvini

## The *BLAS*ics

- **B**asic **L**inear **A**lgebra **S**ubroutines
  - Building blocks for more complex computations
  - Very widely used
- Level means “number of operations”
  - Level 1: vector-vector operations – O($n$) operations
  - Level 2: matrix vector operations – O($n^2$) operations
  - Level 3: Matrix-Matrix operations - O($n^3$) operations
- A *Flop* is any numerical operation
  - Adds, Mults, divisions, square roots (!!!!), etc
    - Of course divisions & square roots are more expensive …
  - Loads/stores are not taken into account (history … )
- BLAS provide a good basis to understand performance issues

## A Fistful of Flops

- BLAS take off with **Vector** processors (70s – 80s)

- Level 1 first, then level 2 BLAS
  - Encapsulate expert knowledge
  - Efficient building blocks
  - “Local” optimisation of code enough to increase performance

## Level 1 BLAS

- O(*n*) operands (I.e. load/stores), operations O(*n*) (flops)

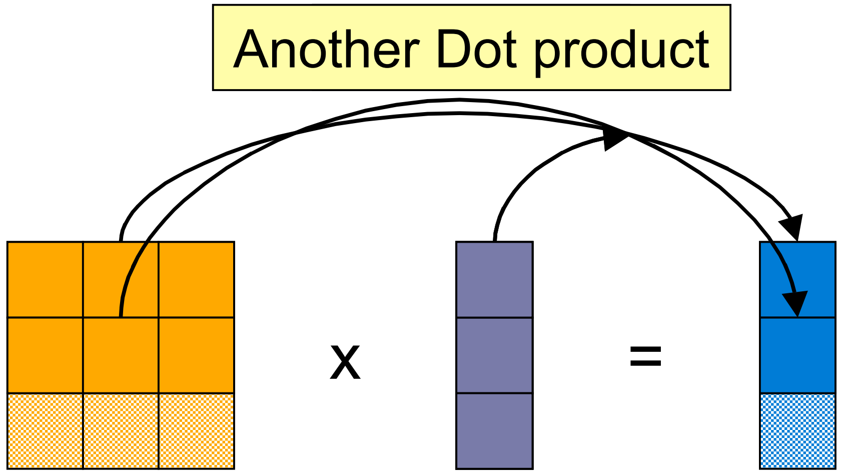
- Vector operations (loved by vector processors)
  - Ratio between load/stores and operations: O(1)
  - E.g. “axpy” : $\alpha\, x + y$ ◊ $y$

- Reduction operations (hated by vector processors)
  - Ratio between load/stores and operations: O(*n*)
  - E.g. dot product: $\alpha = x^T\ y$

- Available:
  - Single & double precision, real and complex
    - Names beginning with S, D, C and Z, respectively
    - Axpy: SAXPY, DAXPY, CAXPY, ZAXPY
    - Dot Products (SDOT, DDOT, CDOTC, ZDOTC)

## Level 2 BLAS

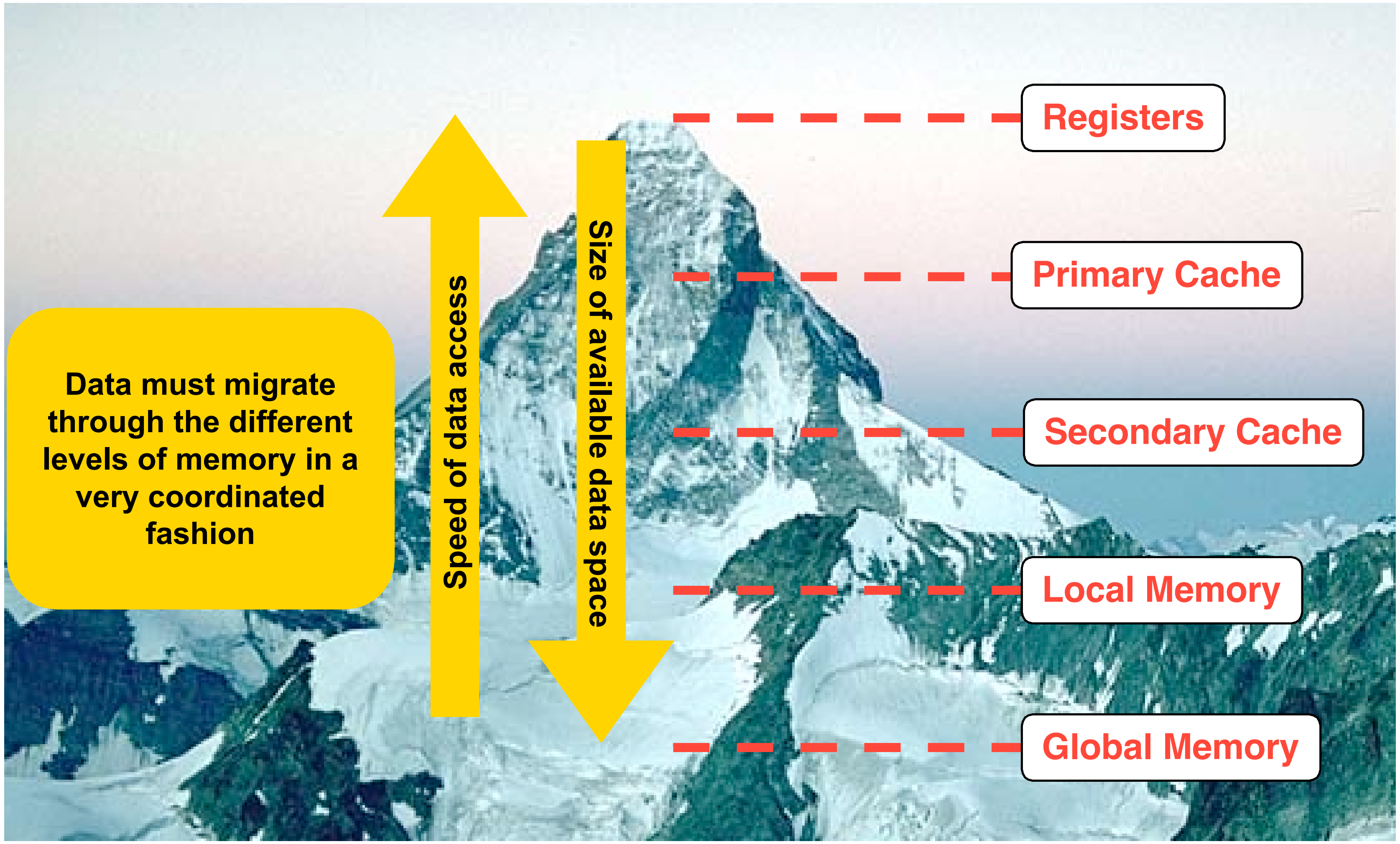
- $O(n^2)$ operands, $O(n^2)$ operations
- Performance can be understood in terms of Level 1 cache
- Matrix-vector product, matrix updates, solution of a triangular system of equations, etc



## Superscalar takes over

- Technology dictated by
  - Cost
  - Widespread use
  - Relatively small HPC market
- Superscalar here means more than one operation per cycle
- All supercalar architecture (give-or-take)
  - No direct access to memory
    - Hierarchical memory layout
    - Use of caches to make use of any data locality
  - **Rule-of-thumb for performance:**
    - *“Thou shalt not have as many operands as operations”*
    - In fact: poor performance of Level 1 and 2 BLAS (sometimes horrifyingly so)
    - Poor performance for indirect addressing
    - FFTs very difficult

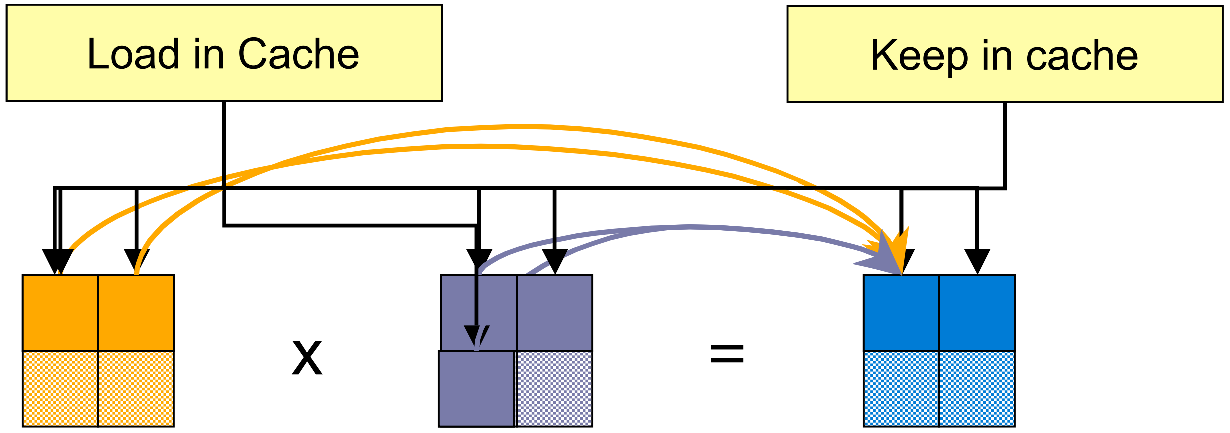
## The Data View of an SS Architecture



Data must migrate through the different levels of memory in a very coordinated fashion

# For A Few Flops More

- Level 3 BLAS (matrix-matrix operation) to the rescue!
- Why Level 3 BLAS are *good* (example of matrix-matrix product)
  - Use blocked algorithms to maximise cache reuse
  - O($b^2$) loads/stores – O($b^3$) flops
  - Enough operations to “hide” memory traffic costs



## The Good the Bad and the Ugly

- Lots of Packages depend on and benefit from BLAS
  - LAPACK (Linear Algebra)
  - Many Sparse Solvers (using local dense blocks of sparse matrices, such as SuperLU, MUMPS, etc)
- A Myth
  - BLAS are parallelised by vendors, hence all LAPACK etc is parallel and scalable – NOT TRUE!
    - Level 1 BLAS: NEVER parallelised
    - Level 2 BLAS: SOME parallelised
    - Level 3 BLAS: ALL parallelised
- Most codes do not contain the nice packed aligned data that BLAS require (indirect addressing on SS architectures very tough!)
- What about SSE & SSE2 on Intel & AMD architectures
  - They are for multimedia!
    - Pack several words (numbers) in register
    - Operate simultaneous on all words in register
    - Operations crossing low & high in register very expensive! (What about complex numbers: well, they do not exist for vendors)

## Sparse BLAS

**Slide Intentionally left (almost) Blank**

Zillions of sparse formats

Efforts to generate sparse BLAS automatically (performance poor – indirect addressing)

## Getting the BLAS

- "Model" BLAS
  - Model implementation in Fortran
  - No optimization in source
  - Some compilers can block Level 3-BLAS approaching level of more sophisticated implementations (only DGEMM)
  - C interface is available
- Vendor BLAS
  - Hand-optimized by vendors (IESSL/IBM, MKL/Intel, ACML/AMD, …)
  - Achieves highest performance on vendors' platforms.
  - **YOU SHOULD USE THIS!**
- ATLAS
  - "Automatically Tuned Linear Algebra Software"
  - Brute force optimization
    - trying out all possible combinations of memory layout, loop reordering, etc.
  - Competitive performance on Level 3 BLAS
  - Can be generated for virtually all platforms

## The “Mythical” Goto BLAS

- BLAS designed by **Kazushige Goto**
- Optimizes all memory traffic in a very clever way
- Currently beats most commercial libraries
- Only few non-threaded BLAS
- http://www.tacc.utexas.edu/resources/software/

## Measuring Performance

- Performance is measured in Mflops/s
- E.g.: multiplication of two square N x N matrices (DGEMM)
  - $N^3$ multiplications and
  - $N^3$ additions
  - = 2 $N^3$ flops
- t seconds for m `dgemm` calls gives

$$\frac{2n^3m}{10^6t} \text{ Mflops}$$

## Benchmarks

- 1. Loop to determine number **m** of calls to run for **T** seconds (say **T = 1.0**)
  - This loop does many timings which is a significant overhead
- 2. Time **m** calls
- 3. Repeat step 2 several times and take best timing

## The henrici Systems

- Intel® Xeon™ CPU 3.20 GHz
- 512 KB L2 cache
- 2 Processors with hyperthreading
- 2GB main memory

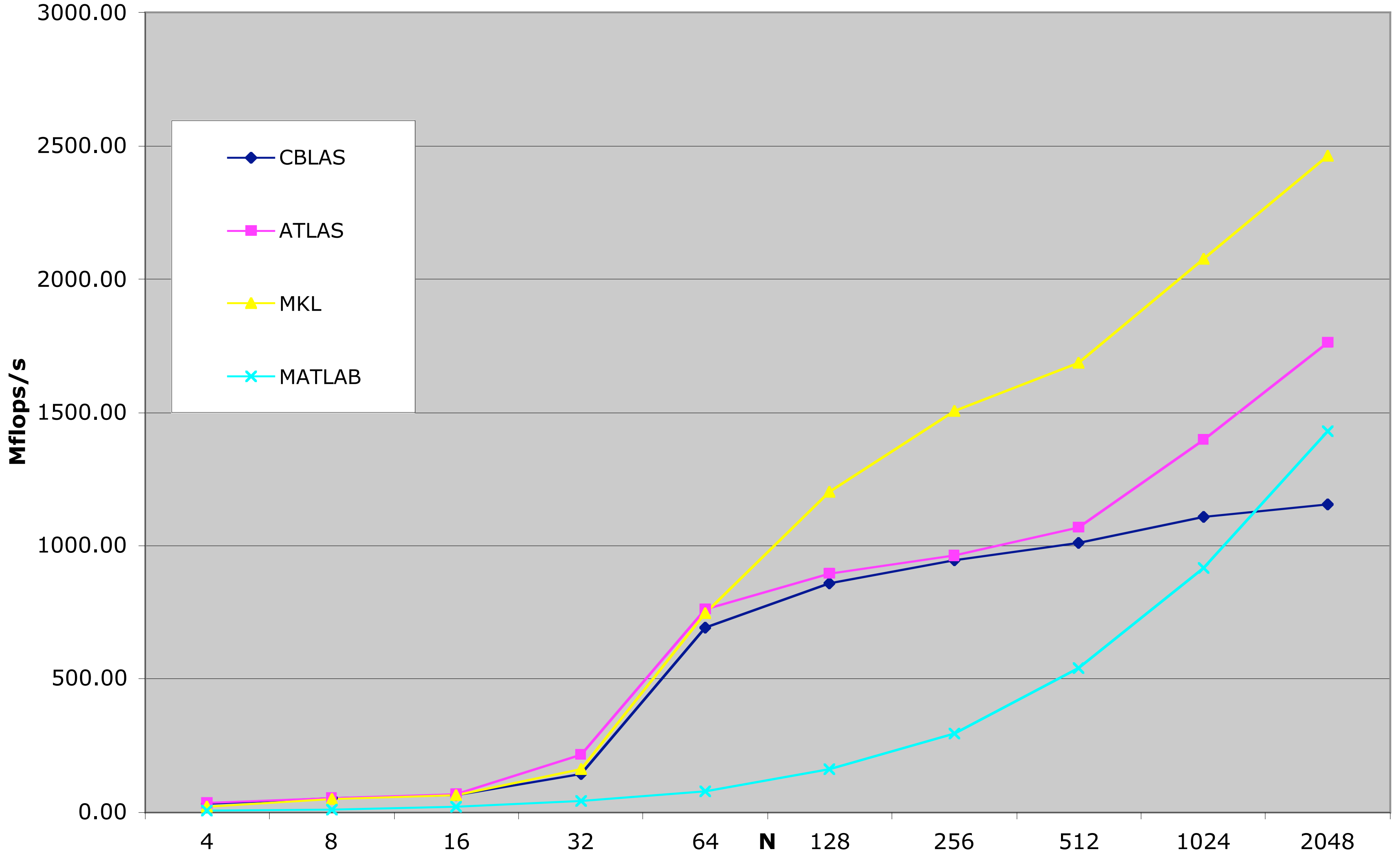
- (theoretical **serial** peak peformance:
  6400 MFLOPs/s)

## DDOT

- **`a = x' * y;`**

-
  ```
  double cblas_ddot(const int N, const double *X,
        const int incX, const double *Y, const int incY);
  ```
- 2 N operations + 2 N memory accesses

## DDOT Performance



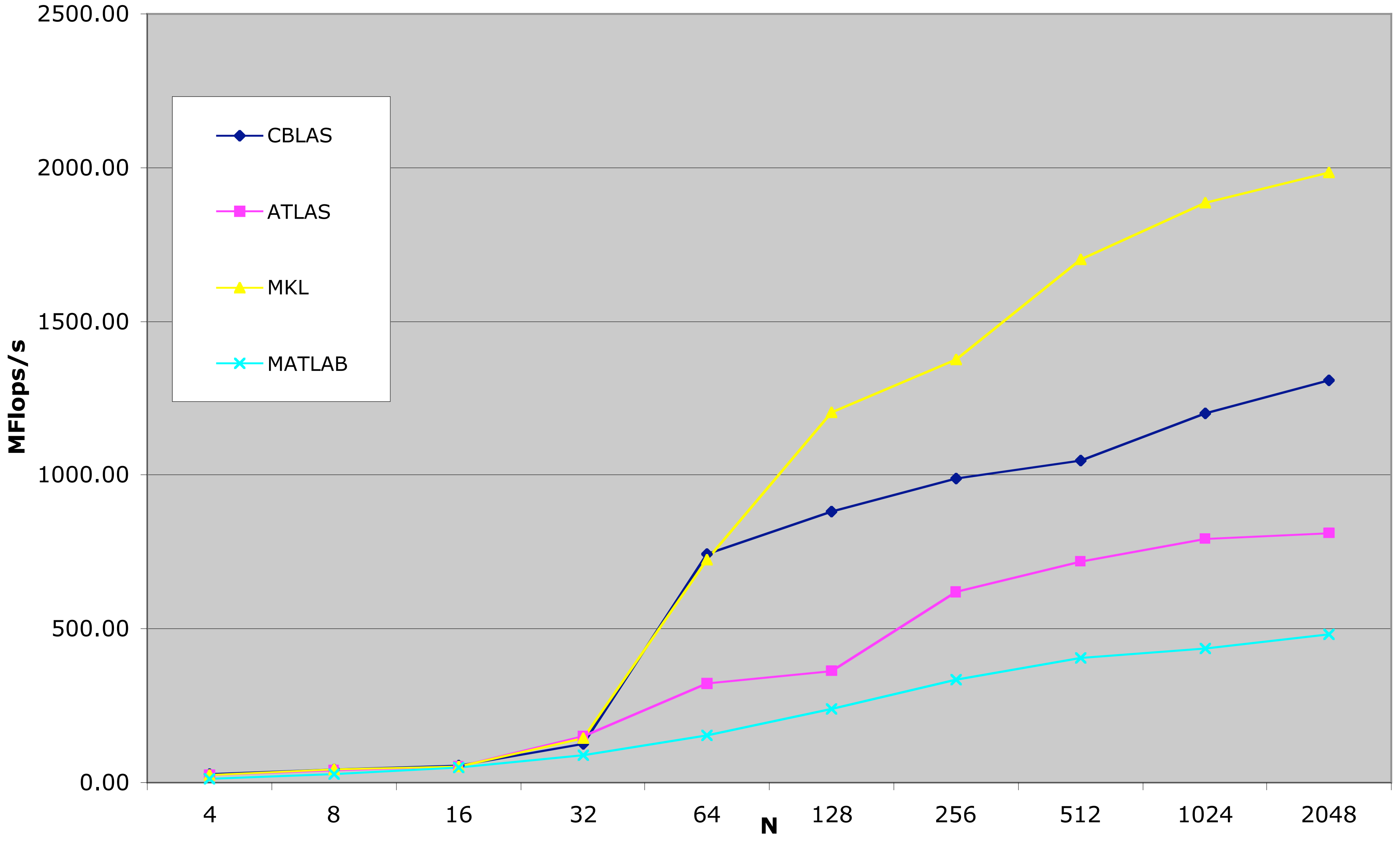
## DAXPY

- **`y = y + alpha * x;`**

-
  ```
  void cblas_daxpy(const int N, const double alpha,
                   const double *X, const int incX,
                   double *Y, const int incY);
  ```

- 2 N operations + 3 N memory accesses

## DAXPY Performance

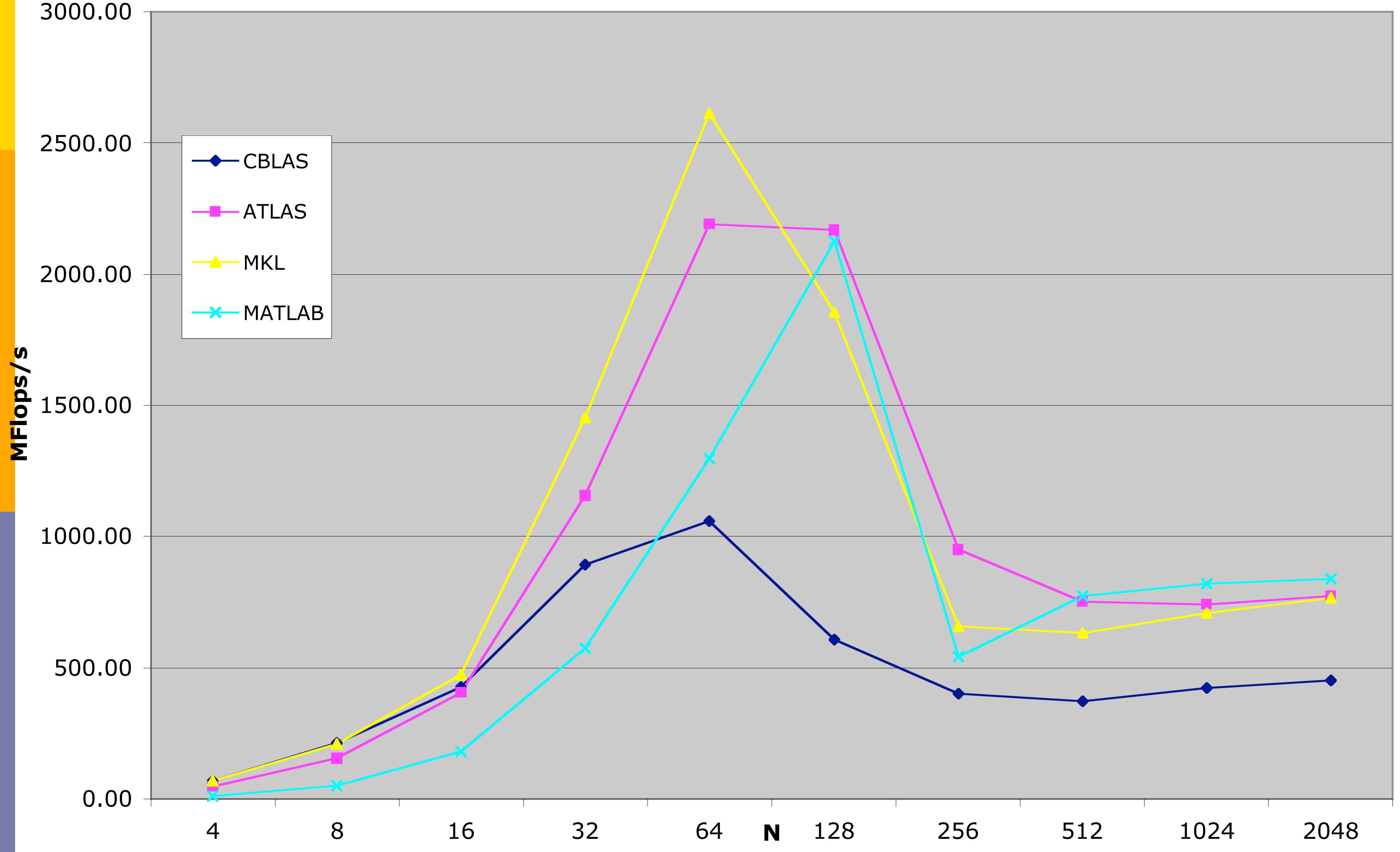


## DGEMV

- **`y = alpha * A * x + y;`**

-
  ```
  void cblas_dgemv(const enum CBLAS_ORDER Order,
                   const enum CBLAS_TRANSPOSE TransA, const int M,
                   const int N, const double alpha, const double *A,
                   const int lda, const double *X, const int incX,
                   const double beta, double *Y, const int incY);
  ```
- 2 $N^2$ operations + $N^2$ memory accesses

# DGEMV Performance



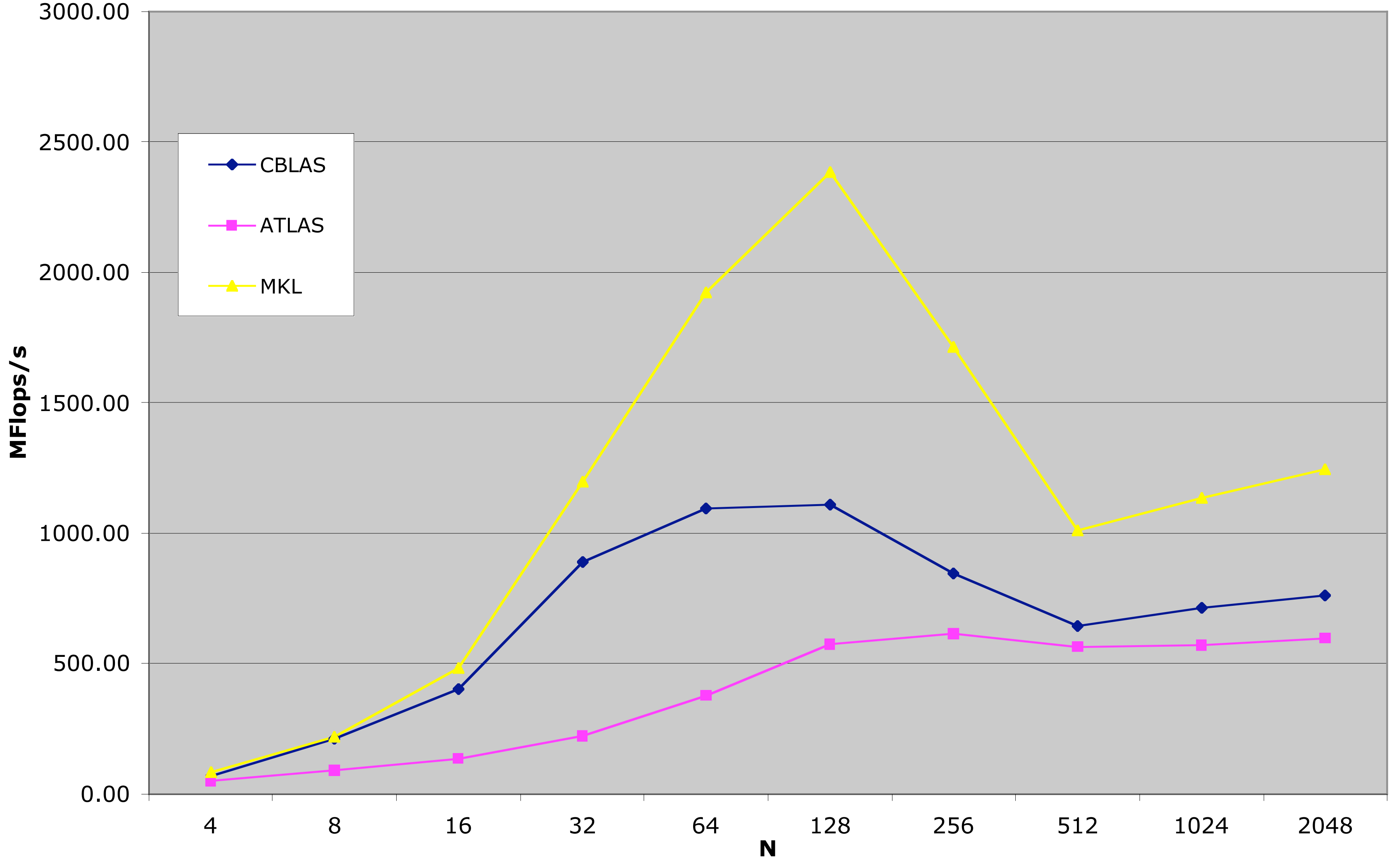
## DSYMV

- **`y = alpha * A * x + y;     % A symmetric`**

-
  ```
  void cblas_dsymv(const enum CBLAS_ORDER Order,
                    const enum CBLAS_UPLO Uplo, const int M,
                   const int N, const double alpha, const double *A,
                   const int lda, const double *X, const int incX,
                   const double beta, double *Y, const int incY);
  ```

- 2 $N^2$ operations + $N^2$ / 2 memory accesses

## DSYMV Performance



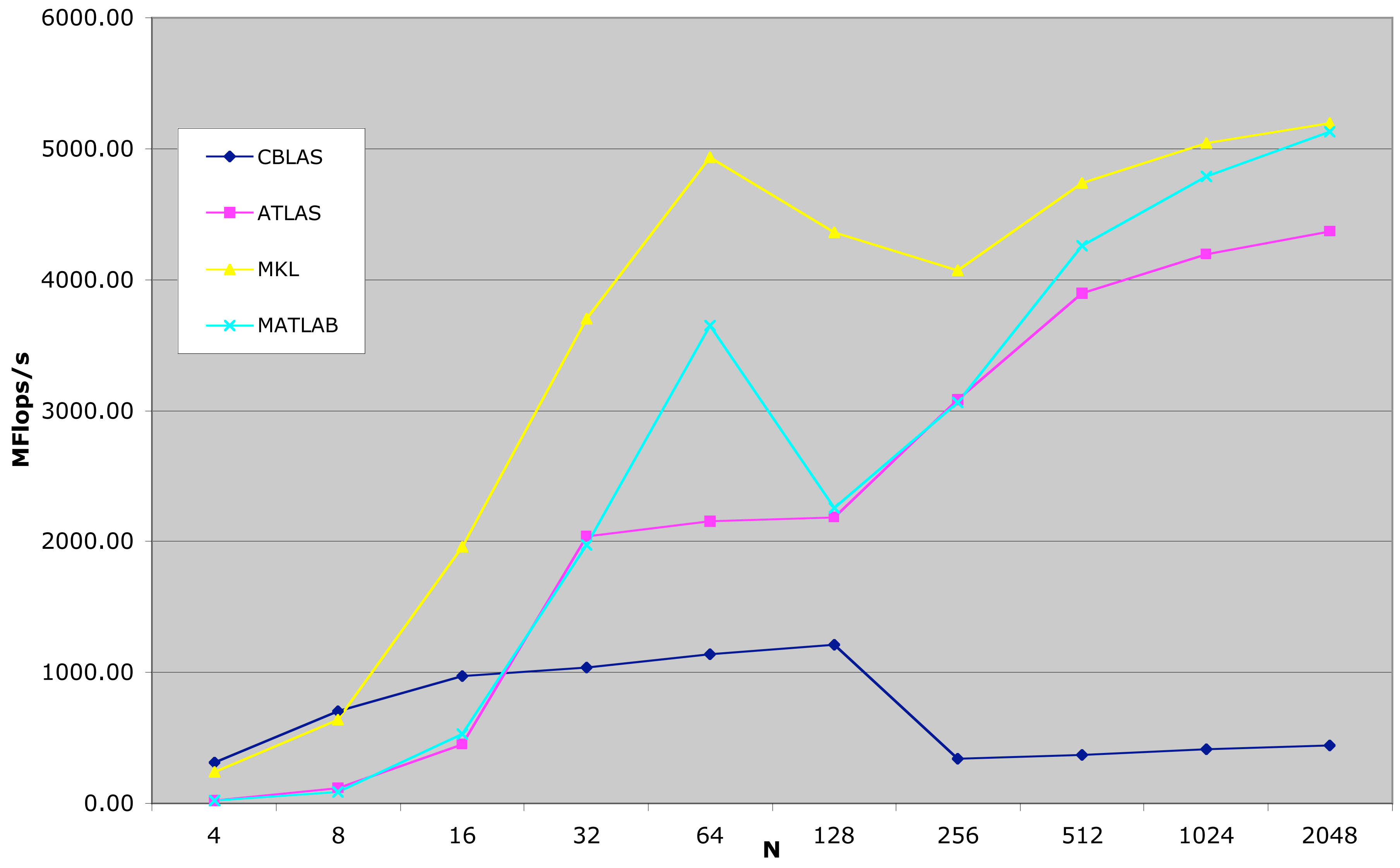
## DGEMM

- **`C = alpha * A * B + C;`**

-
  ```
  void cblas_dgemm(const enum CBLAS_ORDER Order,
     const enum CBLAS_TRANSPOSE TransA,
     const enum CBLAS_TRANSPOSE TransB,
     const int M, const int N, const int K,
     const double alpha, const double *A, const int lda,
     const double *B, const int ldb, const double beta,
     double *C, const int ldc);
  ```

- 2 $N^3$ operations + 3 $N^2$ memory accesses

# DGEMM Performance



## Blitz++

- C++ Array Library
- Fully object oriented and templated
- Supports operator overloading
- Level 1 BLAS via **Expression Templates**
- ‘Lightweight’ classes for small vectors or matrices
- Current Version: 0.9
  http://www.oonumerics.org/blitz/
- Similar Package: uBLAS (boost Library)

## Expression Templates

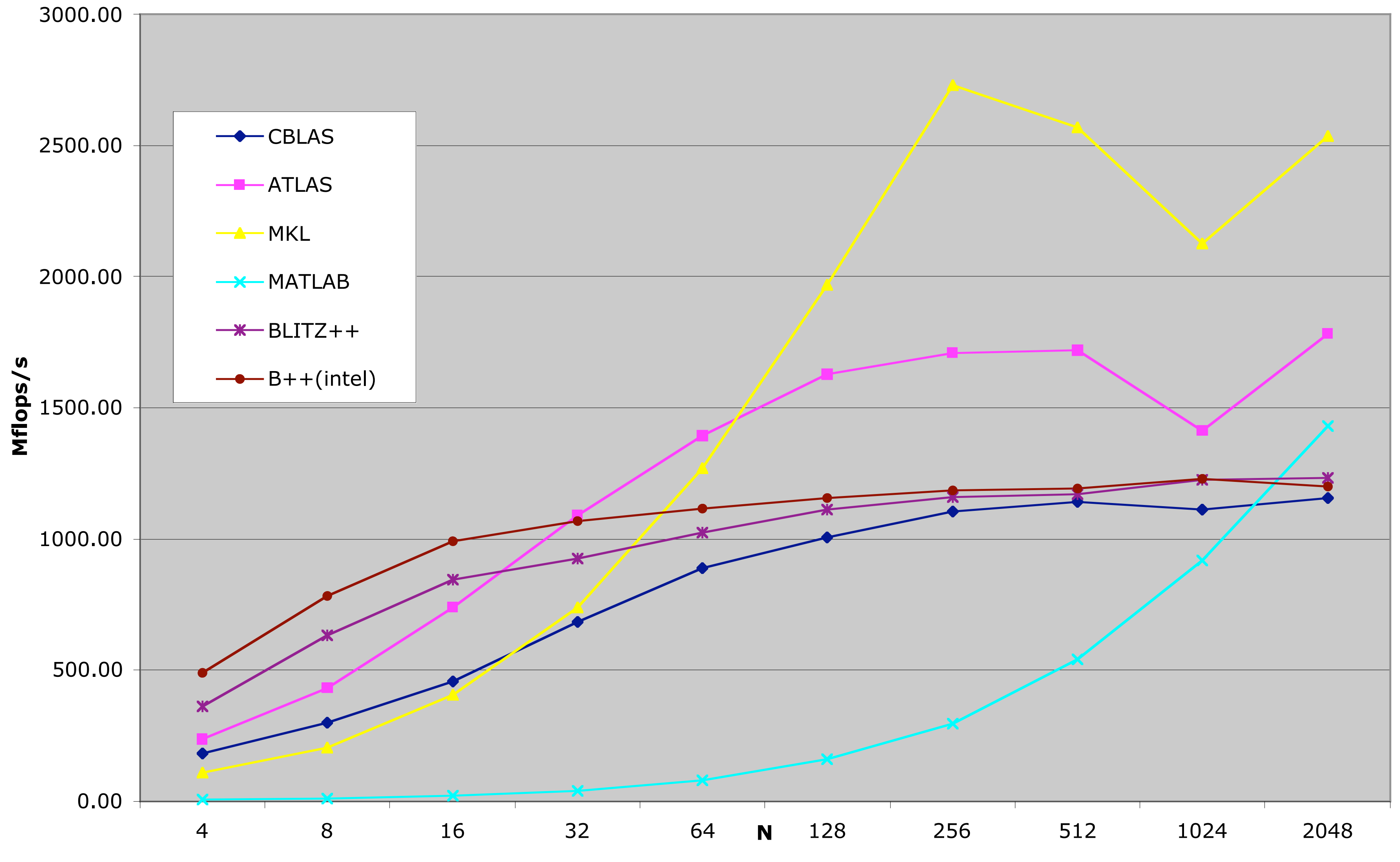
- daxpy: **`y = y + a x`**

- Expression Templates:

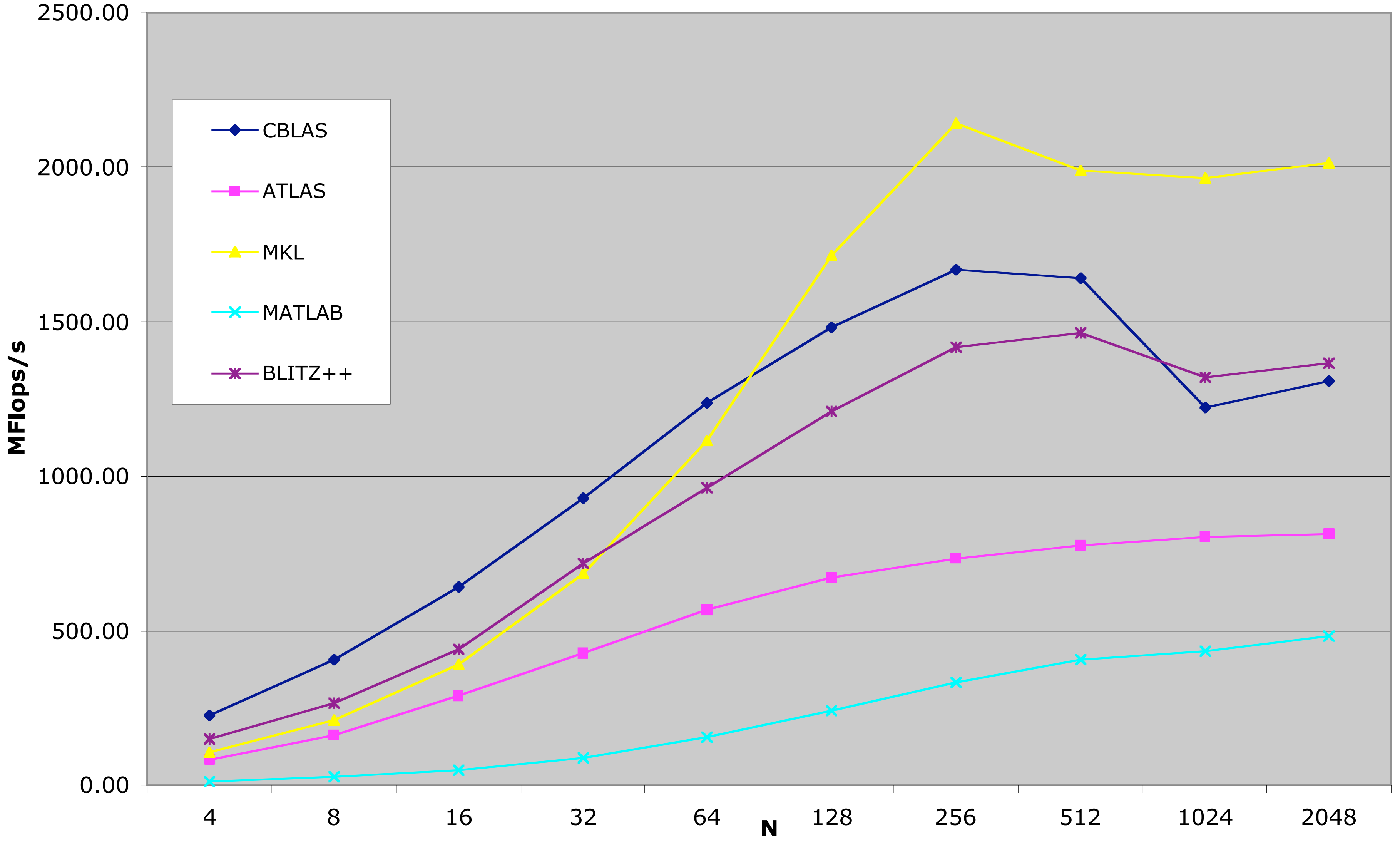
```
for i = 1:N
    y(i) = y(i) + a * x(i);
```

- Expression Templates create such loop for any possible operation of any possible Object (e.g. Arrays of short vectors for explicit codes)

## DDOT Performance



# DAXPY Performance



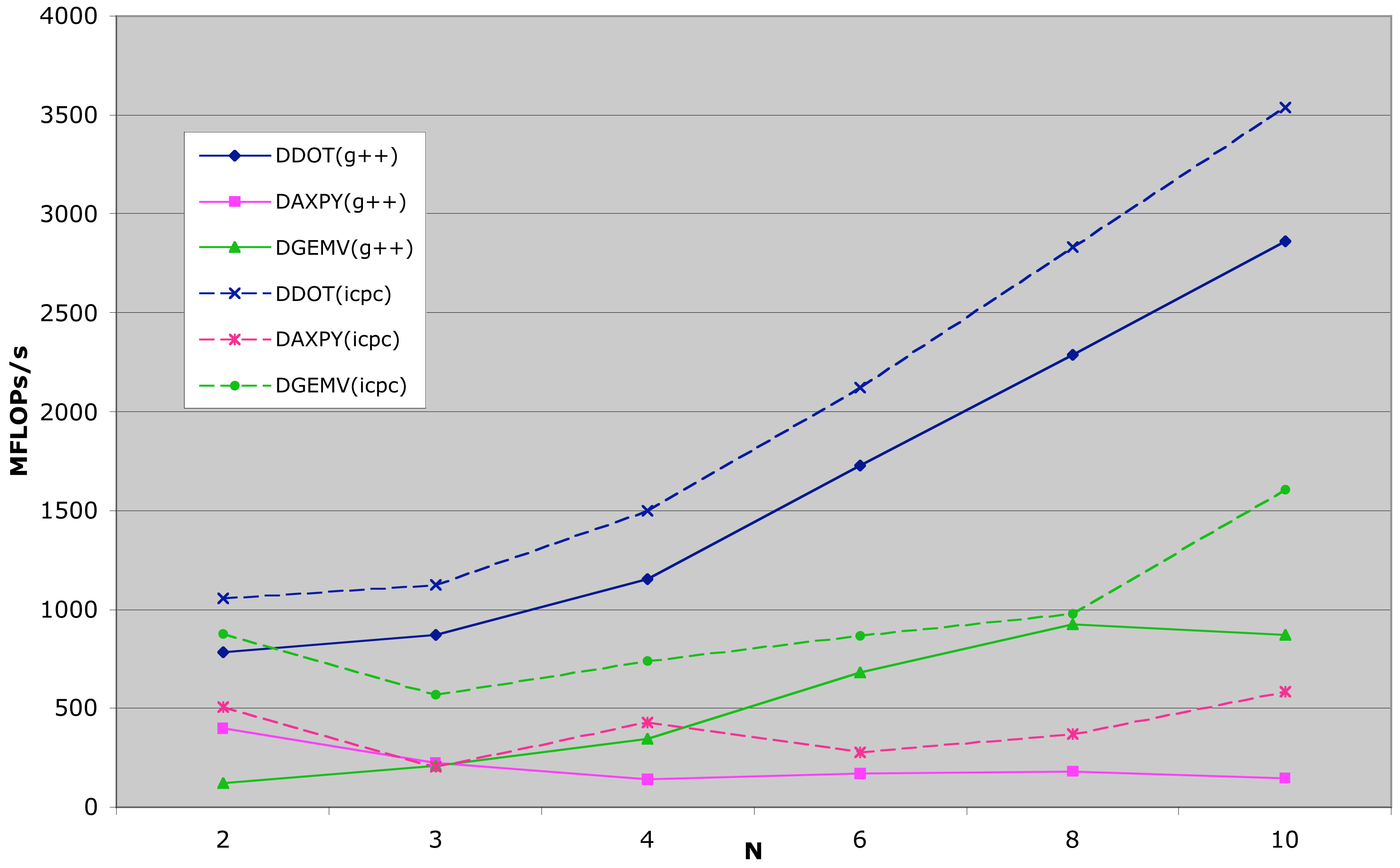
## TinyVector, TinyMatrix

- Fully Templated: e.g.

  **`TinyVector<double, 3> x, y, z;`**
  creates 3D-vectors of type double.
- Vector length known at compile time.
- Fully unroll all loops: **`z = x + y;`** becomes

```
z(1) = x(1) + y(1);
z(2) = x(2) + y(2);
z(3) = x(3) + y(3);
```

# TinyVector/Matrix Performance



## Conclusion

- BLAS are essential items in scientific computing
- Standardized interface to basic matrix and vector operations
- Highly optimized BLAS are available
- Many applications/packages/libraries depend dramatically on BLAS (e.g. dense and sparse solvers, both direct and iterative)

- **We recommend you use VENDOR BLAS!**