Closing the Performance Gap between Java and Fortran in Technical Computing

José E. Moreira

in collaboration with Manish Gupta, Sam Midkiff, Marc Snir, Peng Wu

IBM T. J. Watson Research Center
P.O. Box 218
Yorktown Heights NY 10598

presented at:
Java for High Performance Network Computing Workshop
EuroPar 1998 – September 2,3 1998

Why consider Java for technical computing?

- Java is increasingly used as a first programming language in colleges: large supply of programmers.
- Java supports OO programming well, without too many complications.
- Java ports easily (write once, run everywhere).
- Java will be pervasive in the environment surrounding and driving numeric-intensive computing: high-level control, GUIs, communication, ...

What about performance?

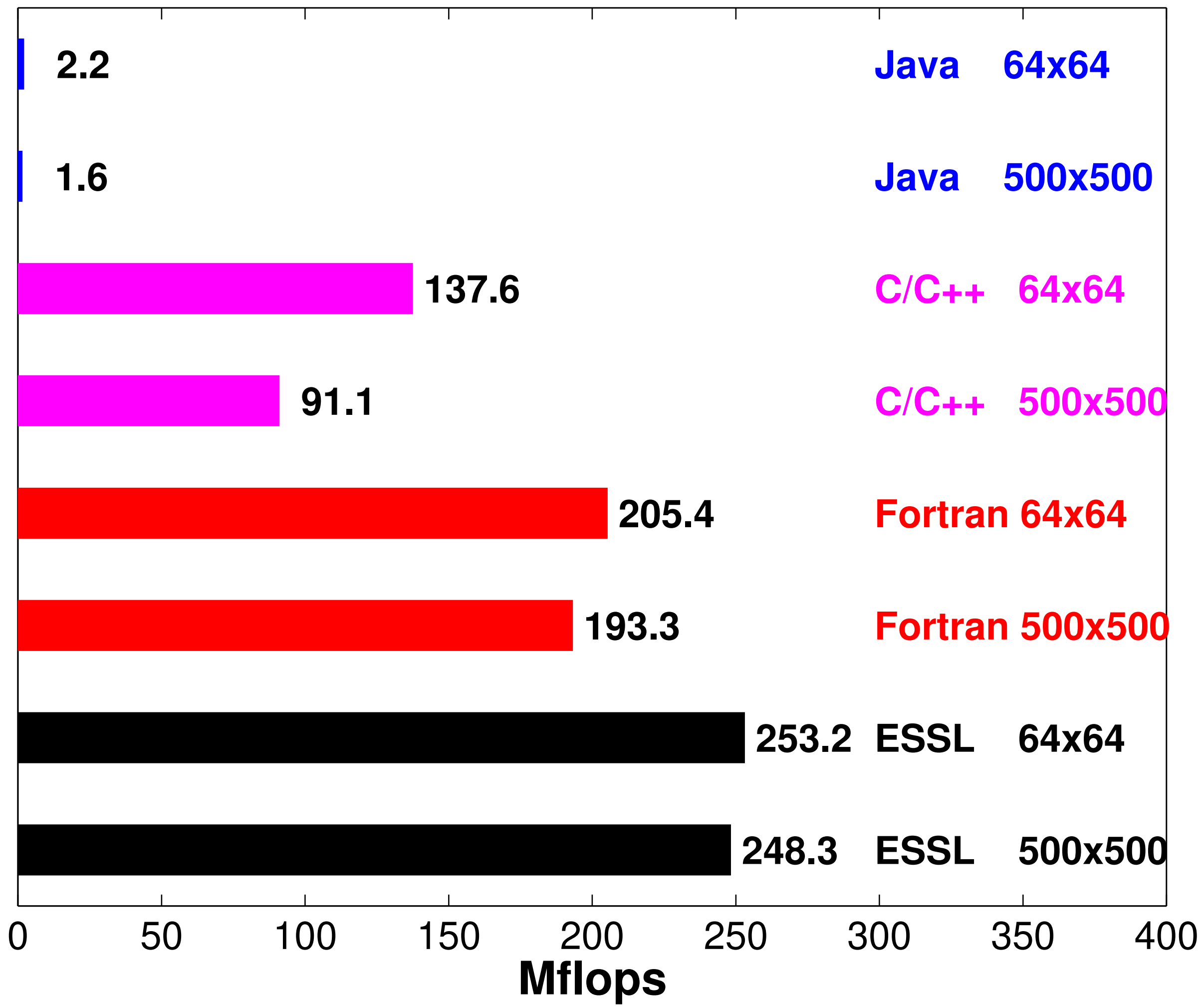
MATMUL benchmark description:

- static **void** matmul(**double**[][] A,**double**[][] B,**double**[][] C,
  int m,**int** n,**int** p) {
  int i,j,k;
  for $(i = 0; i < m; i{+}{+})$
  for $(j = 0; j < p; j{+}{+})$
  for $(k = 0; k < n; k{+}{+})$
  $C[i][j] {+}{=} A[i][k] * B[k][j]$
  }

- $m = n = p = 64$ (fits in cache of POWER2).

- $m = n = p = 500$ (too big for cache of POWER2).

MATMUL results on RS/6000 590



Why is Fortran so much faster?

- At 205 Mflops, Fortran is 50% faster that C, and ~ 100 times faster than Java!
- The IBM XLF 4.1 compiler uses several high-order transformations:
  - blocking, for better memory behavior,
  - multiple loop unrolling,
  - loop interchange,
  - scalar replacement,
  - loop fusion.
- The same transformations can be applied to safe regions of Java! Normally, exceptions prevent reordering.

Creating a safe region

- Avoid index exceptions.

$$
\begin{array}{l}
\textbf{if}\ (\ \ (m \le C_{\text{rows}}) \wedge (p \le C_{\text{cols}}) \wedge (m \le A_{\text{rows}}) \wedge (n \le A_{\text{cols}}) \wedge \\
\qquad (n \le B_{\text{rows}}) \wedge (p \le B_{\text{cols}})\ )\ \{ \\
\quad \textbf{for}\ (i = 0; i < m; i{+}{+}) \\
\qquad \textbf{for}\ (j = 0; j < p; j{+}{+}) \\
\qquad\quad \textbf{for}\ (k = 0; k < n; k{+}{+}) \\
\qquad\qquad C[i][j] \mathrel{+}= A[i][k] * B[k][j] \\
\}\ \textbf{else}\ \{ \\
\quad \textbf{for}\ (i = 0; i < m; i{+}{+}) \\
\qquad \textbf{for}\ (j = 0; j < p; j{+}{+}) \\
\qquad\quad \textbf{for}\ (k = 0; k < n; k{+}{+}) \\
\qquad\qquad C[i][j] \mathrel{+}= A[i][k] * B[k][j] \\
\}
\end{array}
$$

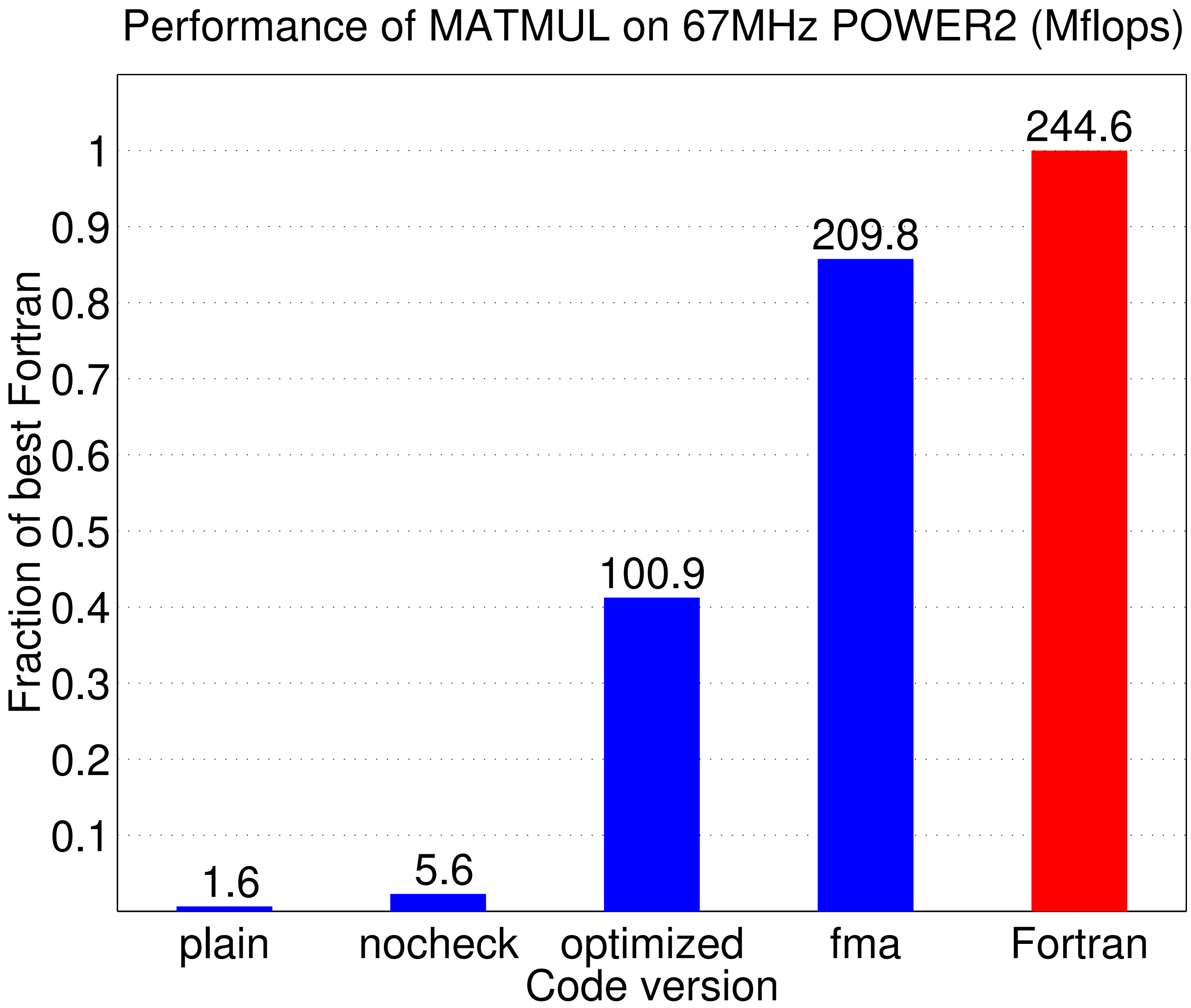
- The red version is safe! Bounds checking is not necessary and reordering is possible!

Optimizing the safe region

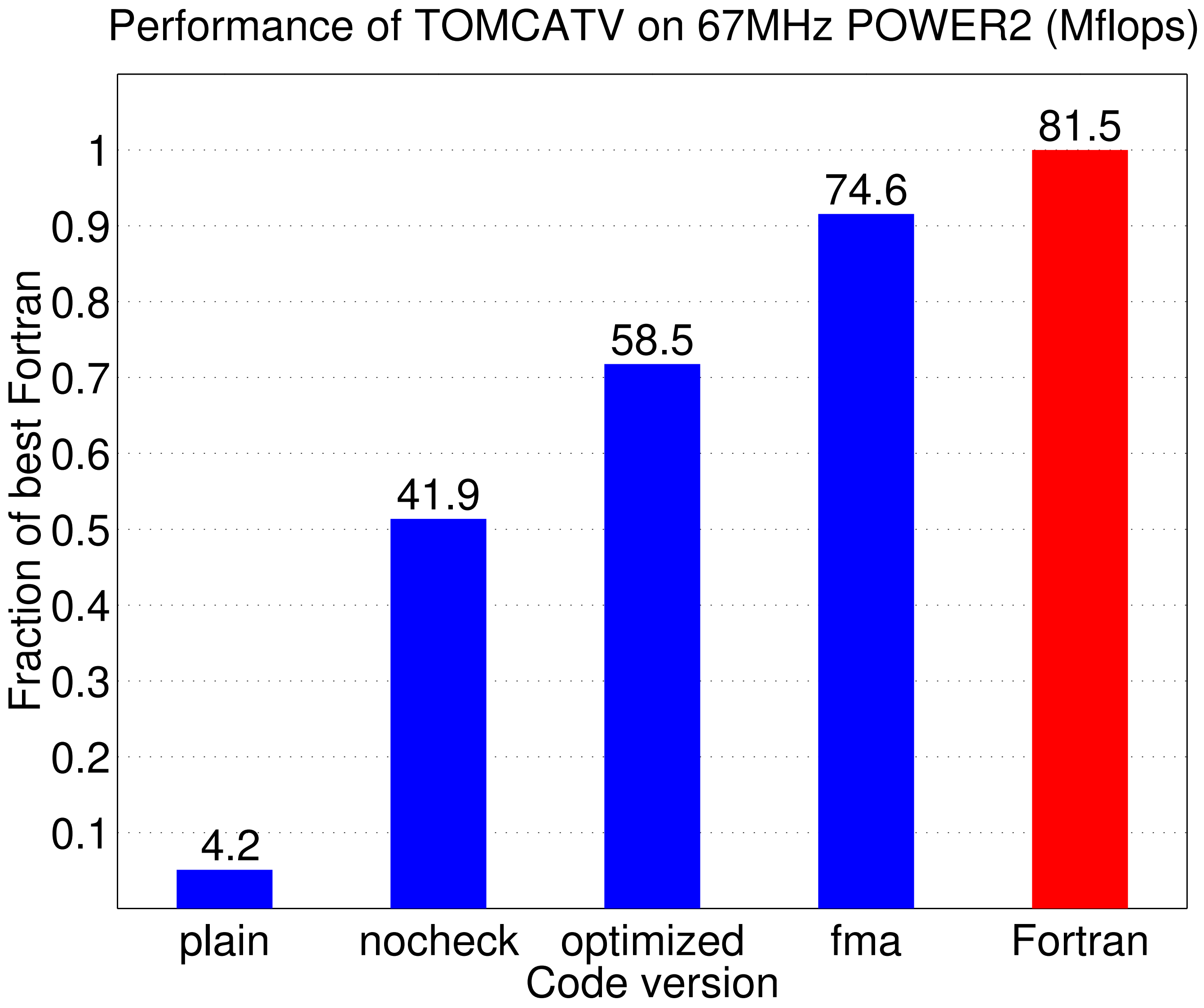
- blocking: for better cache reuse.
- unrolling: of i and j loops for better register reuse and to exploit functional-unit parallelism. Note that k loop cannot be parallelized in Java!
- scalar replacement: requires pointer disambiguation. Can be done at run-time, much easier with Matrix objects.
- fma: uses extended-precision fused-multiply-add instruction. Not legal yet.

In the next plot, library has all the optimizations of the fma version except for bounds checking. This is the best hand implementation with all checks explicit.

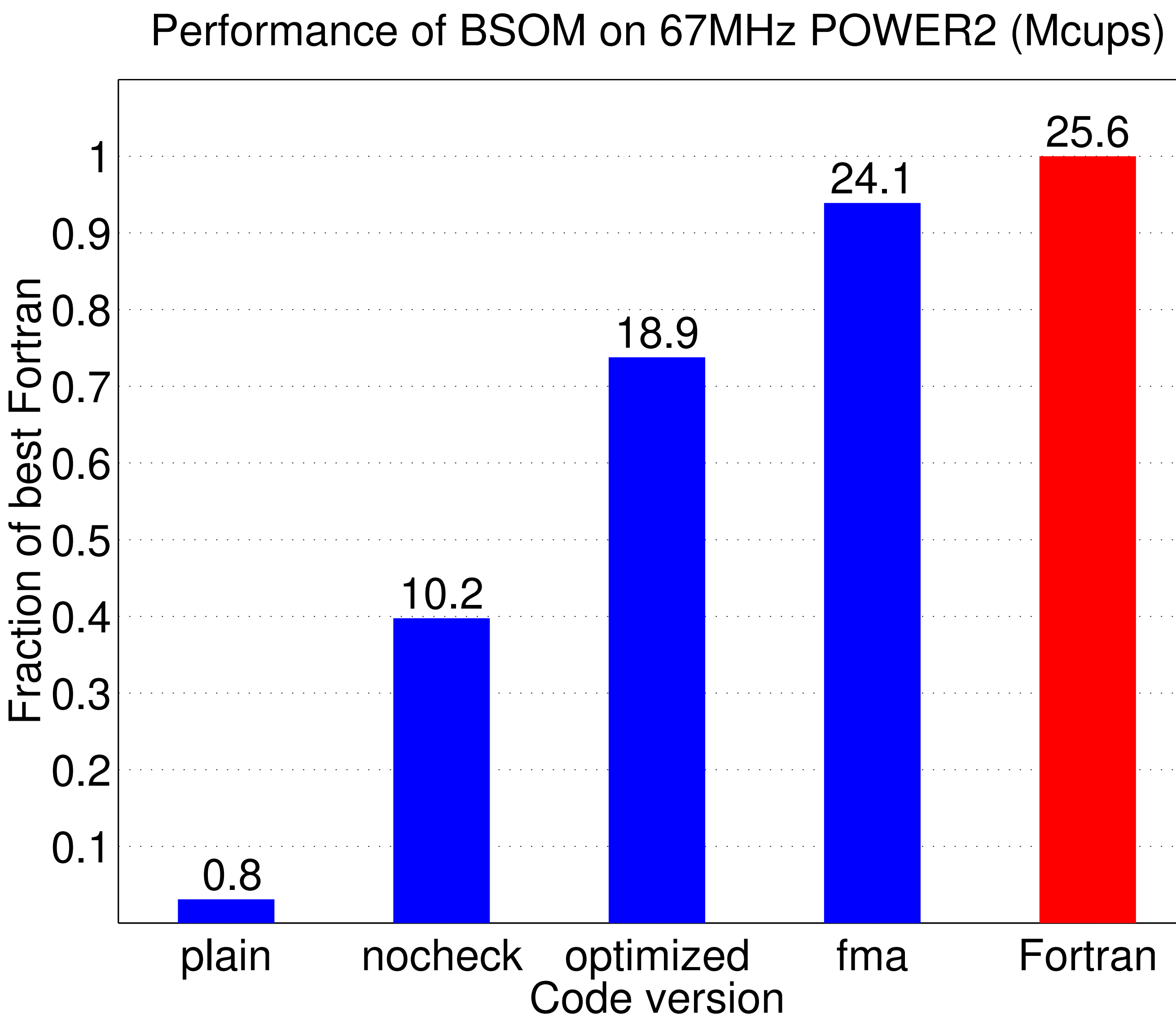
Performance of 500×500 MATMUL on RS/6000 590



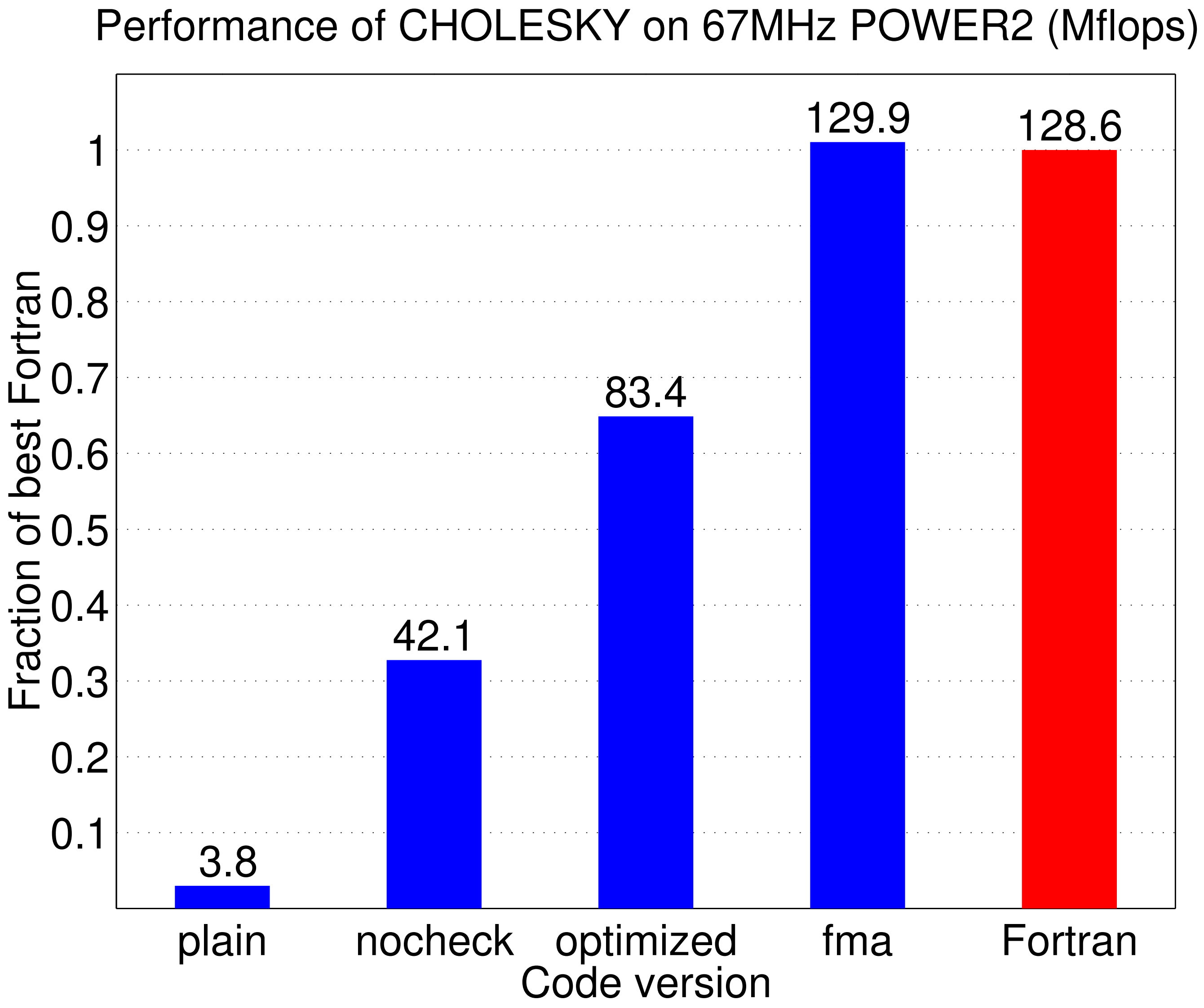
Performance of TOMCATV on RS/6000 590



Performance of BSOM on RS/6000 590



Performance of 1000×1000 CHOLESKY on RS/6000 590



Parallelizing MATMUL

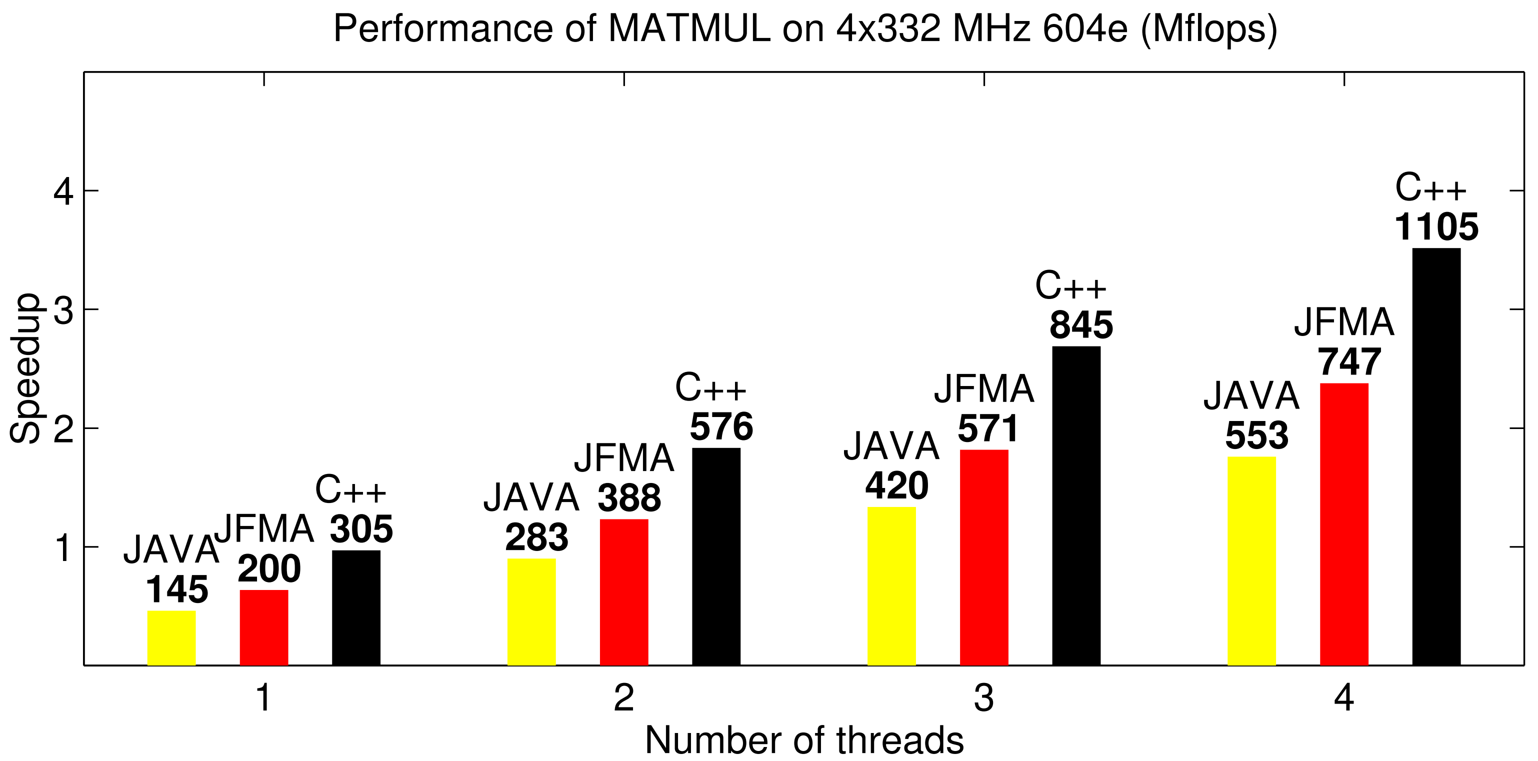
- The safe region can be multithreaded! Exploit parallelism on I and J block loops.

- b_I,b_J, and b_K are the block sizes along I, J, and K. t is the thread id, n_T is the number of threads.

$$\begin{array}{c|ccccc|}
 & 0 & & \xrightarrow{J} & & n_J-1 \\
\hline
0 & T_0 & T_1 & T_2 & T_3 & T_0 \\
 & T_1 & T_2 & T_3 & T_0 & T_1 \\
I\downarrow & T_2 & T_3 & T_0 & T_1 & T_2 \\
 & T_3 & T_0 & T_1 & T_2 & T_3 \\
n_I-1 & T_0 & T_1 & T_2 & T_3 & T_0 \\
\hline
\end{array}$$

for $(b = t; b < n_I n_J; b{+}{=}n_T)$ $\{$
  $I = (b/n_I) \times b_I$
  $J = (b \bmod n_I) \times b_J$
  for $(K = 0; K < n; K{+}{=}b_K)$ $\{$
    for $(i = I; i < \min(m, I + b_I); i{+}{+})$
      for $(j = J; j < \min(p, J + b_J); j{+}{+})$
        for $(k = K; k < \min(n, K + b_K); k{+}{+})$
          $C[i][j]{+}{=}A[i][k] * B[k][j]$
  $\}$
$\}$

Parallelizing MATMUL – results



Parallel ESSL: 1183 Mflops.

Matrix objects

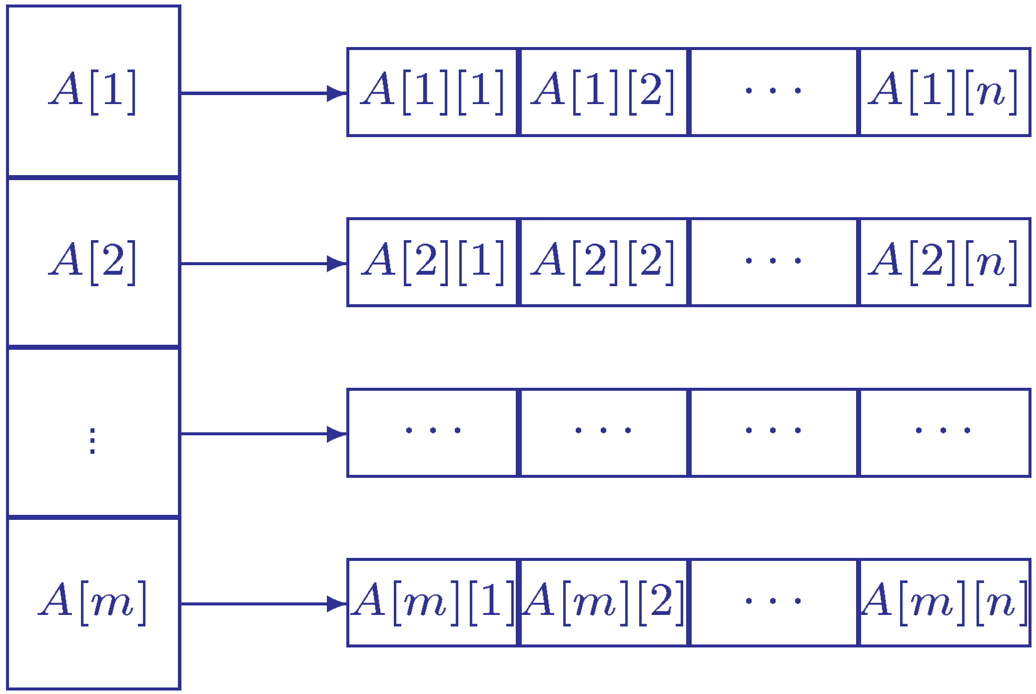
Matrix

$A(1:m,1:n)$

$A(1,1)$	$A(1,2)$	$\cdots$	$A(1,n)$
$A(2,1)$	$A(2,2)$	$\cdots$	$A(2,n)$
$\cdots$	$\cdots$	$\cdots$	$\cdots$
$A(m,1)$	$A(m,2)$	$\cdots$	$A(m,n)$

Array of arrays

$A[1:m][1:n]$



Matrices vs. arrays of arrays

1. The location of an element $A(i, j)$ can be computed using simple arithmetic. Finding $A[i][j]$ involves, in general, pointer chasing.

2. There is no guarantee that $A[1 : m][1 : n]$ is rectangular. Even if it was rectangular when created it may have been modified.

3. Aliasing disambiguation. Given two matrix objects A and B it is sufficient to show that $A \neq B$ to prove there is no aliasing between them. In the case of two arrays of arrays $A[][]$ and $B[][]$ one has to show that $A[i] \neq B[j]\ \forall i, j$.

4. Regular array sections for matrices can be represented compactly.

5. Disambiguation between two array sections reduces to proving that the source arrays are different or that the sections do not intersect.

6. Privatizing matrix $A(1 : m, 1 : n)$, for thread safety, requires copying only one reference.

Java array package

- matrix-multiply code:

```
doubleArray2D matmul(doubleArray2D a, doubleArray2D b) {
  int m = a.size(0), n = a.size(1), p = b.size(1);
  doubleArray2D c = new doubleArray2D(m,p);
  c.assign(0);
  for (int i=0; i<m; i++)
    for (int j=0; j<p; j++)
      for (int k=0; k<n; k++)
        c.set(i,j,c.get(i,j)+a.get(i,k)*b.get(k,j));
  return c;
}
```

- Java Grande forum activities important to standardize Java array package and to simplify notation through operator overloading.

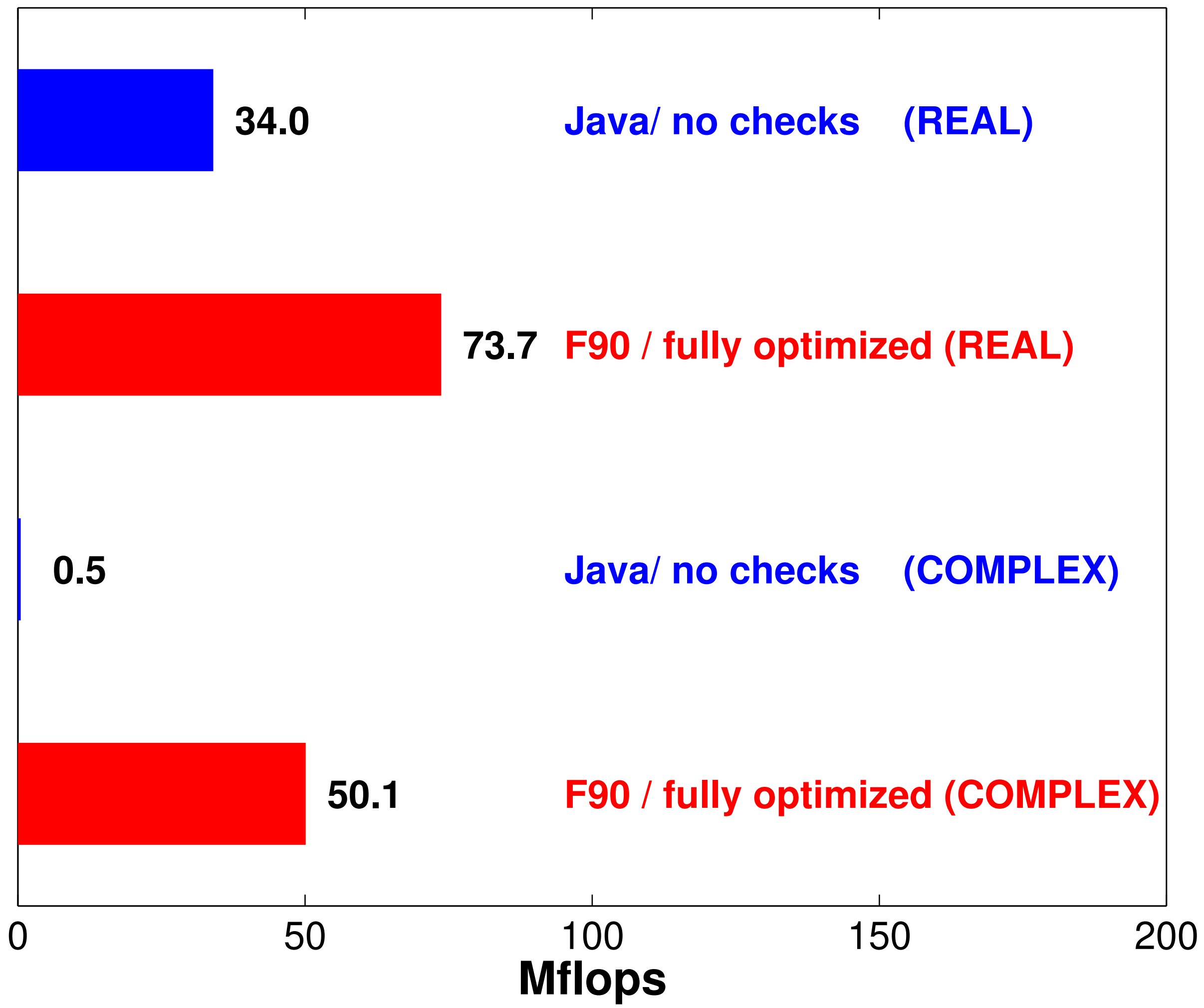
Difficulties with complex numbers

MICROSTRIP benchmark description:

- **do** $i = 1, w - 1$
  do $j = 1, h - 1$
  $B[i][j] = 0.25 * (A[i+1][j] + A[i-1][j] + A[i][j+1] + A[i][j-1])$
  end do
  end do
  do $i = 0, w$
  do $j = 0, h$
  $r = r + |B[i][j] - A[i][j]|$
  end do
  end do

- $w = h = 999.$

MICROSTRIP results on RS/6000 590



Semantic inlining

- Standard array and complex packages and semantic inlining key to optimizing things like MICROSTRIP.

- Through semantic inlining, compiler replaces calls to methods in standard classes by code that direct implements semantics of those methods.

- We want the compiler to understand constructs like

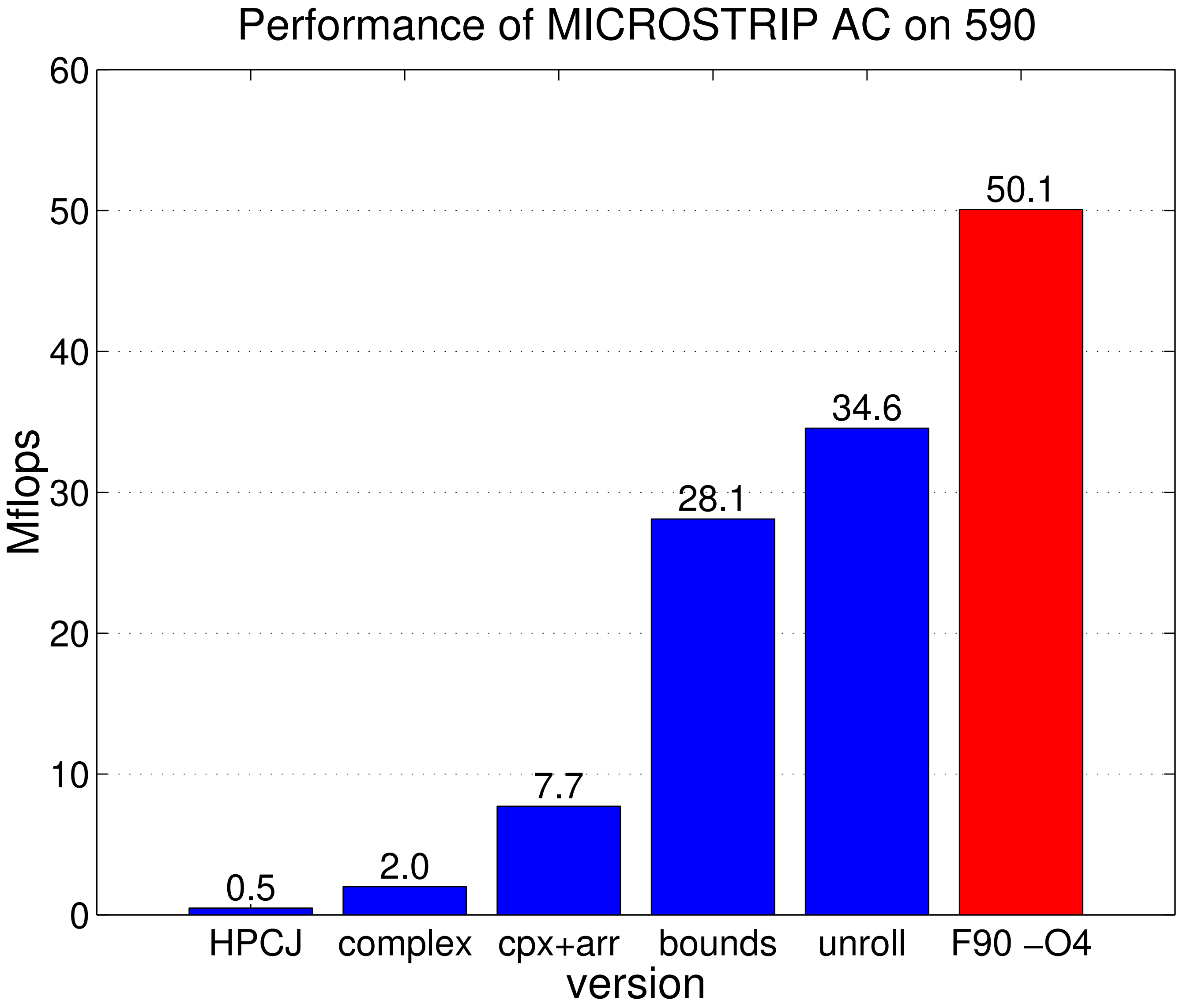
```
DoubleComplex[] a = new DoubleComplex[n];
DoubleComplex[] b = new DoubleComplex[n];
DoubleComplex c = new DoubleComplex(0,0);

for (int i=0; i<n; i++)
   c = DoubleComplex.add(c,DoubleComplex.mul(a[i],b[i]));
```

  as operations on complex numbers.

- Strong synergy with multidimensional arrays.

Results of semantic inlining



Conclusions

- There are good reasons to use Java for technical computing.
- Performance problems can be handled:
  - run-time checks with compiler optimizations;
  - matrices and multidimensional arrays by standard classes, compiler technology;
  - `fma` and reordering by relaxing semantics;
  - safe regions enable high-order transformations and parallelization;
  - complex arithmetic by standard classes, compiler technology.
- Java performance can be 80-100% of Fortran.
- Java Grande Forum: standardization effort important to enable compiler optimizations.
- Next step: development of large Java technical applications.

Details of 500×500 MATMUL