PROGRAMMING AT A HIGH-LEVEL ON MULTI-CORES

What is a compiler to do?

Arun Chauhan
Indiana University

The Multi-core crisis

- Physical limitations
  - ⋆ Transistor sizes
  - ⋆ Clock skew
- Power consumption
- "Moore's Law"

Software Productivity: The Real Crisis

- New software development
  - ⋆ Programming models
  - ⋆ Programming techniques
  - ⋆ Programming languages
- Porting legacy code
  - ⋆ Starting point: sequential or parallel?
  - ⋆ Port optimized code
  - ⋆ Source vs binary

Possible Solutions

- Novel languages
  - ⋆ DARPA HPCS
- Extending traditional languages
  - ⋆ Co-Array Fortran
  - ⋆ UPC
- Libraries
  - ⋆ ScalaPACK, MATAB*P
- High-level "scripting" languages

High-Level Scripting Languages

- Available and in use
- Modern
  - ★ Support modern software engineering practices
- More powerful and general than libraries
- Programmers available

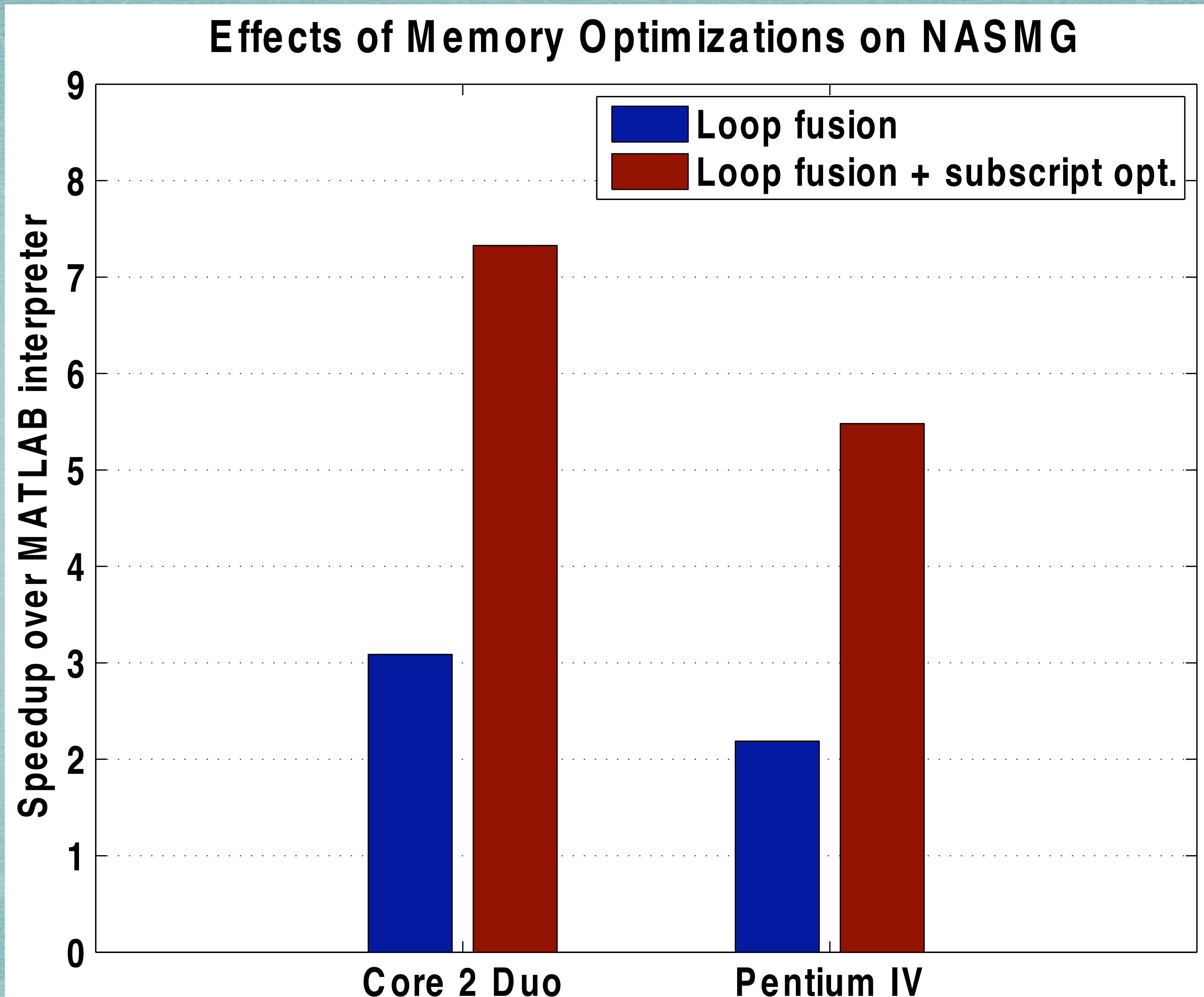
High-Level Scripting Languages

- Available and in use
- Modern
  - Support modern software engineering practices
- More powerful and general than libraries
- Programmers available

Can they solve the multi-core programming crisis?

Example: NASMG in MATLAB

```
m = f(1).*(n(c,c,c)) + ...
   f(2).*(n(c,c,u)+n(c,c,d)+n(c,u,c)+n(c,d,c)+n(u,c,c)+n(d,c,c)) + ...
   f(3).*(n(c,u,u)+n(c,u,d)+n(c,d,u)+n(c,d,d)+n(u,c,u)+n(u,c,d)+ ...
          n(d,c,u)+n(d,c,d)+n(u,u,c)+n(u,d,c)+n(d,u,c)+n(d,d,c)) + ...
   f(4).*(n(u,u,u)+n(u,u,d)+n(u,d,u)+n(u,d,d)+n(d,u,u)+n(d,u,d)+ ...
           n(d,d,u)+n(d,d,d));
```



Effects of Memory Optimizations on NASMG
Speedup over MATLAB interpreter
0
1
2
3
4
5
6
7
8
9
Loop fusion
Loop fusion + subscript opt.
Core 2 Duo
Pentium IV

Why Compilation is Unnecessary

- Most of the computation takes place in libraries
- Interpretive overheads insignificant with byte-code
- Just-in-time compilation does a good job

Why Compilation is Unnecessary

- Most of the computation takes place in libraries
  - ★ True for some applications, but not for many others
  - ★ Parallelization on heterogeneous platforms
- Interpretive overheads insignificant with byte-code
- Just-in-time compilation does a good job

Why Compilation is Unnecessary

- Most of the computation takes place in libraries
  - ★ True for some applications, but not for many others
  - ★ Parallelization on heterogeneous platforms
- Interpretive overheads insignificant with byte-code
  - ★ Byte-code does not eliminate library call overheads
- Just-in-time compilation does a good job

Why Compilation is Unnecessary

- Most of the computation takes place in libraries
  - ⋆ True for some applications, but not for many others
  - ⋆ Parallelization on heterogeneous platforms
- Interpretive overheads insignificant with byte-code
  - ⋆ Byte-code does not eliminate library call overheads
- Just-in-time compilation does a good job
  - ⋆ JIT compiler operates at byte-code level, missing many opportunities at high-level

Why Compilation is Necessary

Why Compilation is Necessary

Interprocedural Optimization

Benefits of Source-level Compilation

- Specialization
  - ⋆ Type-based specialization can reduce or eliminate function call overheads
- Library function selection
  - ⋆ Sequences of operations can be implemented efficiently
- Memory footprint reduction
  - ⋆ Intermediate arrays and array computations can be eliminated
- Parallelization
  - ⋆ Macro-operations provide naturally coarse granularity

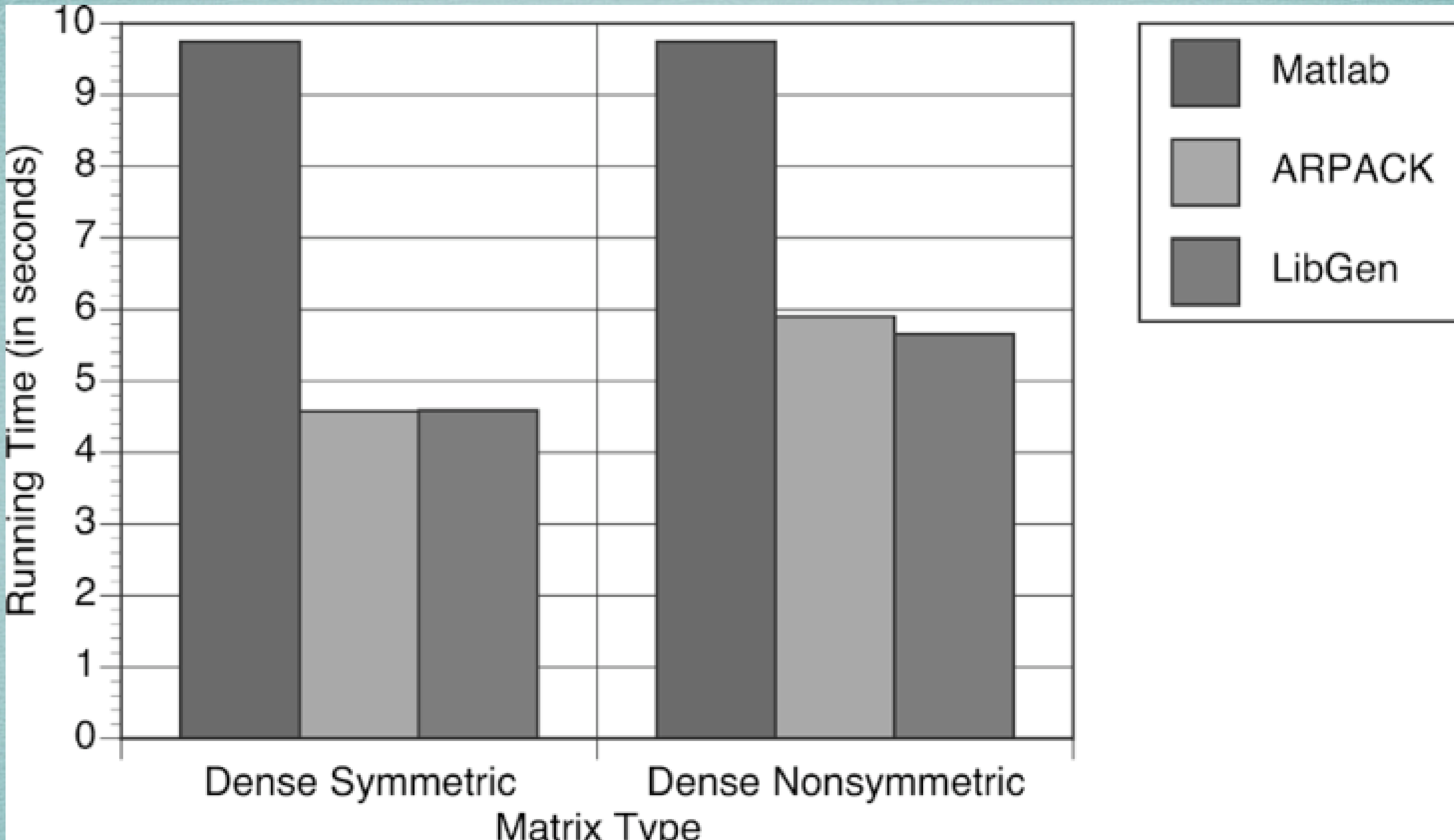
Benefits of Source-level Compilation

- Specialization
  - ⋆ Type-based specialization can reduce or eliminate function call overheads
- Library function selection
  - ⋆ Sequences of operations can be implemented efficiently
- Memory footprint reduction
  - ⋆ Intermediate arrays and array computations can be eliminated
- Parallelization
  - ⋆ Macro-operations provide naturally coarse granularity

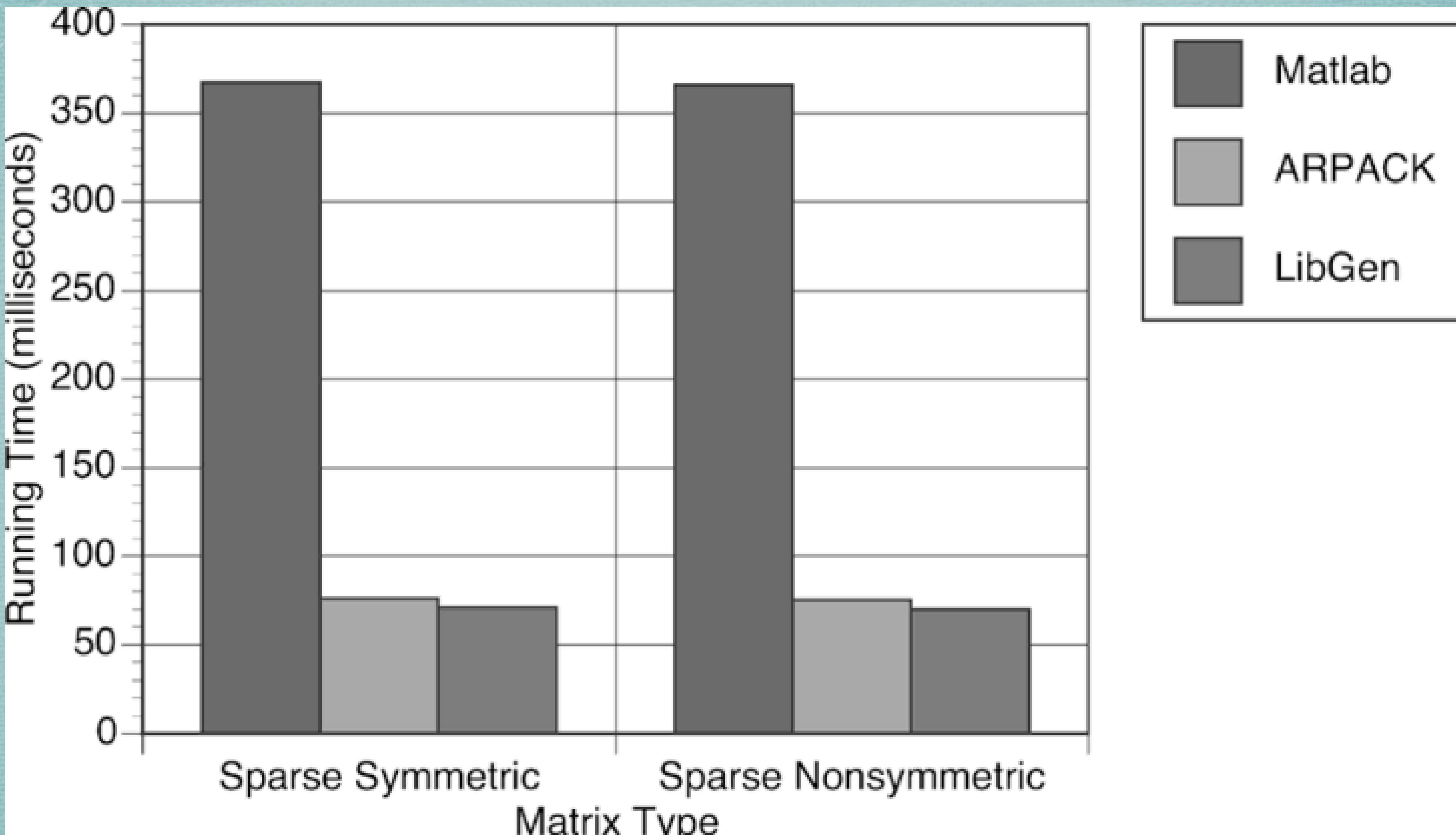
Benefits of Source-level Compilation

- Specialization
  - ★ Type-based specialization can reduce or eliminate function call overheads
- Library function selection
  - ★ Sequences of operations can be implemented efficiently
- Memory footprint reduction
  - ★ Intermediate arrays and array computations can be eliminated
- Parallelization
  - ★ Macro-operations provide naturally coarse granularity

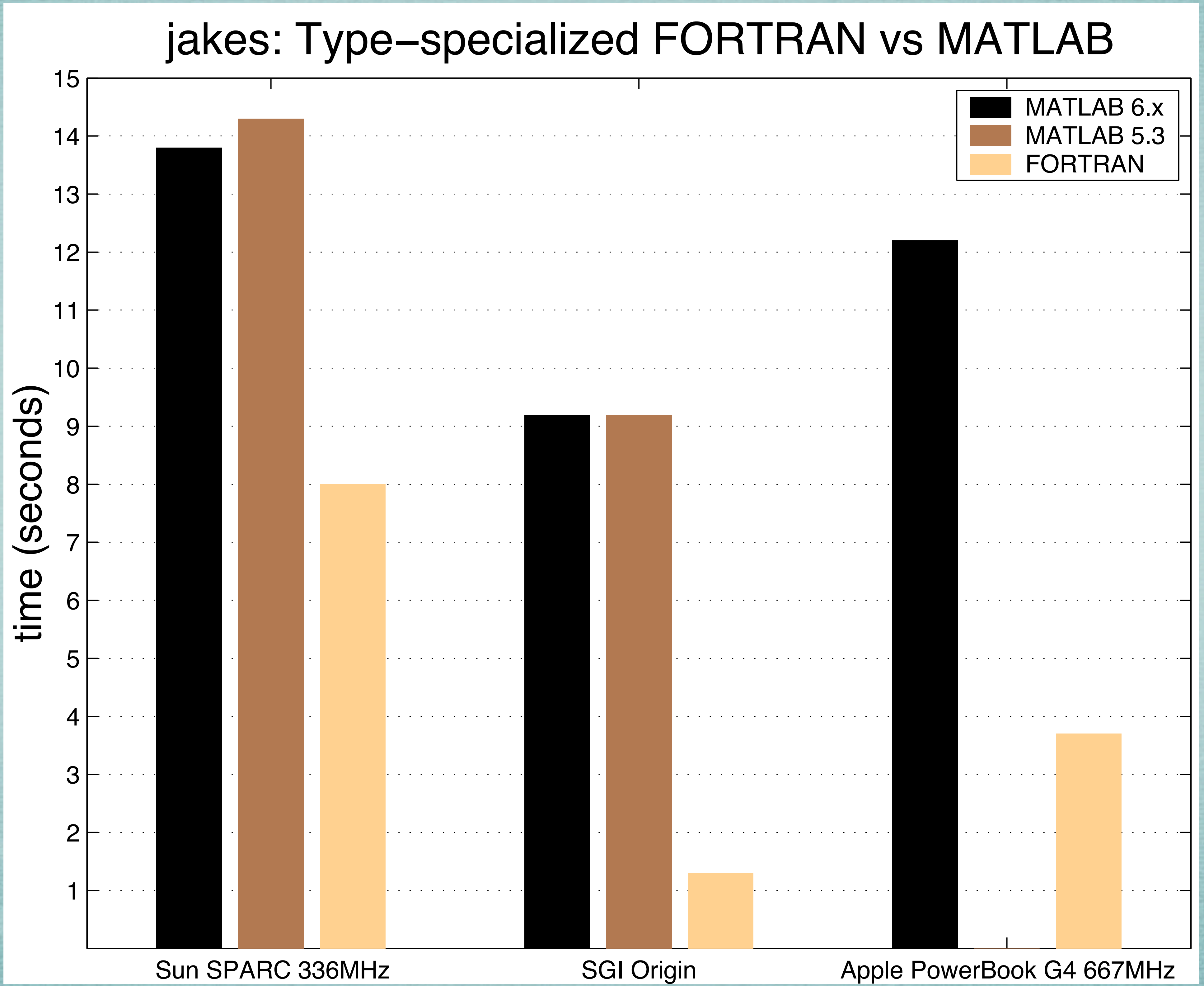
Specialization: ARPACK Dense Matrix Kernel



Specialization: ARPACK Sparse Matrix Kernel



Type-based Specialization: DSP



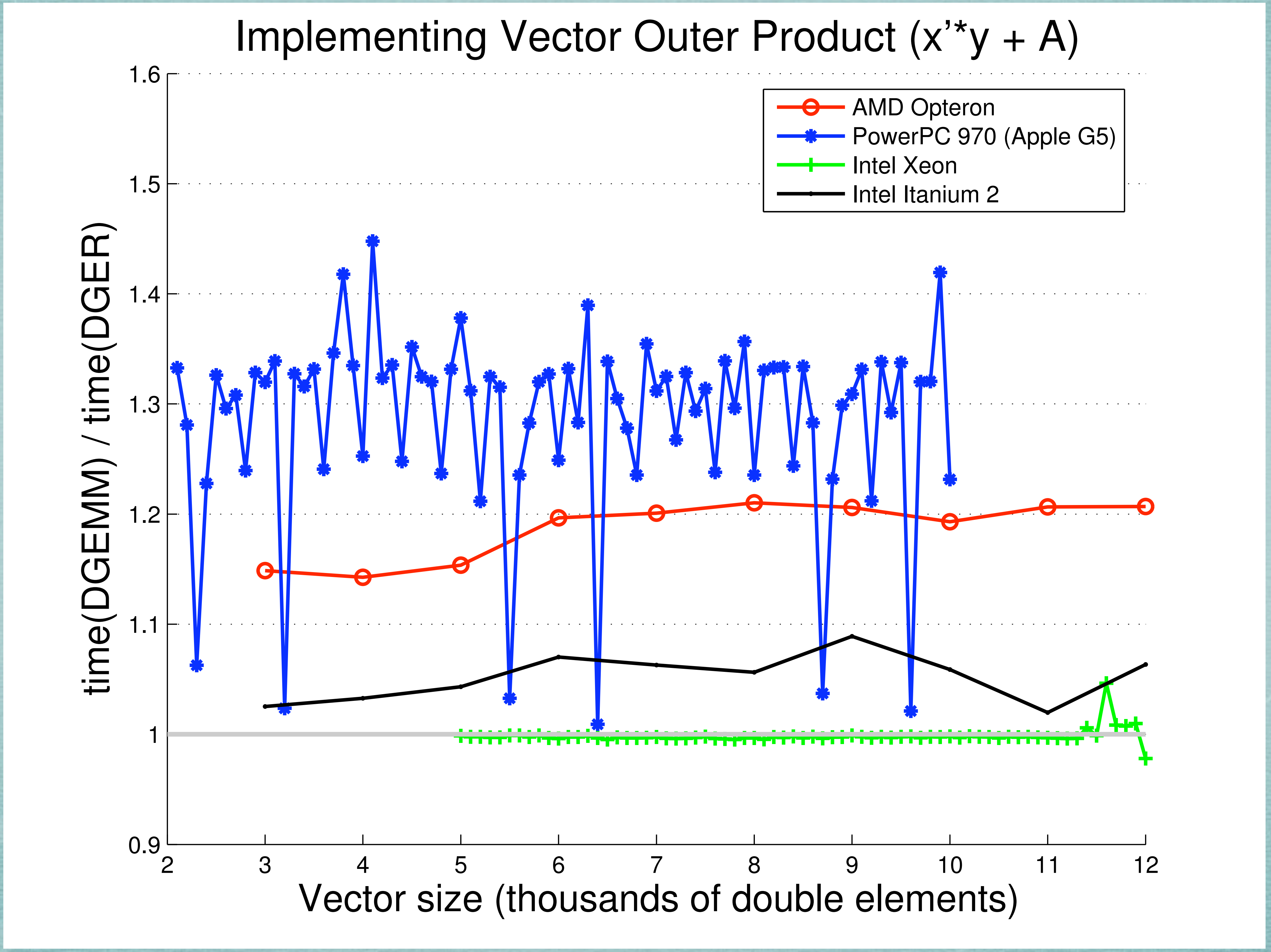
Benefits of Source-level Compilation

- Specialization
  - ⋆ Type-based specialization can reduce or eliminate function call overheads
- Library function selection
  - ⋆ Sequences of operations can be implemented efficiently
- Memory footprint reduction
  - ⋆ Intermediate arrays and array computations can be eliminated
- Parallelization
  - ⋆ Macro-operations provide naturally coarse granularity

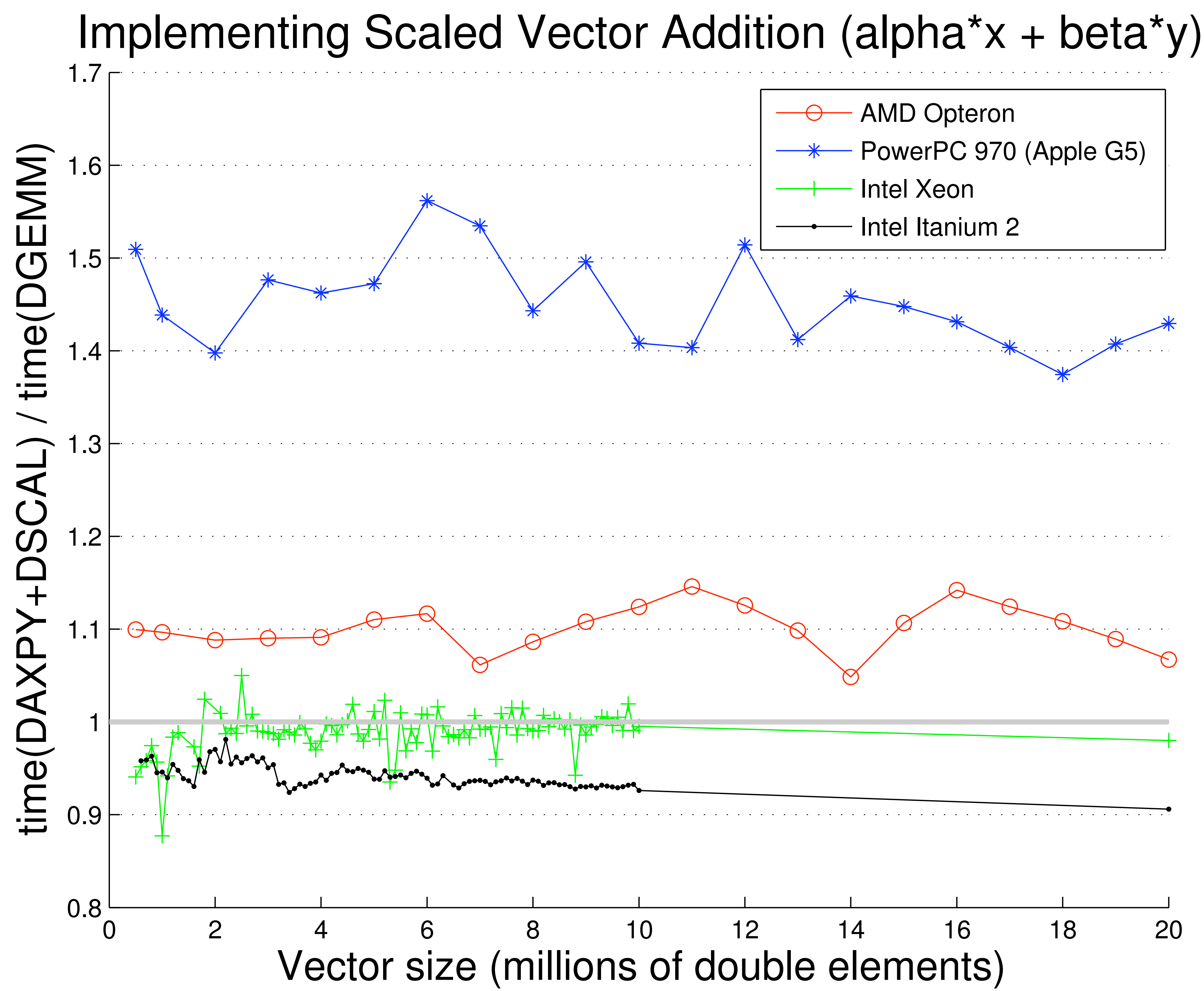
Benefits of Source-level Compilation

- Specialization
  - ⋆ Type-based specialization can reduce or eliminate function call overheads
- Library function selection
  - ⋆ Sequences of operations can be implemented efficiently
- Memory footprint reduction
  - ⋆ Intermediate arrays and array computations can be eliminated
- Parallelization
  - ⋆ Macro-operations provide naturally coarse granularity

Library Function Selection: Vector Outer-product



Library Function Selection: Scaled Vector Add



Benefits of Source-level Compilation

- Specialization
  - ⋆ Type-based specialization can reduce or eliminate function call overheads
- Library function selection
  - ⋆ Sequences of operations can be implemented efficiently
- Memory footprint reduction
  - ⋆ Intermediate arrays and array computations can be eliminated
- Parallelization
  - ⋆ Macro-operations provide naturally coarse granularity

Benefits of Source-level Compilation

- Specialization
  - ⋆ Type-based specialization can reduce or eliminate function call overheads
- Library function selection
  - ⋆ Sequences of operations can be implemented efficiently
- Memory footprint reduction
  - ⋆ Intermediate arrays and array computations can be eliminated
- Parallelization
  - ⋆ Macro-operations provide naturally coarse granularity

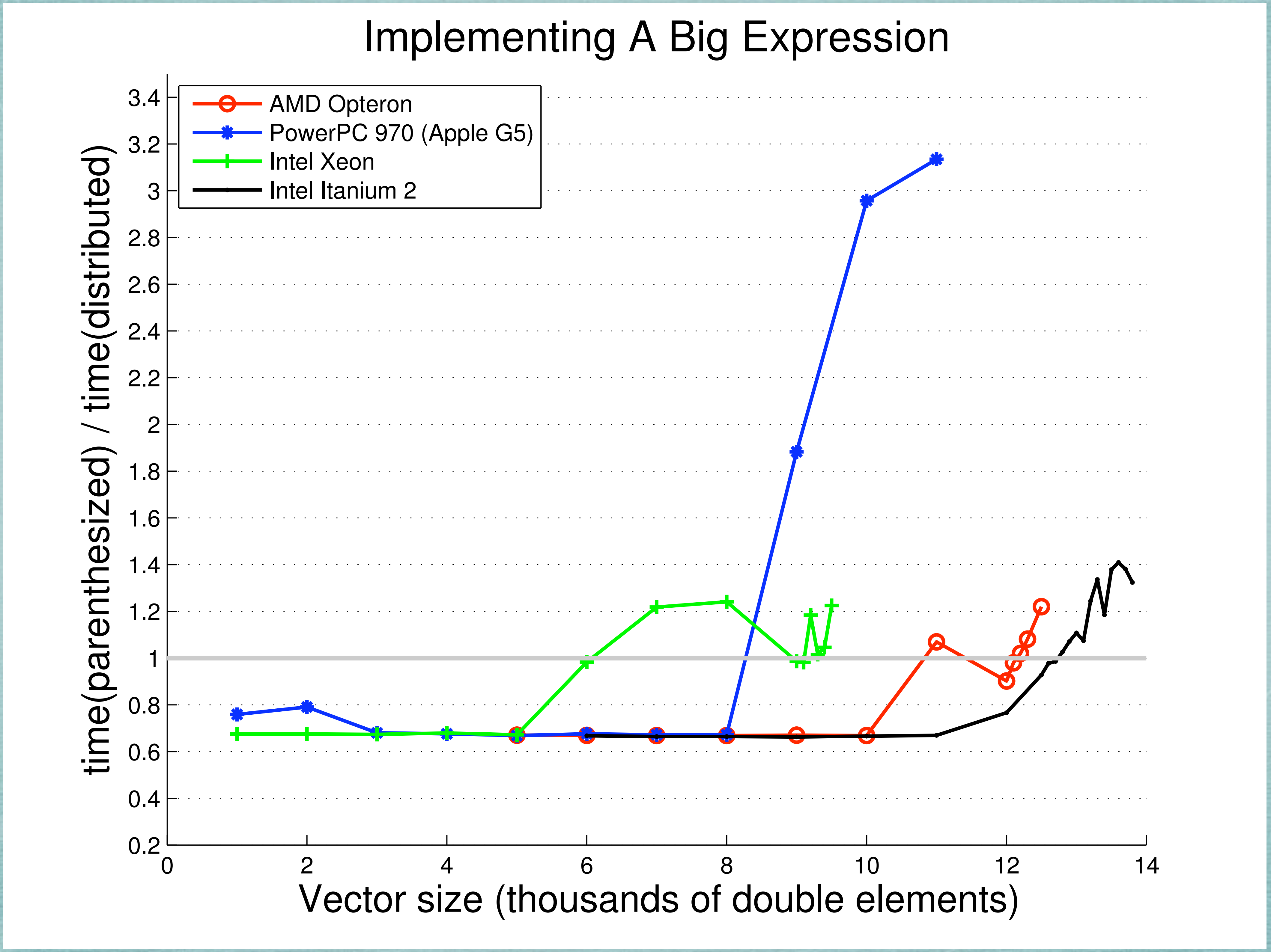
Temporary Arrays: Matrix Expressions

$$A+A*B' + 2*(A+B)'*A + (x+y)*x'$$

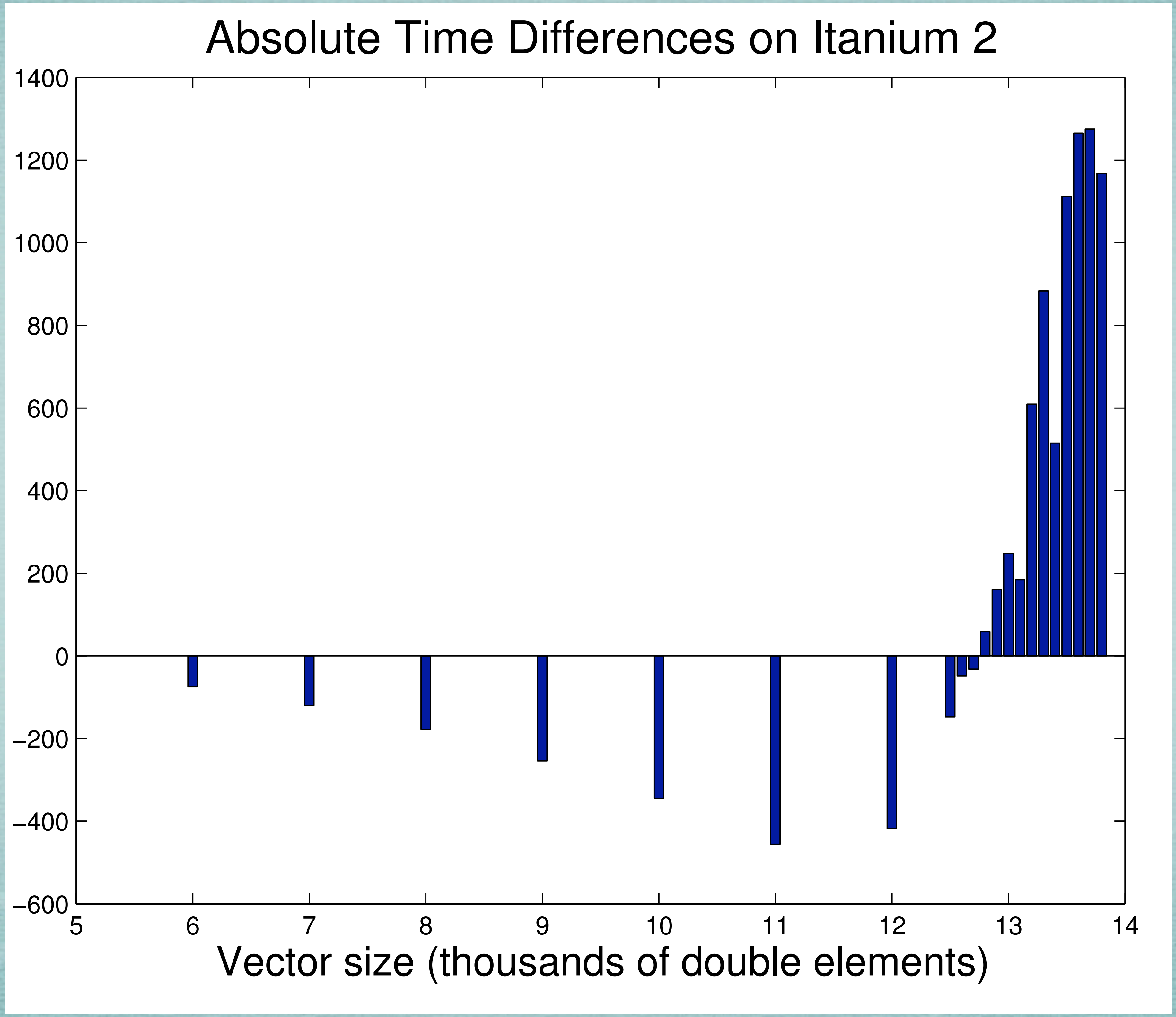
OR

$$A + A*B' + 2*A'*A + 2*B'*A + x*x' + y*x'$$

Parenthesized vs Distributed



Absolute Time Difference



Temporary Arrays: Matrix Expressions

$$A + A * B' + 2 * (A+B)' * A + (x+y) * x'$$

OR

$$A + A * B' + 2 * A' * A + 2 * B' * A + x * x' + y * x'$$

Temporary Arrays: Matrix Expressions

$A{+}A{*}B' + 2{*}(A{+}B)'{*}A + (x{+}y){*}x'$

```
copy(A,tmp0);
gemm(1,A,B,1,tmp0);
copy(A,tmp1);
axpy(1,B,1,tmp1);
gemm(2,tmp1,A,1,tmp0);
copy(x,tmp1);
axpy(1,y,1,tmp1);
ger(1,tmp1,x,tmp0);
```

$A + A{*}B' + 2{*}A'{*}A + 2{*}B'{*}A + x{*}x' + y{*}x'$

```
copy(A,tmp0);
gemm(1,A,B,1,tmp0);
copy(A,tmp1);
axpy(1,B,1,tmp1);
gemm(2,tmp1,A,1,tmp0);
```

Function Selection Algorithm

```
algorithm basic-block-function-selector
inputs: P = Octave source code (as AST)
        S = SSA graph of P
        L = target library
outputs: R = Modified version of P with operations
             mapped to the function calls in L
             whenever possible
________________
set R to an empty AST
for each simple statement, s, in P, do
  if (s does not have an operation implemented by L)
    add s unchanged to R
  else
    let ⊗ be the operation in s
    let ω be the optimal choice of function in L
      implementing ⊗ based on the current context
    if (ω is a multi-op function and
        ⊗ is a candidate operation)
      for each operand u
        let d be the statement defining u, obtained from S
        if (d can be subsumed in ω)
          add the operands of d to ω
        endif
      endfor
    endif
    add the call to ω to R
  endif
endfor
```

MEMORY BEHAVIOR MODELING

Motivation

The traditional theoretical approach to analysis involves counting basic operations performed on an abstract computer. These operations are generic abstractions of CPU instructions, each assumed to have some unit cost. However, on modern architectures, instructions have varying costs; in particular, a memory access can be several orders of magnitude more time consuming than any single machine instruction, depending on where the operands are located. New abstract models are needed to describe these new types of costs.

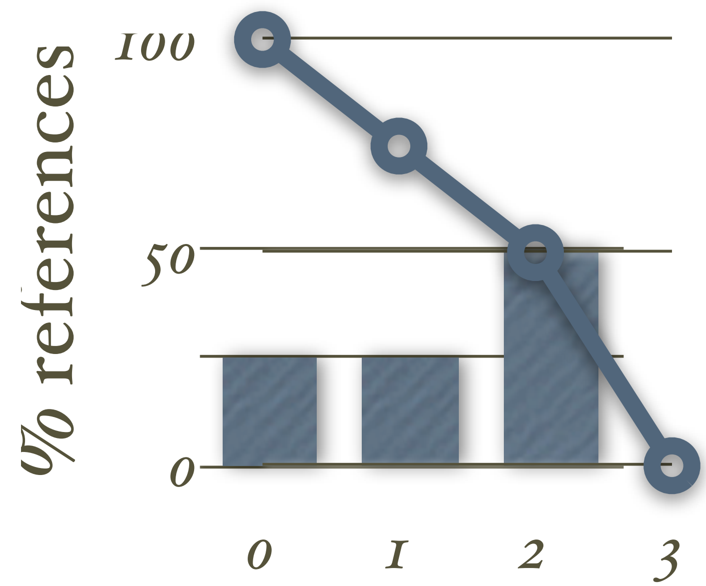
Catherine C. McGeoch,
Experimental Algorithmics,
Communications of the ACM **Volume 50, Number 11 (2007), Pages 27-31**

Reuse Distance *(courtesy: Chen Ding @ Rochester)*

Reuse Distance

- Reuse distance of an access to data *d*
  - the volume of data between this and the previous access to *d*
- Reuse signature of an execution
  - the distribution of all reuse distances
  - gives the miss rate of fully associative cache of all sizes

∞	∞	∞	2	0	1	2
a	b	c	a	a	c	b



Mattson, Gecsei, Slutz, Traiger
IBM Systems Journal, vol. 9(2), 1970, pp. 78-117

Source-level Reuse Distance

The source-level reuse distance between two memory references (to the same location) is the *volume of the source-level data* accessed between the two memory references.

```
x = a + b;
c = a + d[i]*100;
y = x;
```

Distance = 6 = Five variables (`a`, `b`, `c`, `d`, `i`) + one constant (`100`)

Complex Expressions

```
C = x + foo(i,j)*B[i+j,10];
D = bar(i,j) + x + y;
```

⬇

```
t_1 = i + j;
t_2 = foo(i,j);
t_3 = t_2 * B[t_1, 10];
C   = x + t_3;
t_4 = bar(i, j);
t_5 = t_4 + x;
D   = t_5 + y;
```

Reuse from Dependence Information

- Dependence for code transformations
  - Between two references to the same memory location
  - One of the references is a write
- Dependence for reuse: drop the write requirement
  - True dependence (δ)
  - Anti-dependence (δ^{-1})
  - Output dependence (δ^{o})
  - Input dependence (δ^{i})

Reuse within a Simple Statement

$dist = S(s\_func, \ldots)$

```
[x, u, x] = s_func(u, y, w, w)
```

$dist = 0$ $\qquad$ $dist = 0$

Computing Reuse within a Basic Block

Algorithm **Compute_R_simple**

Input: basic block B,
dependence graph D restricted to B

Output: $R_{p_1.p_2}$ $\forall$ pairs of ref. points p_1 and p_2 in B that are successive accesses of the same memory location

begin

let N_v be the unique number associated with the vertex v in D induced naturally by the total ordering of ref. points in B

let the length of an edge, $e = v_1 \to v_2$ be defined as $N_{v_2} - N_{v_1}$

foreach edge $e = v_1 \to v_2$ in D sorted by length

  if (either v_1 or v_2 is unexamined)

    mark v_1 and v_2 as "examined"

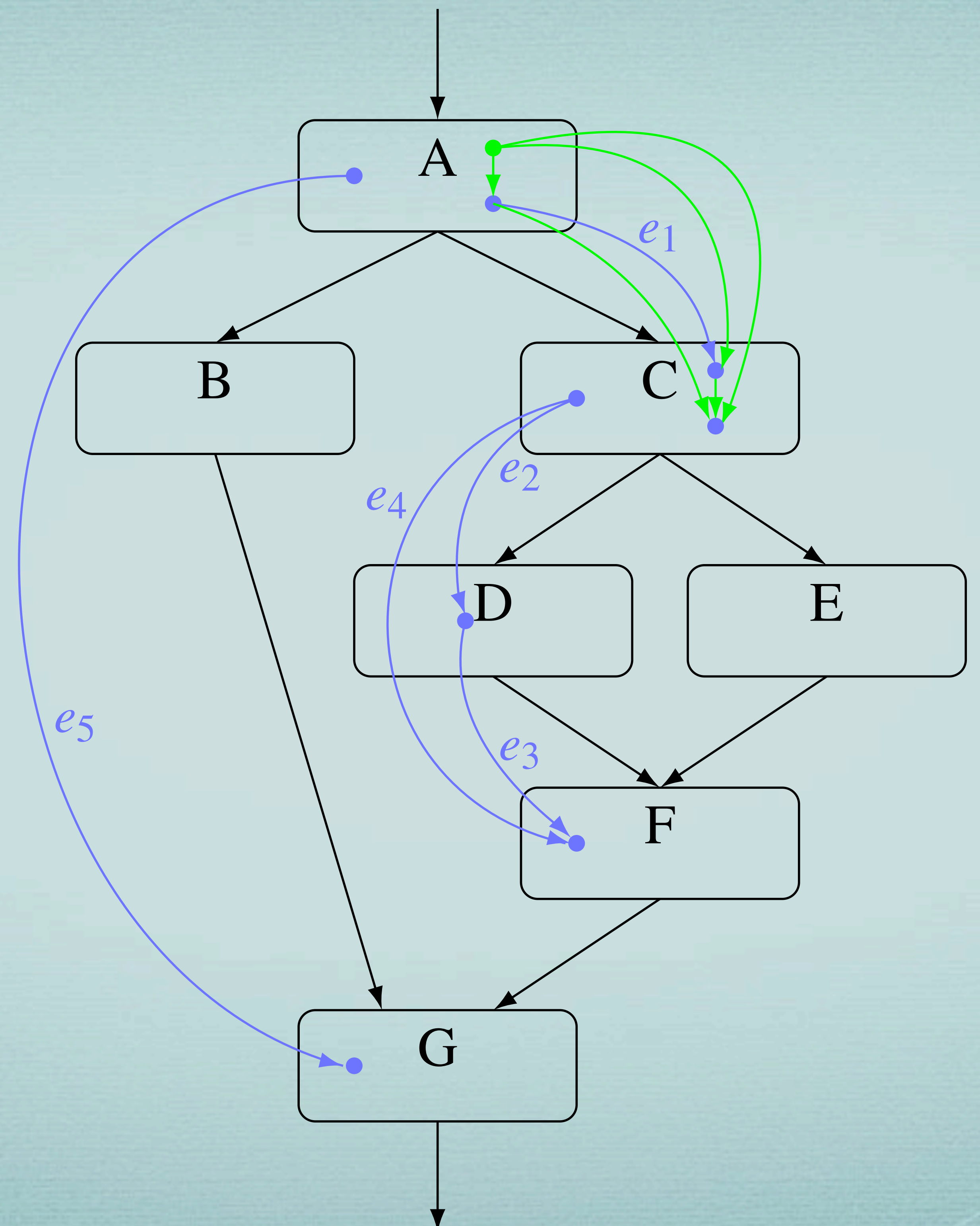
    $R_{v_1,v_2} = N_{v_2} - N_{v_1} -$ number of edges lying wholly between v_1 and v_2

  end

end

end

Reuse with Forward Control Flow



Computing Reuse with Forward Control Flow

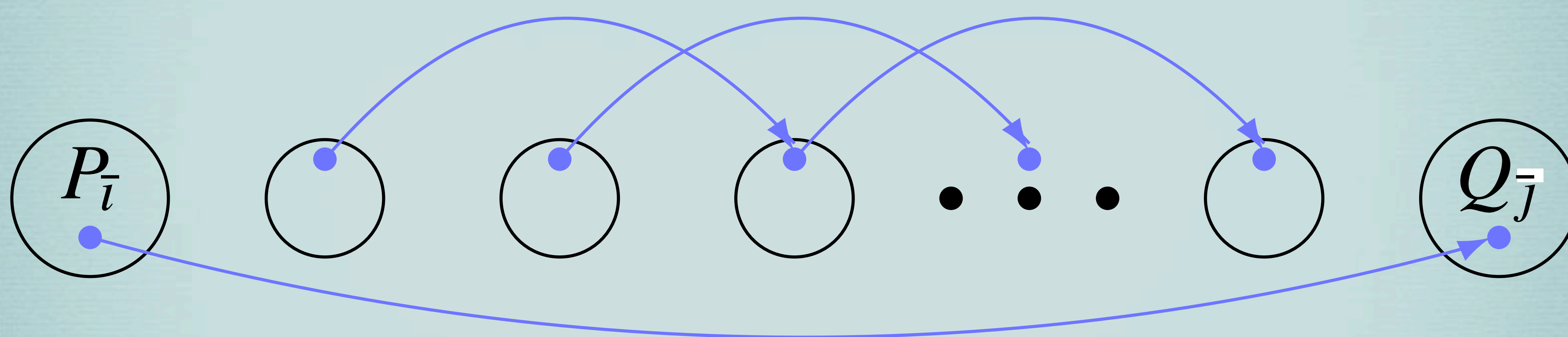
Algorithm **Compute_R_With_Branches**

Input: control flow graph G with weighted edges, dependence graph D
Output: $R_{p_1.p_2}$ $\forall$ pairs of ref. points p_1 and p_2 in B that are successive accesses of the same memory location

begin
1 restrict D to cross-block edges such that only the first incoming and the last outgoing edge is retained for each sequence of dependence edges
2 **foreach** block, b, in G
3 $(U[b], D[b]) = Exposed\_Data\_Volumes(b, D)$
end
4 **foreach** connected component, C, of D
5 **foreach** pair of vertices v_1 and v_2 in C
6 $P =$ set of control-flow nodes that lie on any path between $v1$ and v_2 that does not go through any control-flow node containing any other vertex in C
7 **if** (P is not empty)
8 $R_{v_1,v_2} = Weighted\_Dist(v_1, v_2, P, U, D, G)$
end
end
end
end

Reuse with Reverse Control Flow (Loops)

These dependence edges reduce the unique memory access count



Challenges

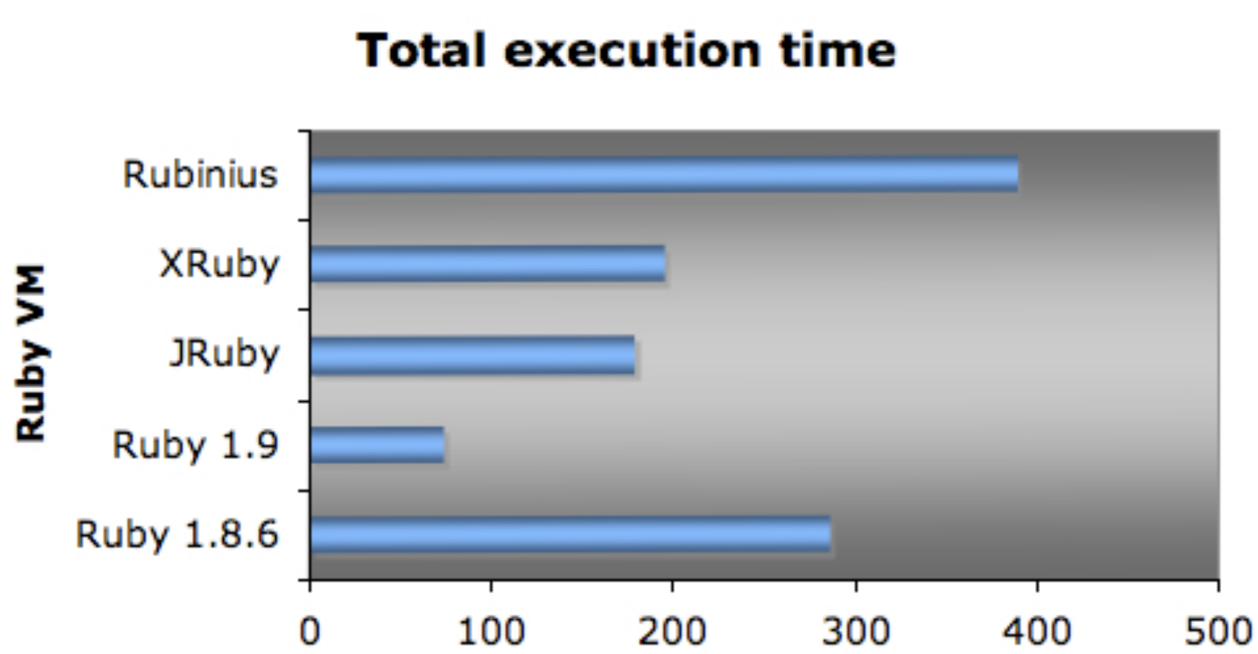
- Correlation between source-level and binary reuse distances
- Efficient estimation of source-level reuse distances
- Composition
  - ★ Bottom up computation
  - ★ Empirical methods for “black box” libraries
- Effective and efficient summarization
- Code optimization using the source-level reuse distance information

RUBY

Ruby on One Slide

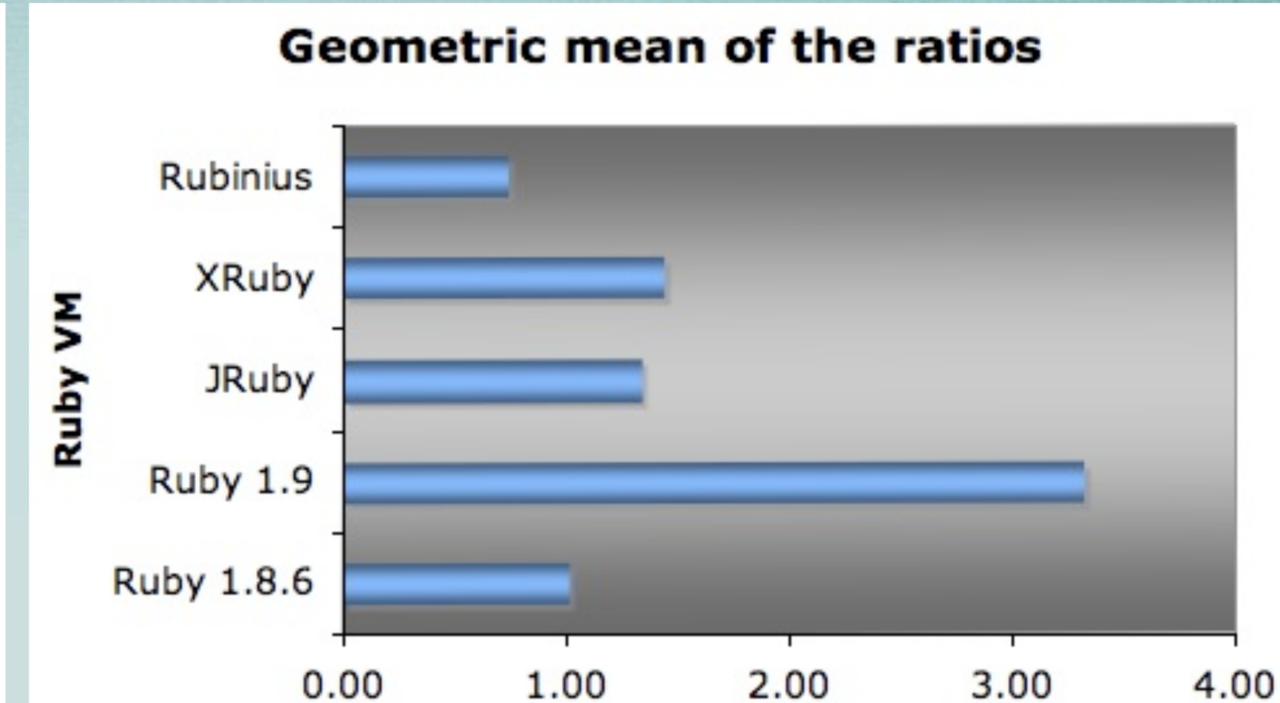
- Fully object oriented
  - ⋆ Derived from Smalltalk
  - ⋆ Everything is an object, including classes
  - ⋆ No standard operators
- Powerful meta-programming support
  - ⋆ Classes and objects may be redefined
  - ⋆ Methods may be (re/un)defined
- Advanced language features
  - ⋆ Co-routines
  - ⋆ Continuations
  - ⋆ Blocks (generalized anonymous functions)

Ruby is Slow!



	Ruby 1.8.6	Ruby 1.9	JRuby	XRuby	Rubinius
Total	284.713	71.7655	177.319	193.74	386.971

Total in Seconds



	Ruby 1.8.6	Ruby 1.9	JRuby	XRuby	Rubinius
Geo. Mean	1.00	3.32	1.32	1.43	0.73

*Courtesy: **Zen and the Art of Programming***
By Antonio Cangiano, Software Engineer & Technical Evangelist at IBM
http://antoniocangiano.com/2007/12/03/the-great-ruby-shootout/

Current Efforts on Ruby

- Ruby 1.8: The current stable implementation, also known as the MRI (Matz Ruby Interpreter)
- Ruby 1.9: The YARV based implementation of Ruby, faster with language improvements, but incompatible with 1.8
- JRuby: Ruby on the Java VM, both interpreter and compiler to Java bytecode (Ruby 1.8)
- XRuby: Ruby to Java bytecode compiler (Ruby 1.8)
- IronRuby: Ruby on the Microsoft CLR (Ruby 1.8)
- Rubinius: Ruby compiler based on the Smalltalk 80 compiler (Ruby 1.8)
- MacRuby: MacRuby, port of Ruby 1.9 to Objective-C, using Objective-C objects to implement Ruby and the Objective-C 2.0 garbage collector (Ruby 1.9)

Partially Evaluating Ruby

- Bottom-up approach
  - ★ Tabulate the “safe” primitives
    - ✳ Most are written in C
  - ★ Partially evaluate include libraries
  - ★ Partially evaluate the user code
- Target applications
  - ★ Ruby on Rails
  - ★ Home-grown graduate student database management
- Does not work for libraries
- Too much redundant effort
  - ★ Can we partially evaluate libraries conditionally?

Current Status

- Ruby front-end
- C front-end for semi-automatically classifying primitives as "safe" for partial evaluation
- Software infrastructure for partial evaluation by interfacing with Ruby interpreter

Other Projects

- Declarative approach to parallel programming
  - Let users specify parallelism
  - Separate computation from communication
  - Funded by NSF two days ago!
- MPI-aware compilation
  - Optimizing legacy MPI code for multicore clusters
- Distributed VM for Ruby
- Parallelizing for heterogeneous targets

THANK YOU!

http://www.cs.indiana.edu/~achauhan/
http://phi.cs.indiana.edu/