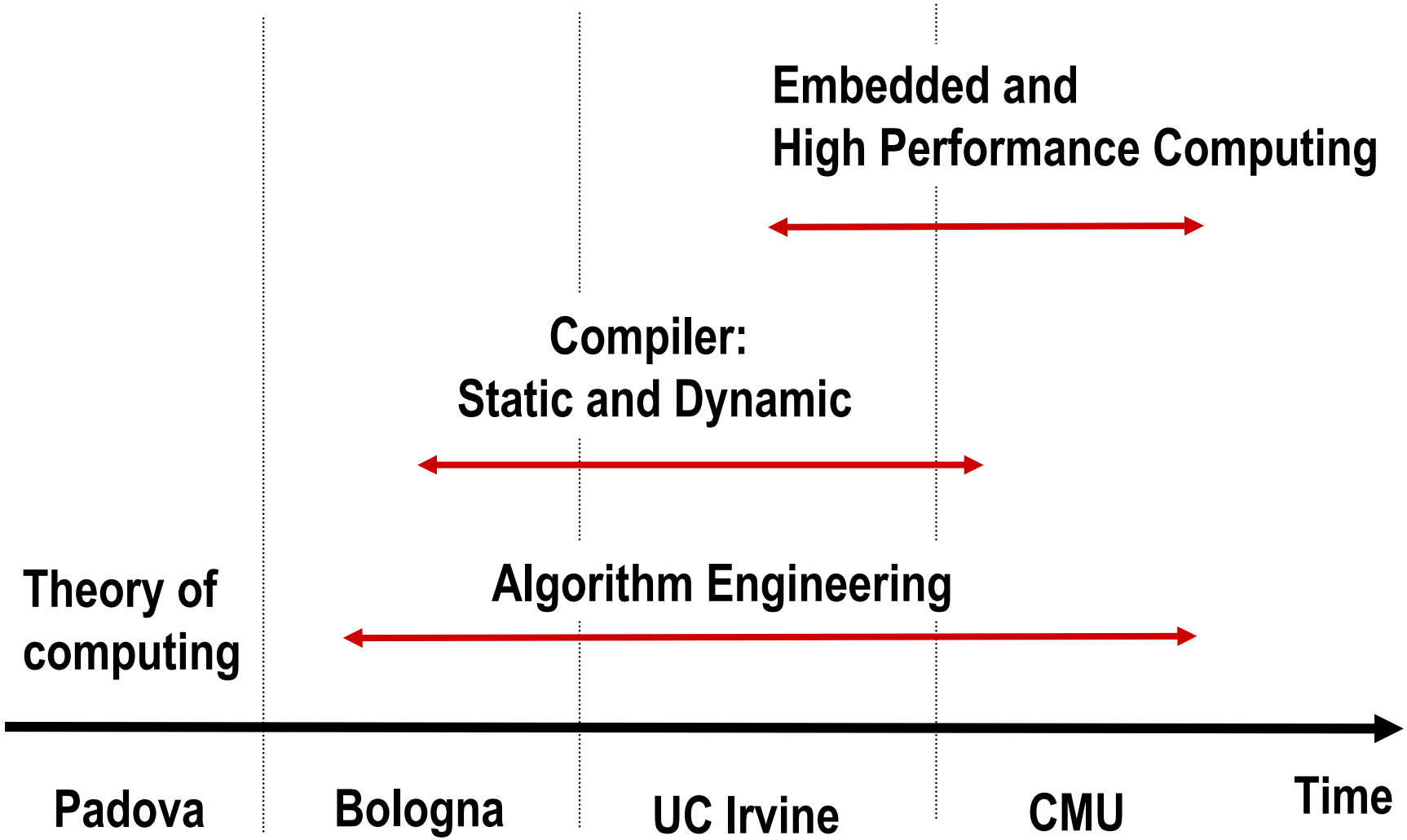
Algorithm Engineering

Paolo D'Alberto
Electrical and Computer Engineering
Carnegie Mellon University

Personal Research Background



Algorithm Engineering (*A*E)

What is it ?

- **The research of known solutions with new technologies**
- **Search of the best implementation of an algorithm**
- **Code generation (SPIRAL, ATLAS)**

When we reache the technology limits, what should AE be ?

- **Strassen's algorithm is often label as the first example of AE**
- **The re-evaluation of known algorithms and discovery of new ones**
  - Thanks to a drastic change of the computation paradigms
- **It must be open and revolutionary**

Algorithm Engineering (*A*E)

Two recent examples :

- Kleene's all pair shortest path (APSP) and Strassen's Matrix Multiplication (MM)
- They are recursive algorithms
- They exploit data locality
- Where FEWER operations mean MORE performance

R-Kleene
All-Pair Shortest Path Algorithm (APSP)

Problem (all-pairs shortest path)

- **Given a directed graph *G=(V,E)***
  - ***V*** nodes labeled as {0,1,2, …n-1} with n = |***V***|
  - ***E*** edges s.t. |***E***|<=n^2 (if (j,k) in ***E*** is unique)
- **A is an adjacency matrix of *G***
  - a_{jk} in Z or R is the cost to go from node j to node k;
  - a_{jk} = 0, if j=k;
  - a_{jk} = **infinity**, if (j,k) is not in ***E***;

- **We compute the power matrix**
  $A^* = A^n = A\,A^{(n-1)}$
- C = AB **is the matrix multiplication**
  $c_{ij} = \Sigma_{k=[0,n-1]}\, a_{ik} * b_{kj}$
- **with** $a_{ik} * b_{kj} = a_{ik} + b_{kj}$
- **with** $\Sigma_{k=[0,n-1]}\, z_k = \min(z_0\,,\, \Sigma_{k=[1,n-1]}\, z_k)$

Related Work (APSP ⇔ MM in a semi-ring)

- **Dijkstra's [1959] $O(n^3)$**
  - Shortest path for all nodes (used for sparse algorithms)
- **Floyd-Warshall's [1962] $O(n^3)$**
  - Used for dense graphs
- **Kleene's [1974] $O(n^3)$**
  - This is the first blocked algorithm
- **Park et al. [2002] $O(n^3)$**
  - Recursive and cache-aware algorithm
- **Sung-Chul et al. [2006] $O(n^3)$ (Spiral)**
  - ATLAS-like implementation of blocked Floyd-Warshall
- **Transitive Closure $O(n^{2.3})$**
  - The four-Russians algorithm $O(n^3/\log n)$,
  - Extension to a ring: Strassen-Winograd, Pan, Coppersmith

Kleene [1974] -- > Recursive & cache oblivious

Ullman and Yannakis 1990

(1) for k = 1 to $n/\sqrt{s}$ do begin
(2) $M_{k,k} = M^*_{k,k}$,
(3) for i = 1 to $n/\sqrt{s}$ do
(4) for j = 1 to $n/\sqrt{s}$ do
(5) $M_{i,j} = M_{i,j} + M_{i,k} \times M_{k,k} \times M_{k,j}$
end

Kleene for k=1,2

```
Kleene(J) {
  /*          | A  B |  */
  /* J =      | C  D |  */

  1:  A  = FLOYD_WARSHALL(A);
  2:  A += A*A*A;
  3:  B += A*A*B;
  4:  C += C*A*A;
  5:  D += C*A*B;

  6:  D  = FLOYD_WARSHALL(D);
  7:  A += B*D*C;
  8:  B += B*D*D;
  9:  C += D*D*C;
 10:  D += D*D*D;
}
```

We compute J* where J is an adjacency matrix *n* x *n*

R-Kleene [2007]

R-Kleene with only Algebraic Transformations

```
R-Kleene(J) {
  /*        | A  B | */
  /* J  =   | C  D | */

  1:  A = R-Kleene(A);
  2:  B += A*B;
  3:  C += C*A;
  4:  D += C*B;

  5:  D = R-Kleene(D);
  6:  B += B*D;
  7:  C += D*C;
  8:  A += B*C;
}
```

Self Matrix Multiplication

```
J += J*J {
  /*        | A  B | */
  /* J  =   | C  D | */

  A += A*A;
  B += A*B;
  C += C*A;
  D += C*B;

  D += D*D;
  B += B*D;
  C += D*C;
  A += B*C;
}
```

We inherit the computational property of MM:

- I/O complexity $\Theta(n^3/s)$ with Cache size s^2 (e.g. Cache 64KB s = 512)

R-Kleene: Balanced Division Process

```
R-Kleene(J) {
  /*          | A   B | */
  /* J   =    | C   D | */

  1:   A = R-Kleene(A);
  2:   B += A*B;
  3:   C += C*A;
  4:   D += C*B;

  5:   D = R-Kleene(D);
  6:   B += B*D;
  7:   C += D*C;
  8:   A += B*C;
}
```

- **We could make A small and D large**
  - Fish spine recursion tree (tail recursion)
  - D+=C*B and R-Kleene(D) dominant
  - The rest particular cases
- **We chose a balance division A~B~C~D**
  - Balanced recursion tree
  - Similar operands size
  - Similar operation complexity
  - Eliminating particular cases

We compute J* where J is an adjacency matrix *n* x *n*

R-Kleene: Parallelism and Register Allocation

```
R-Kleene(J) {
  /*        | A  B |  */
  /* J  =   | C  D |  */

  1:  A = R-Kleene(A);
  2:  B += A*B;
  3:  C += C*A;
  4:  D += C*B;

  5:  D = R-Kleene(D);
  6:  B += B*D;
  7:  C += D*C;
  8:  A += B*C;
}
```

- D+=C*B and A+=B*C are MM with different operands and destinations
- We can apply aggressive register allocation (e.g., MM register allocation such as in ATLAS)
- We proved that when A and D are Kleene's closure matrices, we can apply the same aggressive schedule to

  B+=A*B, C+=C*A, B+=B*D, and C+=D*C

Memory Accesses:

From $2n^3$ to $(2/r)n^3$

with $1 <= r^2 < R$ registers available.

For example, with r=2, we half the memory accesses

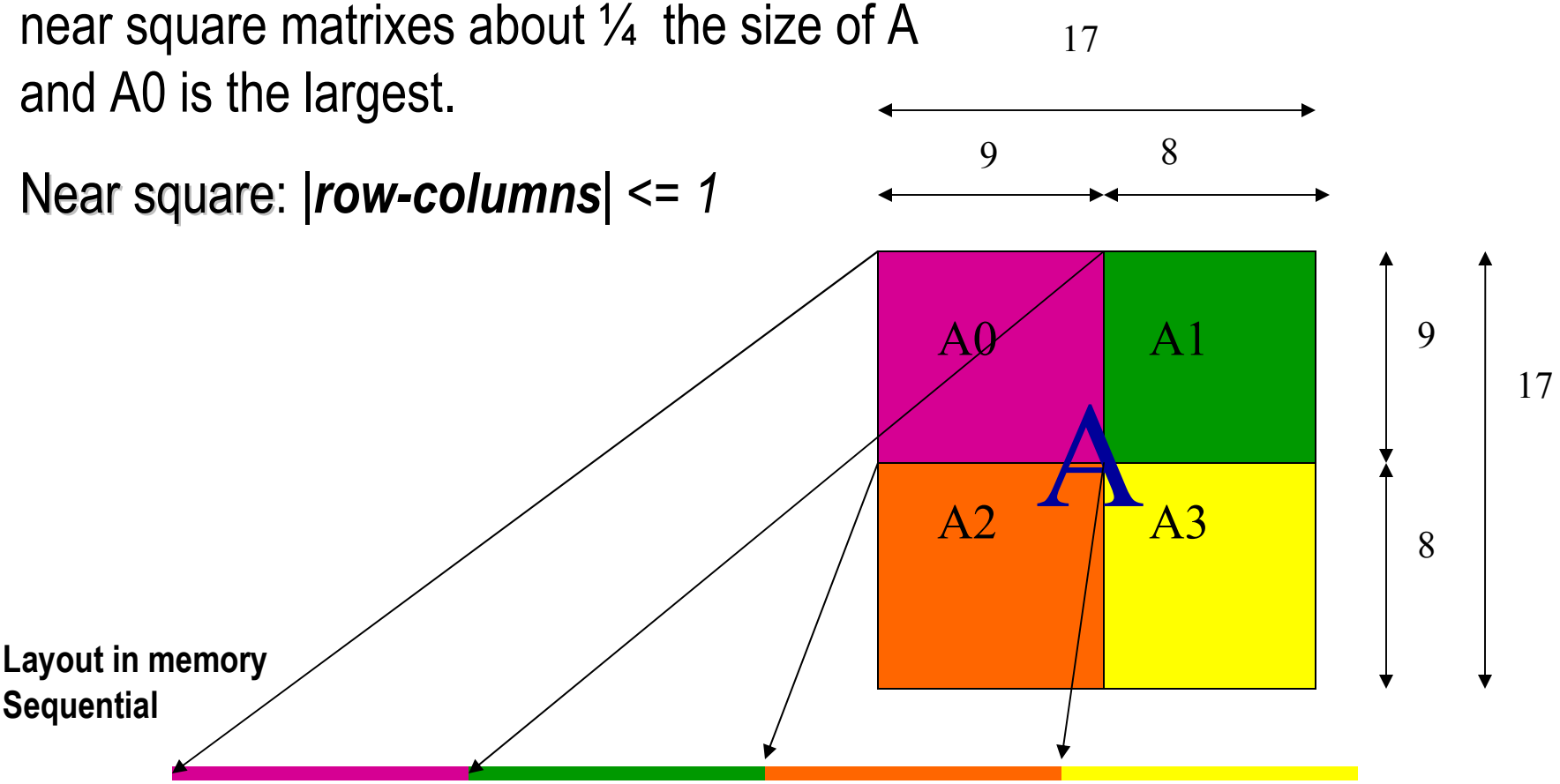
Experimental Setup

- **We tested 4 algorithms (What algorithms ?)**
  - R-Kleene (***A*** is row-major matrix)
  - Floyd-Warshall **FW** (***A*** is row-major matrix)
  - Simple Recursive (**Z-SR**), extension of Park et al. with Z-Morton layout (next slide Z-Morton)
  - ZR-Kleene is the R-Kleene algorithm, Z-Morton layout
- **We quantify (Why these algorithms ?)**
  - The effects of the register allocation alone
  - The effects of the matrix layout alone
  - The effects of matrix layout and register allocation
- **On 5 machines (on what architectures ?)**
  - We measure Million of Instructions Per Second (MIPS)

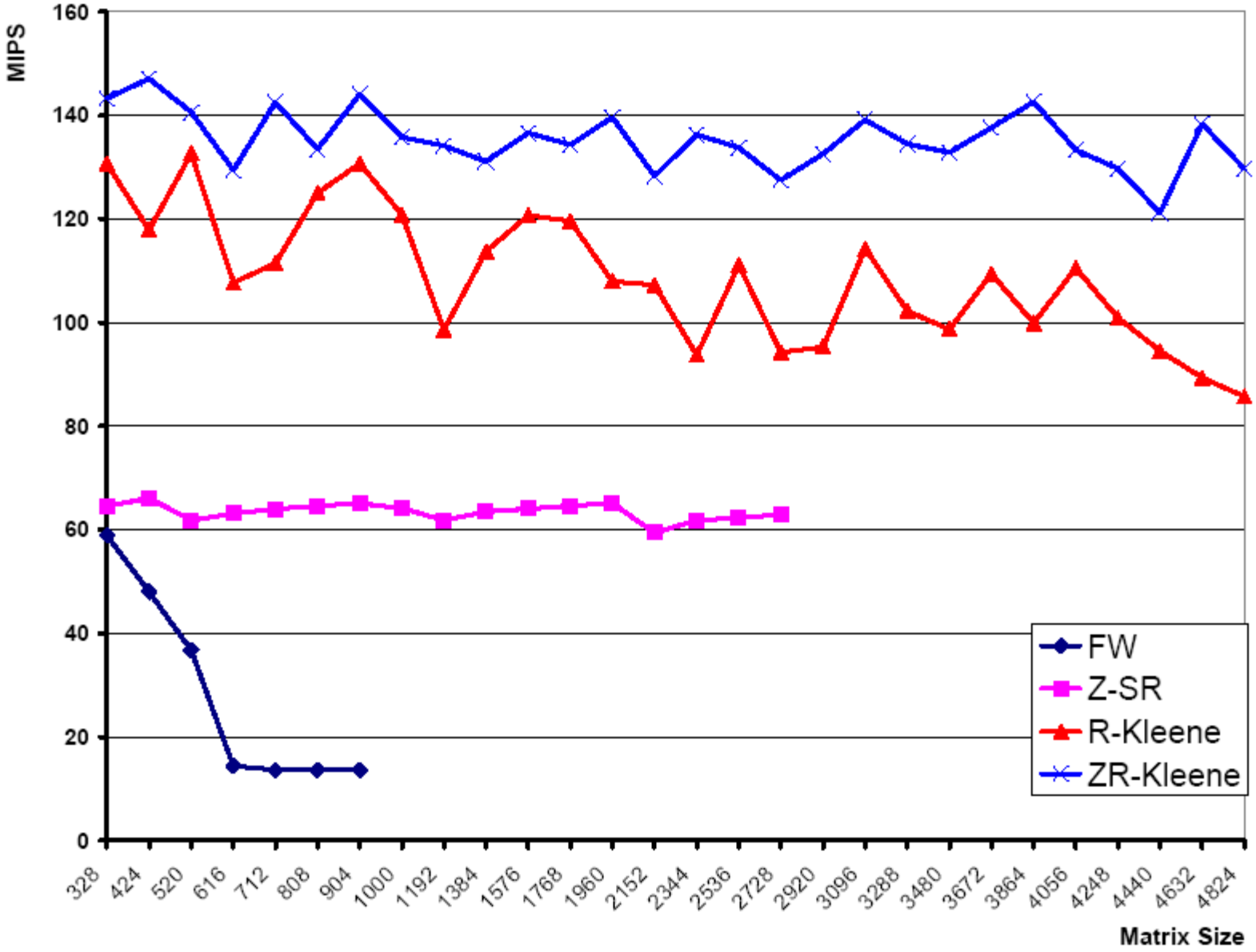
Z-Morton Layout

A is near square matrix then A0, A1, A2, A3 are near square matrixes about ¼ the size of A and A0 is the largest.

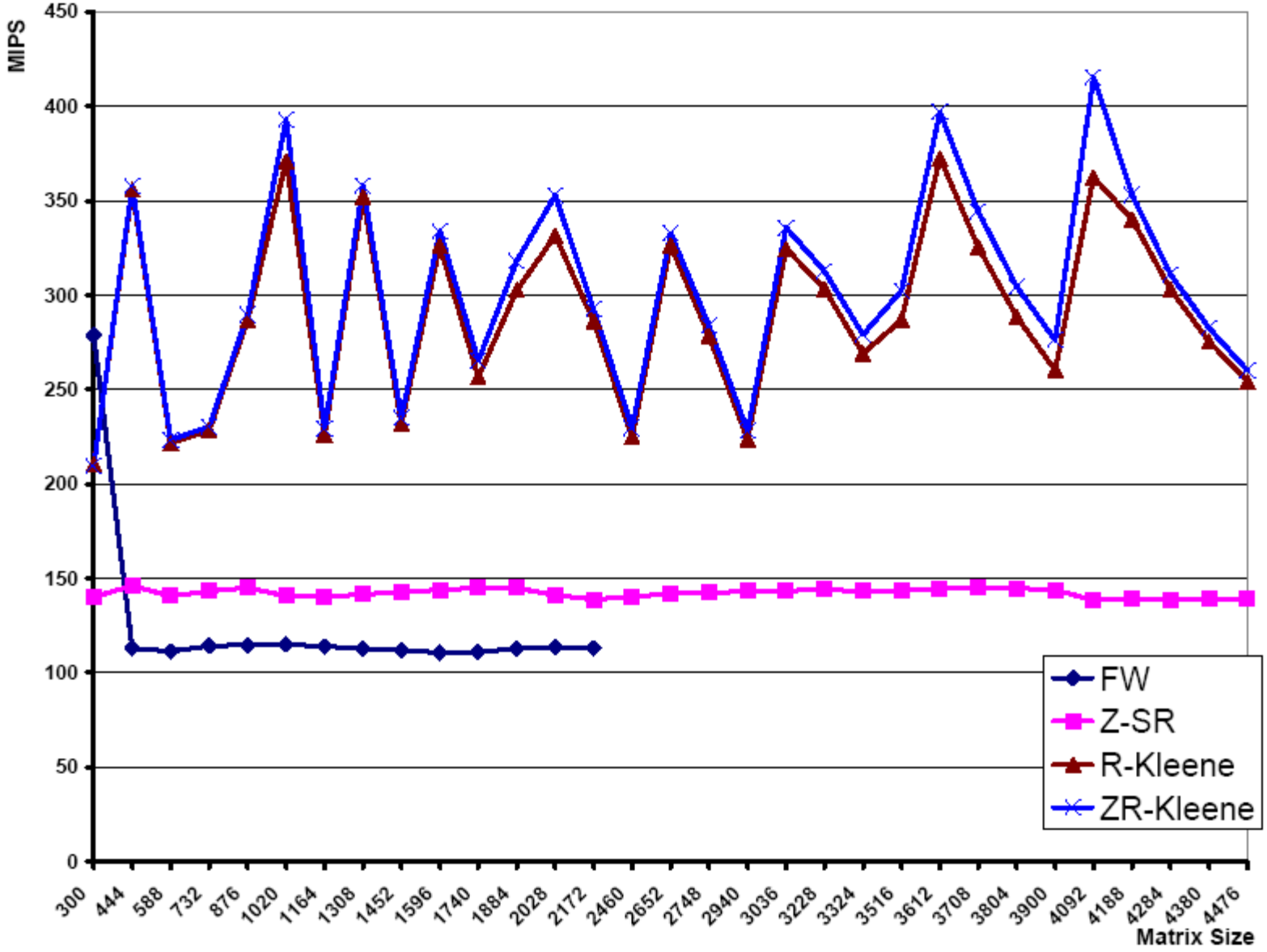
Near square: |***row-columns***| <= *1*



Experimental Results (R12K 300 MHz)



Experimental Results (Athlon 2800+ 2GHz)

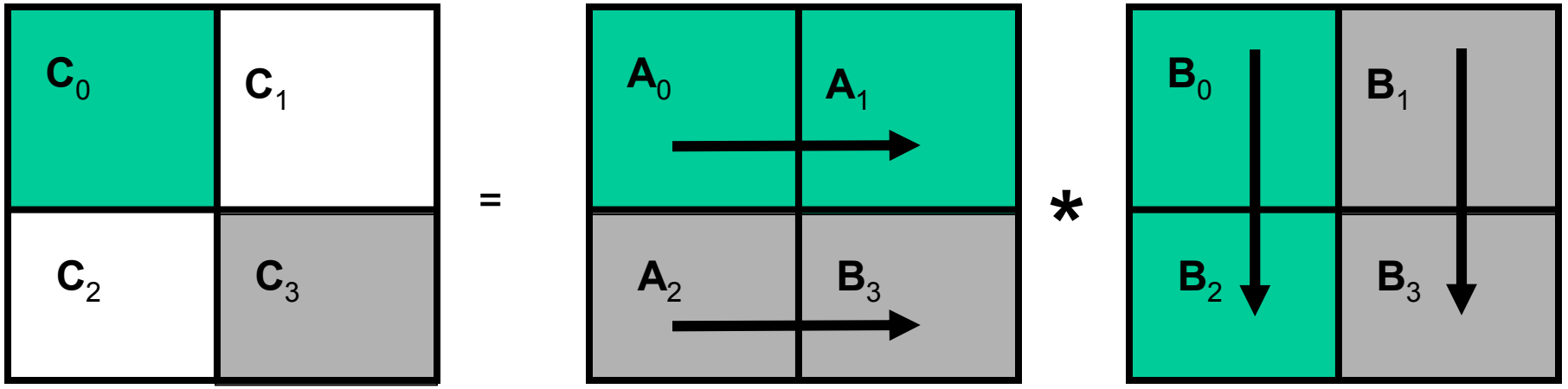


R-Kleene: Conclusions

- **We used the Kleene's algorithm as starting point**
  - Good locality because blocked
- **We obtain a recursive algorithm by algebraic reduction**
  - Correct by construction
  - Cache oblivious
- **We discover algebraic property of the algorithm so that:**
  - Parallelism is explicit in the computation
  - The computation order is revisited
  - The computation order is suitable to aggressive register allocation
- **We propose and investigate the effects of matrix layout and register allocation**

Adaptive Strassen DGEMM

Matrix Multiplication (basics)



$C_0 = A_0B_0 + A_1B_2$

$C_1 = A_0B_1 + A_1B_3$

$C_2 = A_2B_0 + A_3B_2$

$C_3 = A_2B_1 + A_3B_3$

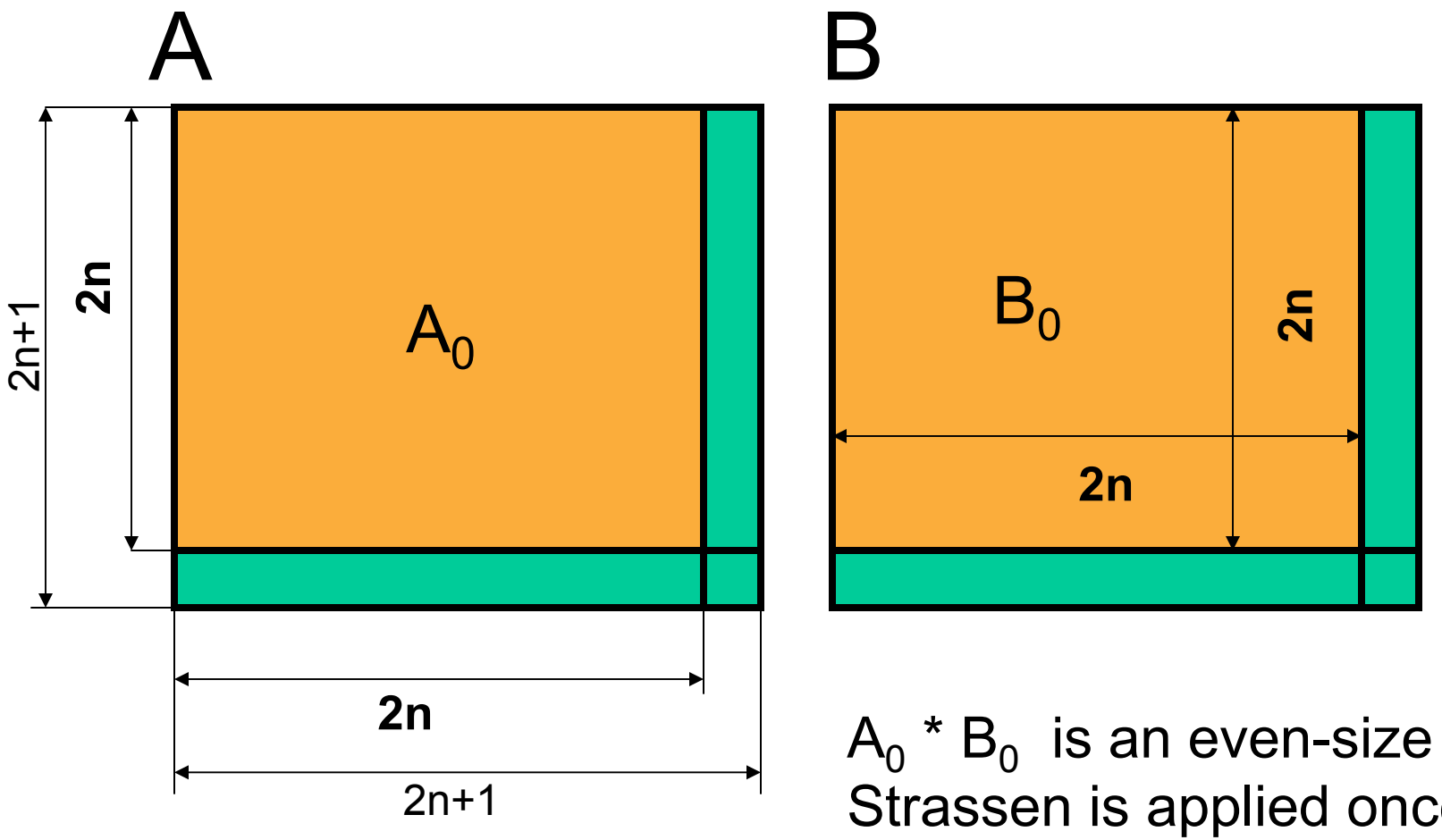
Related work: Matrix Multiply (MM)

- **Precursors of ATLAS (library and almost code generators)**
  - PHiPac (classic only)
  - ESSL (classic and Winograd)
- **ATLAS uses this classic matrix multiply (first automatic code generator)**
  - For square matrices of size nxn, the algorithm takes $O(n^3)$
  - It achieves 80-90% of peak performance
- **Post-ATLAS (assembly code & automatic code generation)**
  - GotoBLAS
- **Strassen's algorithm for large problems.**
  - It reduces the number of computations
  - Thus shortens the execution time
- **We investigate the effects on single-processor systems**

Related Work: Strassen's

- **Strassen [1969]**
  - For 2^n–size matrices $O(n^{\log 7})$
- **Knights [1994]**
  - For rectangular $2^n x 2^m$ size matrices
- **For even-size matrices,**
  - one recursive step is always applicable
- **For odd-size matrices**
  - Dynamic and static padding (extra date and thus extra computations)
  - Peeling (introduction of a conquer step)
  - Peeling is more appealing for operation counts [Huss 97 & Luo 2004]:

Odd-Size Square Matrices [Huss et al. 1996]



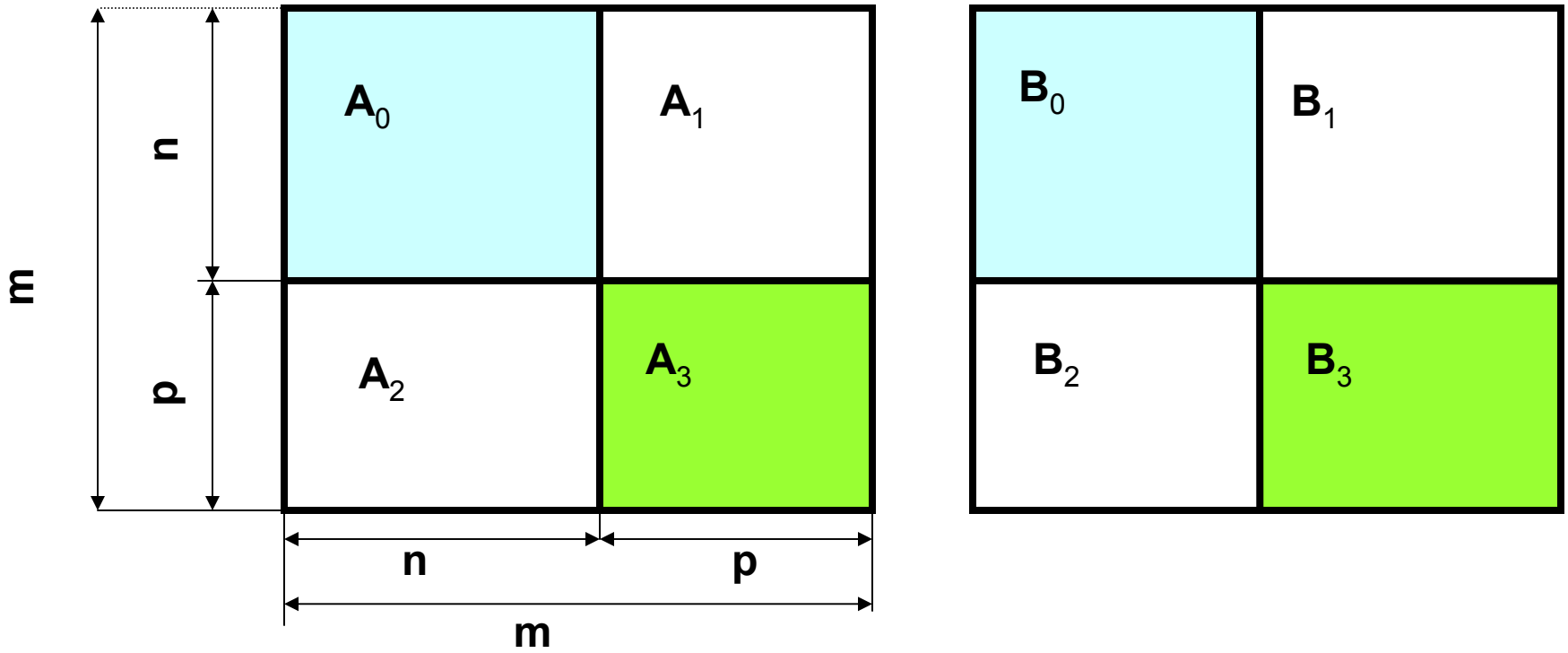
$A_0 * B_0$ is an even-size problem.
Strassen is applied once more

Our Approach for Square Matrices: a Balanced Division

- **For any matrix size, we apply a balanced Strassen's division process**
  - This reduces the number of computations further than an odd/even size problem (or padded)
- **Balanced division = balanced workload**
  - Thus, predictable performance
  - No particular cases
- **Balanced sized operands**
  - Better data cache utilization

Balanced Division Matrices

Near Square: m = n+p with min|n-p|



The quadrants are near square matrices.
At any step of the recursion, all sub-matrices are near square matrices

Experimental Results

- **We considered 14 systems (currently more than 19)**
  - We hand coded the MA for each specific system
  - (we then start using a single/simple MA)
- **We measure performance of ATLAS's MM and MA**
  - We specify an adaptive recursion point size for each system
  - We encode the recursion point in the algorithm
- **We measured the relative performance of**
  - Our Strassen vs ATLAS
  - Our Strassen vs. GotoBLAS
- **We report the details for 2 systems shortly**

Break-even size

Estimated Break-even size

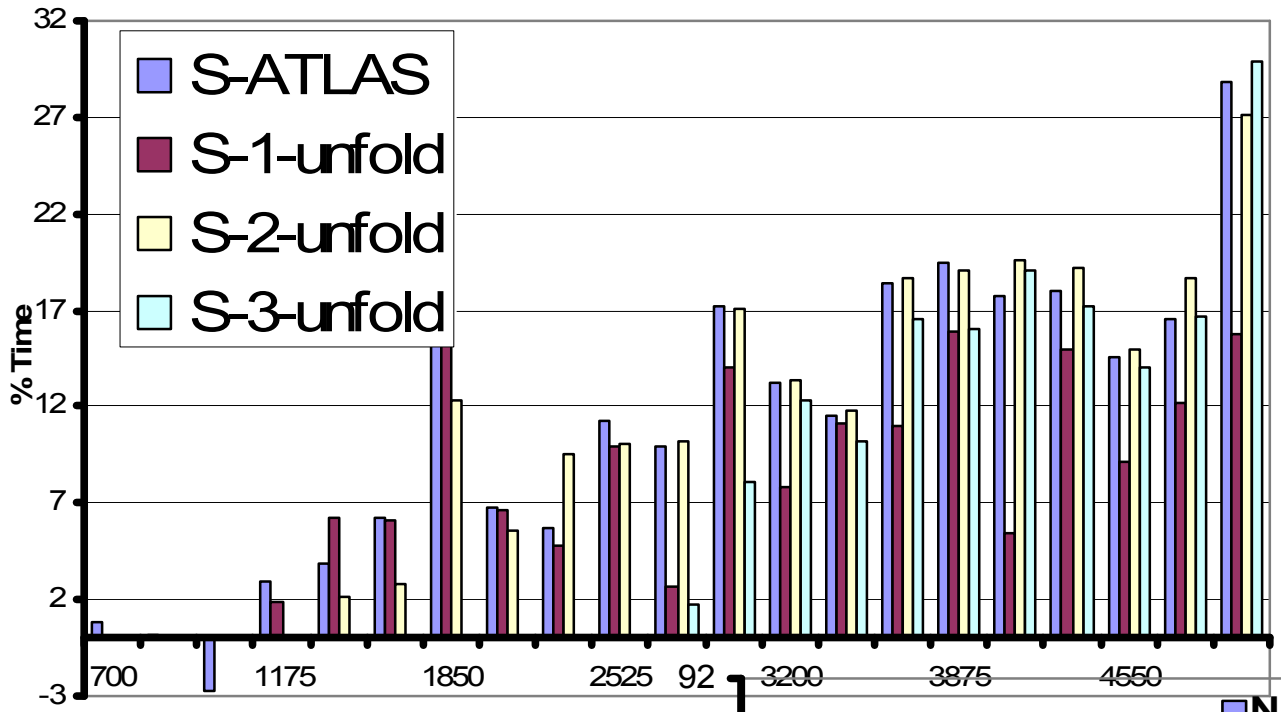
System	Processors	π^{-1} $\times 10^6$	α^{-1} $\times 10^6$	n_1	exp. n_1	Figure
Fujitsu HAL 300	SPARC64 100MHz	177	10	390	400	Fig. 3
RX1600	Itanium 2@1.0GHz	3023	105	487	725	Fig. 4
ES40	Alpha ev67 4@667MHz	1240	41	665	700	Fig. 5
RP5470	8600 PA-RISC 550MHz	763	21	772	1175	Fig. 6
Ultra 5	UltraSparc2 300MHz	407	9	984	1225	Fig. 7
ProLiant DL140	Xeon 2@3.2GHz	2395	53	995	1175	Fig. 8
ProLiant DL145	Opteron 2@2.2GHz	3888	93	918	1175	Fig. 9
Ultra-250	UltraSparc2 2@300MHz	492	10	1061	1300	No
Sun-Fire-V210	UltraSparc3 1GHz	1140	22	1140	1150	Fig. 10
Sun Blade	UltraSparc2 500MHz	460	8	1191	1884	No
ASUS	AthlonXP 2800+ 2GHz	2160	39	1218	1300	Fig. 11
Unknown server	Itanium 2@700MHz	2132	27	1737	2150	Fig. 12
Fosa	Pentium III 800MHz	420	4	2009	N/A	No
SGI O2	MIPS 12K 300MHz	320	2	2816	N/A	No

ATLAS MM(1000) MFLOPS

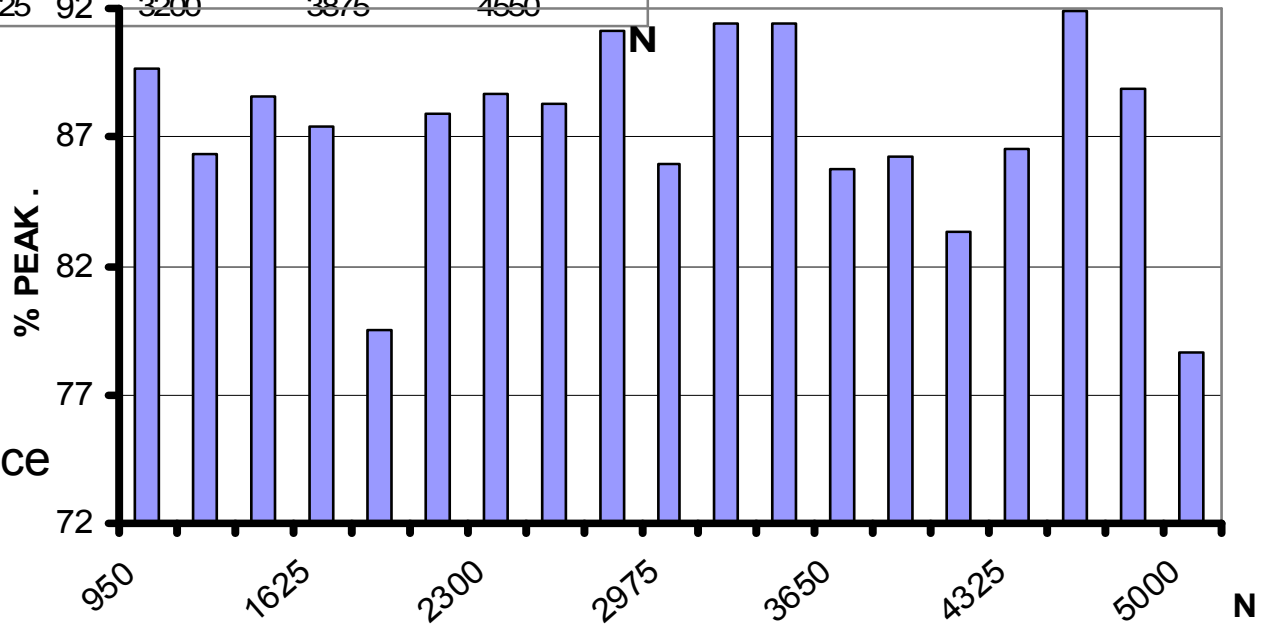
MA(1000) MFLOPS

ALPHA

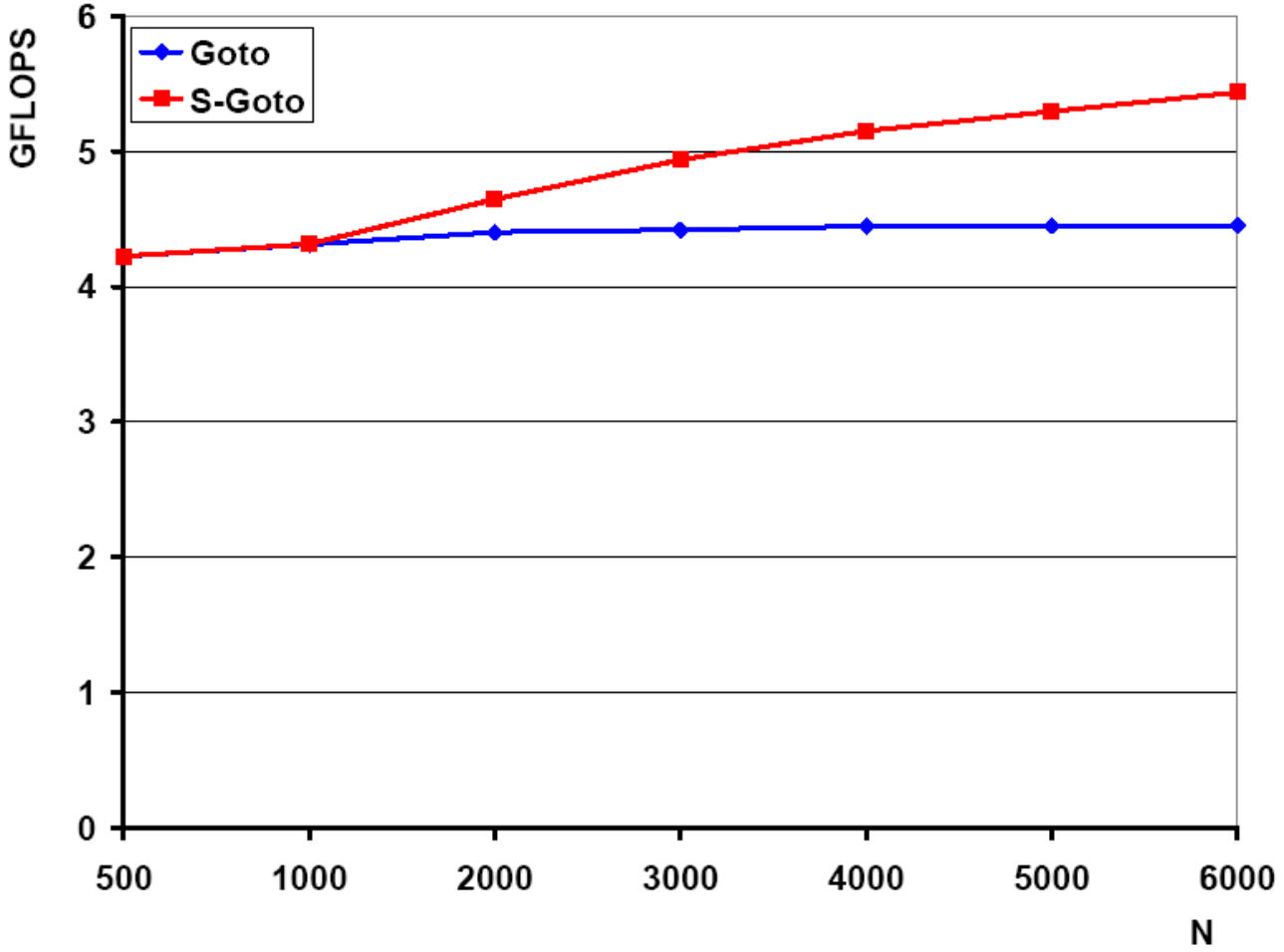
Strassen + ATLAS
Relative Time



ATLAS's
Relative Peak Performance

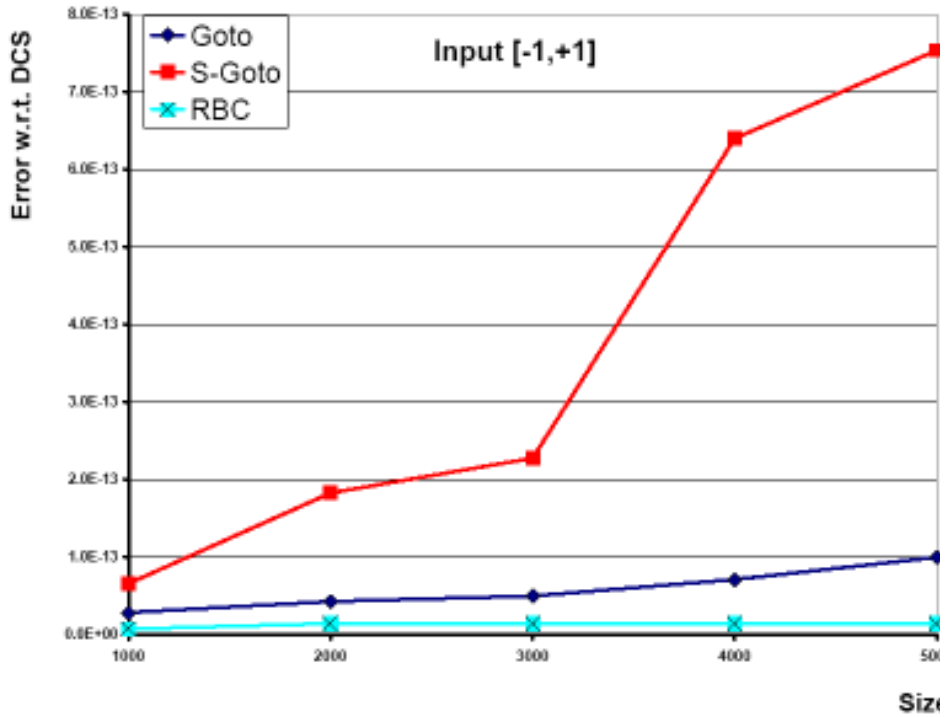


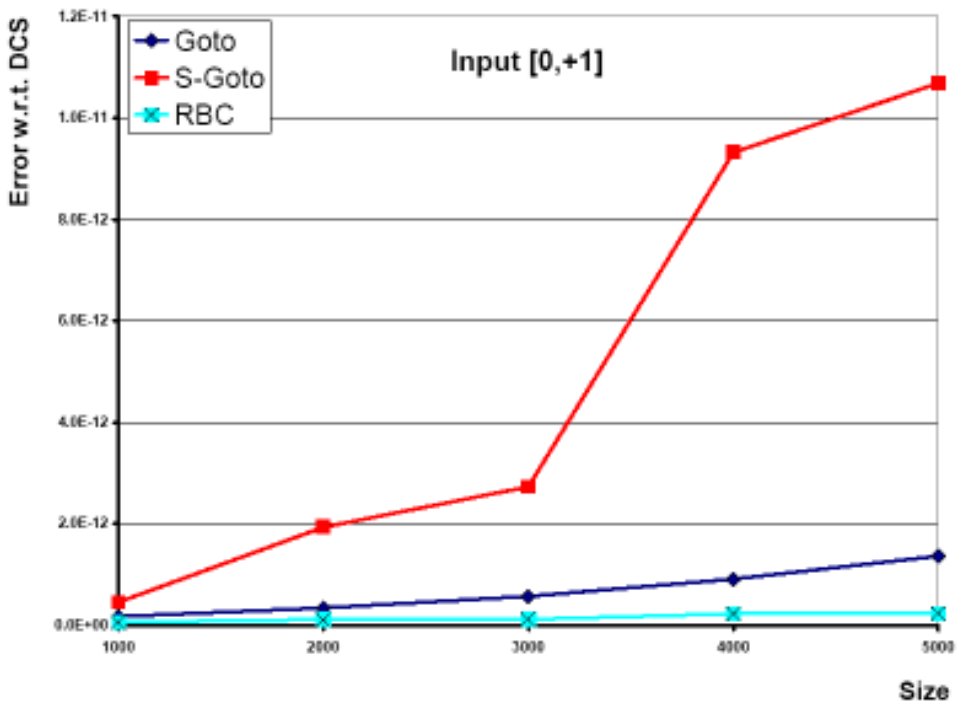
Athlon64: GotoBLAS + Strassen



- We improve ATLAS
- We improve Goto
- We improve every classic MM

Maximum Absolute Error: A Quantitative Evaluation





- **Reference: Priest's Doubly Compensated Summation (DCS)**
  - It is a technique to perform a summation with minimum error
- **The error follows a 2^x instead of 3^x**
  - It means that we loose one decimal digit every three level of recursion of the 16 available.

Conclusions

- **The core of Algorithm engineering is**
  - Re-evaluation of the problem
  - Re-evaluation of the current state-of-the-art solution
  - Dare to ask the obvious questions
  - Investigate/experiment with care and caution
- **Our approaches use the balanced division**
  - However, unbalanced divisions are possible and easy to investigate
  - And could be investigated using code generators
- **We performed an exhaustive testing of performance**
  - Some architectures do not offer any practical performance opportunity
  - Neither for Strassen nor R-Kleene


Thank you