

Software in the Era of Parallelism

David Padua

University of Illinois at Urbana-Champaign



Outline of the talk

- I. Introduction
- II. Languages
- III. Automatic program optimization
 - Compilers
 - Program synthesizers
- IV. Conclusions

I. Introduction (1): The era of parallelism

- The era of parallelism is (finally !) here
- Are we ready for it ?
- After all, we knew for 40 years that it was coming

I. Introduction (2): What we did

- Parallel algorithms.
- Widely used *parallel programming notations*
 - Distributed memory (SPMD/MPI) and
 - Shared memory (pthreads/OpenMP).
- *Compiler and program synthesis algorithms*
 - Automatically map computations and data onto parallel machines/devices.
 - Detection of parallelism.
- Tools. Performance, debugging. Manual tuning.
- Education.

Main focus in the past: numerical computing

I. Introduction (3): Why we did it

- Main goal of software studies: to reduce the additional cost of parallelism. (The same is true of computer architecture)
 - Want efficiency/portable efficiency

I. Introduction (4): But ...

- But much remains to be done and, most likely, widespread parallelism will give us performance at the expense of a dip in productivity.

I. Introduction (5): The future

- Although we learned that advances are not easy (Software only seems easy after the fact), we have now many ideas and significant experience.
- And ... Industry interest → more resources to solve the problem.
- The extensive experience of massive deployment will also help.
- The situation is likely to improve rapidly. Exciting times ahead.

Outline of the talk

- I. Introduction
- II. Languages**
- III. Automatic program optimization
 - Compilers
 - Program synthesizers
- IV. Conclusions

II. Languages (1): OpenMP and MPI

- OpenMP constitutes an important advance, but its most important contribution was to unify the syntax of the 1980s (Cray, Sequent, Alliant, Convex, IBM,...).
- MPI has been extraordinarily effective.
- Both have mainly been used for numerical computing. Both are widely considered as “low level”.
- Alternatives have been designed. Next: an example of higher level language for numerical computing.

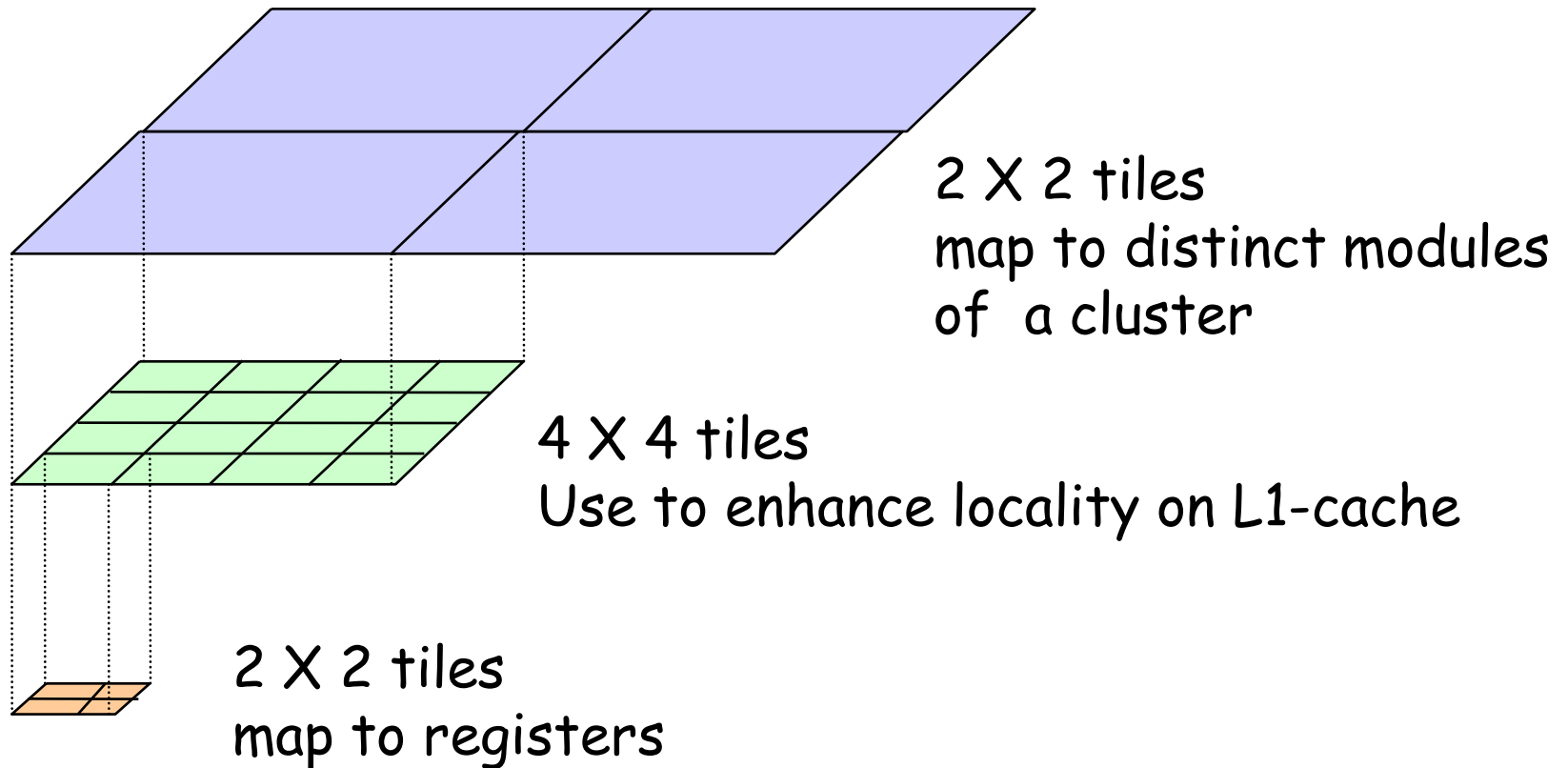
II. Languages (2): Hierarchically Tiled Arrays

- Recognizes the importance of blocking/tiling for locality and parallel programming.
- Makes tiles first class objects.
 - Referenced explicitly.
 - Manipulated using array operations such as reductions, gather, etc..

Joint work with IBM Research.

G. Bikshandi, J. Guo, D. Hoeflinger, G. Almasi, B. Fraguera, M. Garzarán, D. Padua, and C. von Praun. Programming for Parallelism and Locality with Hierarchically Tiled. *PPoPP*, March 2006.

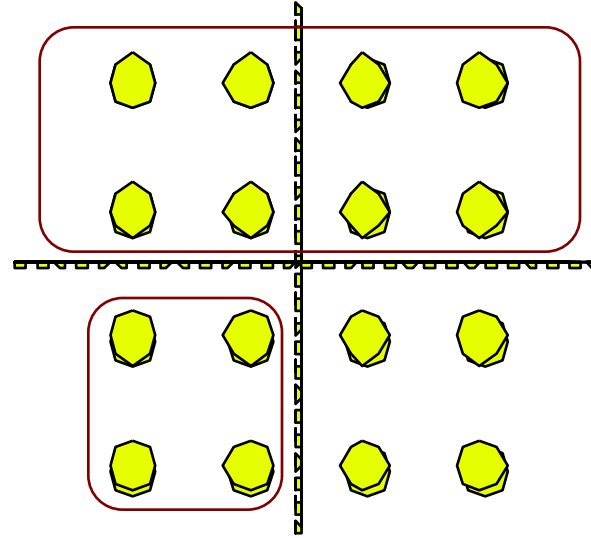
II. Languages (3): Hierarchically Tiled Arrays



II. Languages (4): Accessing HTAs

tiles

$h\{1, 1:2\}$



$h\{2, 1\}$

hierarchical

II. Languages (5): Tiled matrix-matrix multiplication

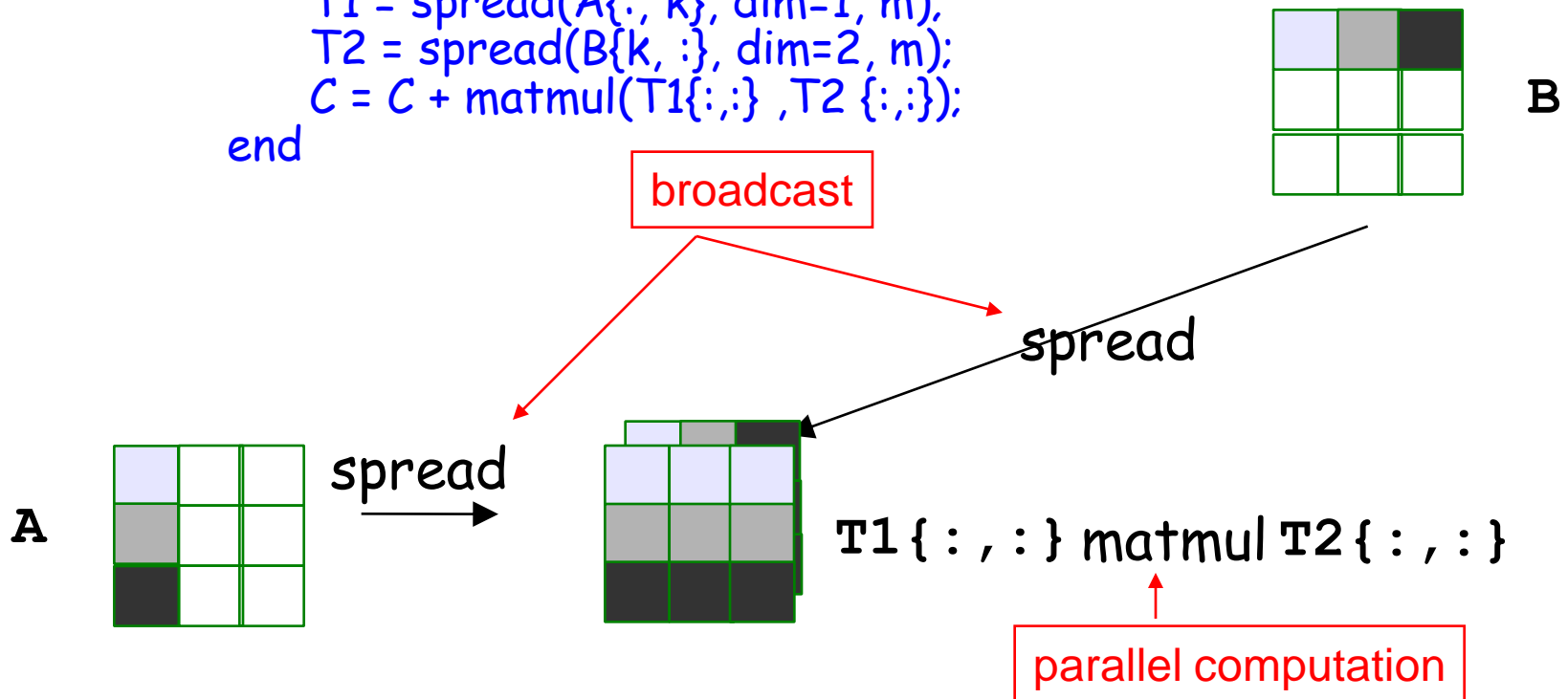
```
for I=1:q:n
  for J=1:q:n
    for K=1:q:n
      for i=I:I+q-1
        for j=J:J+q-1
          for k=K:K+q-1
            C(i,j)=C(i,j)+A(i,k)*B(k,j);
          end
        end
      end
    end
  end
end
```



```
for i=1:m
  for j=1:m
    for k=1:m
      C{i,j}=C{i,j}+A{i,k}*B{k,j};
    end
  end
end
```

II. Languages (6): Parallel matrix-matrix multiplication

```
function summa (A, B, C)
for k=1:m
    T1 = spread(A(:, k), dim=1, m);
    T2 = spread(B(k, :), dim=2, m);
    C = C + matmul(T1{:,:} , T2 {:,:});
end
```



FLAME

Algorithm: $[A, p] := \text{LUPIV_BLK}(A)$

Partition

$$A \rightarrow \begin{array}{c|c} A_{TL} & A_{TR} \\ \hline A_{BL} & A_{BR} \end{array}, p \rightarrow \begin{array}{c} p_T \\ \hline p_B \end{array}$$

where A_{TL} is 0×0 , p_T has 0 elements

while $n(A_{TL}) < n(A)$ do

Determine block size b

Repartition

$$\begin{array}{c|c} A_{TL} & A_{TR} \\ \hline A_{BL} & A_{BR} \end{array} \rightarrow \left(\begin{array}{c|c|c} A_{00} & A_{01} & A_{02} \\ \hline A_{10} & A_{11} & A_{12} \\ \hline A_{20} & A_{21} & A_{22} \end{array} \right),$$

$$\begin{array}{c} p_T \\ \hline p_B \end{array} \rightarrow \left(\begin{array}{c} p_0 \\ \hline p_1 \\ \hline p_2 \end{array} \right)$$

where A_{11} is $b \times b$, p_1 is $b \times 1$

$$\frac{A_{11}}{A_{21}}, p_1 := \text{LUPIV} \frac{A_{11}}{A_{21}}$$

$$\begin{array}{c|c} A_{10} & A_{12} \\ \hline A_{20} & A_{22} \end{array} := P(p_1) \begin{array}{c|c} A_{10} & A_{12} \\ \hline A_{20} & A_{22} \end{array}$$

$$A_{12} := L_{11}^{-1} A_{12}$$

$$A_{22} := A_{22} - A_{21} A_{12}$$

Continue with

$$\begin{array}{c|c} A_{TL} & A_{TR} \\ \hline A_{BL} & A_{BR} \end{array} \leftarrow \left(\begin{array}{c|c|c} A_{00} & A_{01} & A_{02} \\ \hline A_{10} & A_{11} & A_{12} \\ \hline A_{20} & A_{21} & A_{22} \end{array} \right),$$

$$\begin{array}{c} p_T \\ \hline p_B \end{array} \leftarrow \left(\begin{array}{c} p_0 \\ \hline p_1 \\ \hline p_2 \end{array} \right)$$

endwhile

HTA

```
void lu(HTA<double,2> A, HTA<int,1> p,int nb)
{
```

```
  A.part((0,0),(0,0));
  p.part((0),(0));
```

```
  while(A(0,0).lsize(1)<A.lsize(1)){
    int b = min(A(1,1).lsize(0), nb);
```

```
    A.part((1,1),(b,b));
    p.part((1),(b));
```

```
    dgetf2(A(1:2,1), p(1));
    dlaswp(A(1:2,0), p(1));
    dlaswp(A(1:2,2), p(1));
    trsm(HtaRight, HtaUpper, HtaNoTrans,
          HtaUnit, One, A(1,1),A(1,2));
    gemm(HtaNoTrans, HtaNoTrans, MinusOne,
          A(2,1), A(1,2), One, A(2,2));
```

```
    A.rmPart((1,1));
    p.rmPart((1));
```

```
  }
```

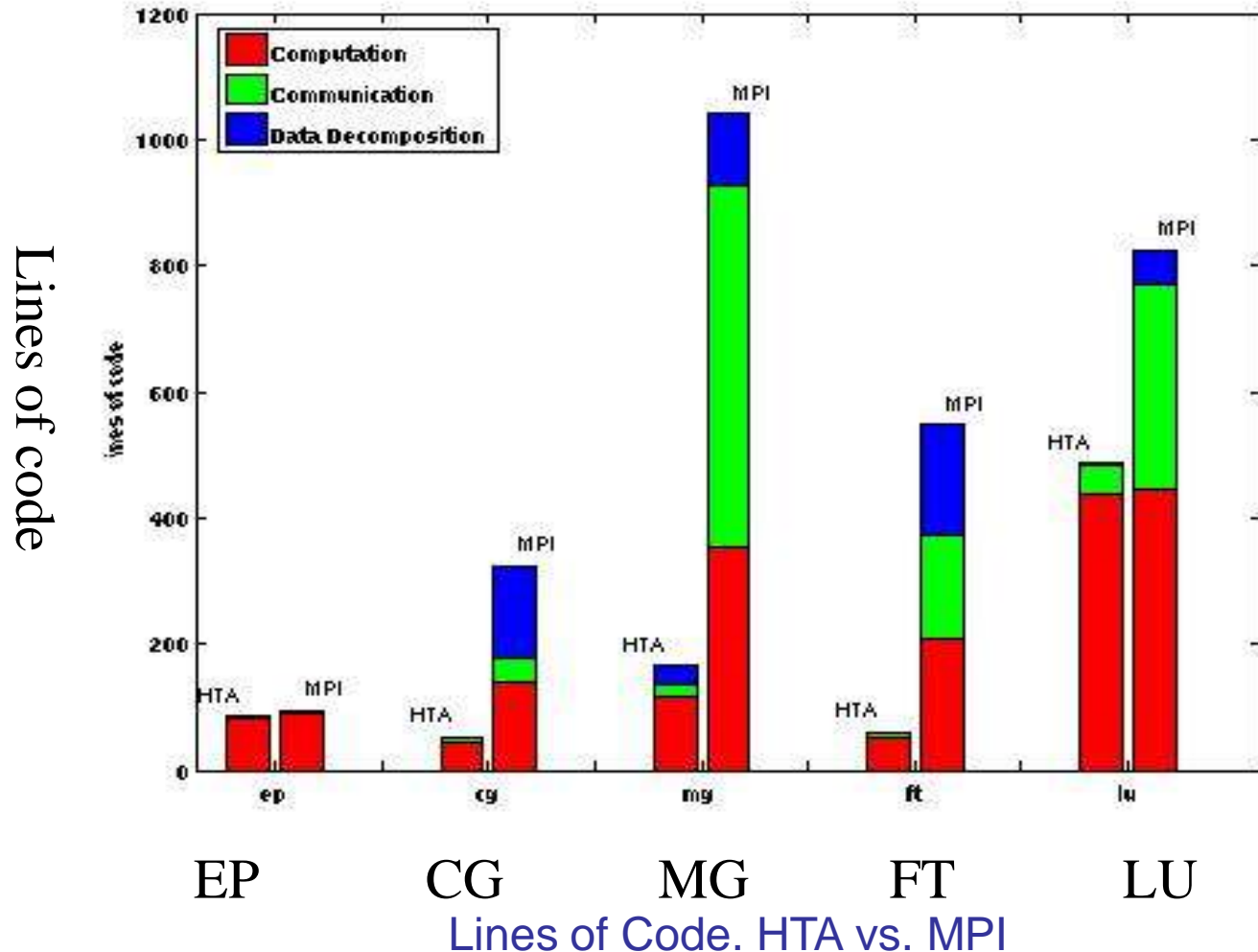
II. Languages (7):

Advantages of tiling as a first class object

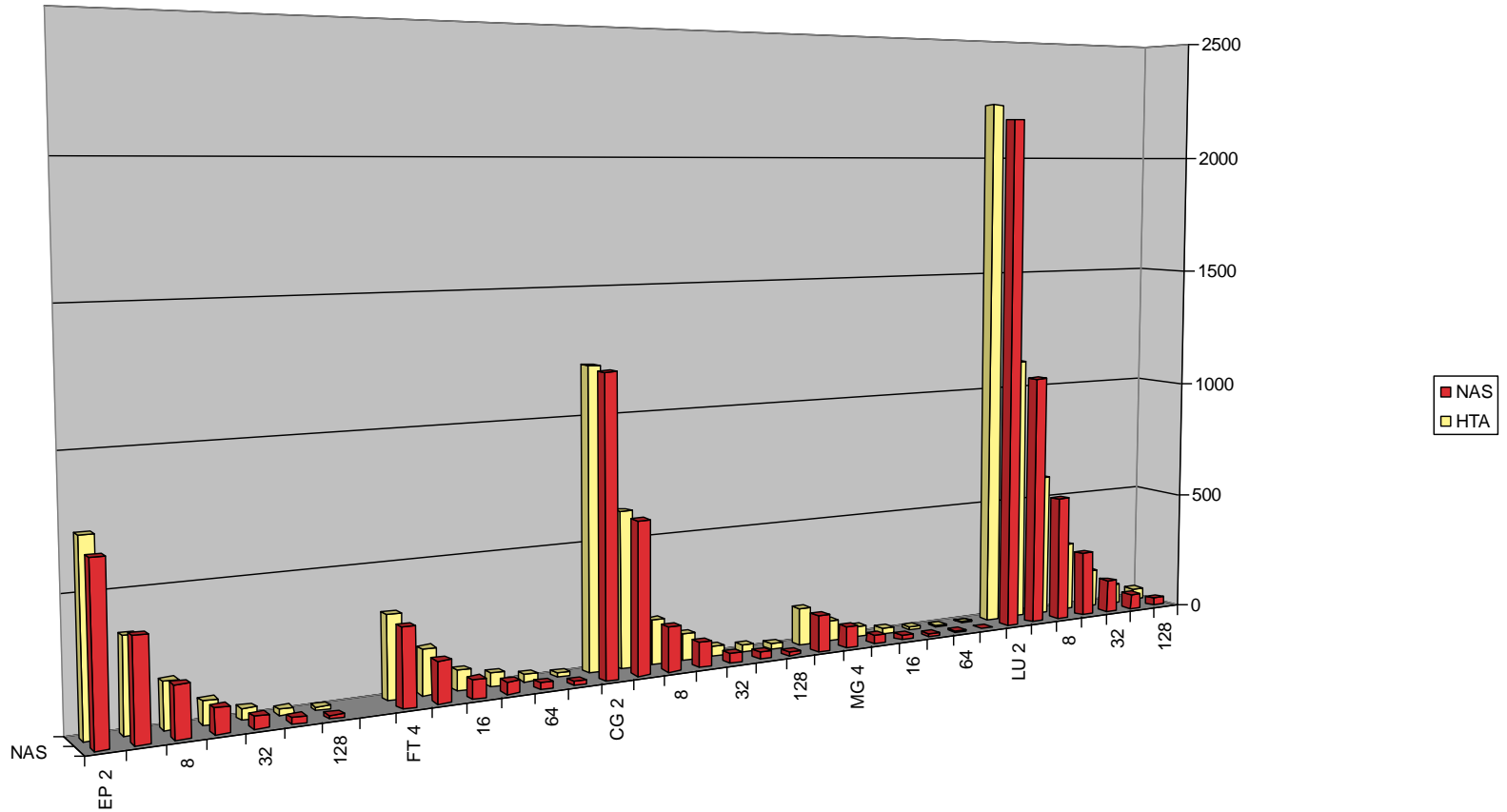
- Array/Tile notation produces code more readable than MPI. It significantly reduces number of lines of code.

II. Languages (8):

Advantages of tiling as a first class object



II. Languages (9): Performance identical to Fortran + MPI



II. Languages (9):

Advantages of making tiles first class objects

- More important advantage: Tiling is explicit. This simplifies/makes more effective automatic optimization.

```
for i=1:m
  for j=1:m
    for k=1:m
      C{i,j}=C{i,j}+A{i,k}*B{k,j};
    end
  end
end
```

Size of tiles ?



II. Languages (10):

Conclusions: What next ?

- High-level notations/new languages should be studied. Much to be gained.
- But .. **New languages by themselves will not go far enough in reducing costs of parallelization.**
- **Automatic optimization is needed.**
- Parallel programming languages should be **automatic optimization enablers.**
 - Need language/compiler co-design.
 - Libraries can be considered part of the language
New languages are not needed
But the compiler must know about the libraries

Outline of the talk

- I. Introduction
- II. Languages
- III. Automatic program optimization**
 - Compilers
 - Program synthesizers
- IV. Conclusions

III. Automatic Program Optimization (1)

- The objective of compilers from the outset.

“It was our belief that if FORTRAN, during its first months, were to translate any reasonable “scientific” source program into an object program only half as fast as its hand coded counterpart, then acceptance of our system would be in serious danger.”

John Backus

Fortran I, II and III

Annals of the History of Computing, July 1979.

III. Automatic Program Optimization (2)

- Still far from solving the problem.
The Fortran challenge is much more difficult now
- Two approaches:
 - Compilers
 - The emerging new area of program synthesis.

Outline of the talk

- I. Introduction
- II. Languages
- III. Automatic program optimization
 - **Compilers**
 - Program synthesizers
- IV. Conclusions

III.1 Compilers (1)

Purpose

- Bridge the gap between programmer's world and machine world. Between readable/easy to maintain code and unreadable high-performing code.
- The idiosyncrasies of multicore machines, however interesting in our eyes, are more a problem than a solution.
- In an ideal world, compilers or related tools should hide these idiosyncrasies.
- But, what is the hope of this happening today ?

III.1 Compilers (2)

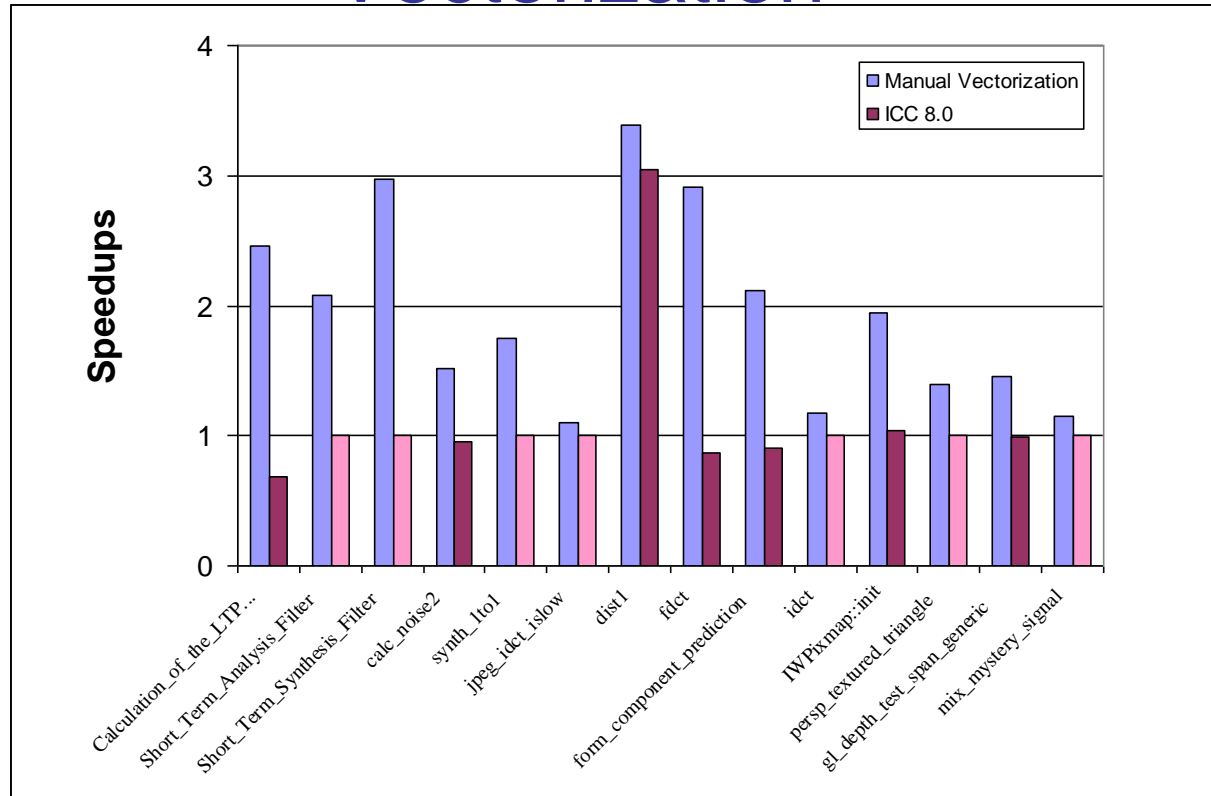
How well do they work ?

- Evidence accumulated for many years show that compilers today fail to meet our expectations.
- Problems at all levels:
 - Detection of parallelism (numerical computing)
 - Vectorization
 - Locality enhancement
 - Traditional compilation
- I'll show only results one example.

III.1 Compilers (4)

How well do they work ?

Vectorization



G. Ren, P. Wu, and D. Padua: An Empirical Study on the Vectorization of Multimedia Applications for Multimedia Extensions. IPDPS 2005

III.1 Compilers (5)

Compiler flaws

- Poor analysis
- Ad hoc optimization strategies
- Uneven implementations

III. 1 Compilers (6)

Obstacles

- Need a new generation of compilers, but
- Several factors conspire against progress in program optimization
 - The myth that the automatic optimization problem is solved or insurmountable.
 - The natural desire to work on fashionable problems and “low hanging fruits”

Outline of the talk

- I. Introduction
- II. Languages
- III. Automatic program optimization
 - Compilers
 - **Program synthesizers**
- IV. Conclusions

III.2 Program Synthesizers (1)

- Emerging new field.
- Goal is to automatically generate highly efficient code for each target machine.
- Typically, a generator is executed to empirically search the space of possible algorithms/implementations.
- Examples:
 - In linear algebra: ATLAS, PhiPAC
 - In signal processing: FFTW, SPIRAL

III.2 Program Synthesizers (3)

- Automatic generation of libraries
 - Reduce development cost
 - For a fixed cost, enable a wider range of implementations and thus make libraries more usable.
- Advantage over compilers: Can make use of semantics
 - More possibilities can be explored.
- Disadvantage over compilers: Domain specific.

III.2 Program Synthesizers (4)

Three synthesis projects

1. Spiral

Joint project with CMU and Drexel.

M. Püschel, J. Moura, J. Johnson, D. Padua, M. Veloso, B. Singer, J. Xiong, F. Franchetti, A. Gacic, Y. Voronenko, K. Chen, R. W. Johnson, and N. Rizzolo. SPIRAL: Code Generation for DSP Transforms. Proceedings of the IEEE special issue on "Program Generation, Optimization, and Platform Adaptation". Vol. 93, No. 2, pp. 232-275. February 2005.

2. Analytical models for ATLAS

Joint project with Cornell.

K. Yotov, X. Li, G. Ren, M. Garzaran, D. Padua, K. Pingali, P. Stodghill. Is Search Really Necessary to Generate High-Performance BLAS? Proceedings of the IEEE special issue on "Program Generation, Optimization, and Platform Adaptation". Vol. 93, No. 2, pp. 358-386. February 2005.

3. Sorting and adaptation to the input

In all cases results are surprisingly good. Competitive or better than the best manual results.

Special Issue on:

PROGRAM GENERATION, OPTIMIZATION, AND PLATFORM ADAPTATION

Papers on:

Design & Implementation of FFTW3 • SPIRAL: Code Generation for DSP Transforms
• Synthesis of Parallel Programs for *Ab Initio* Quantum Chemistry Models • Self-Adapting
Linear Algebra Algorithms & Software • Parallel VSIP++: An Open Standard for Parallel
Signal Processing • Parallel MATLAB: Doing it Right • Broadway: Exploiting the Domain-
Specific Semantics of Software Libraries • Is Search Really Necessary in Generic High-
Performance BLAS? • Telescoping Languages: Automatic Generation of Domain Languages
• Efficient Utilization of SIMD Extensions • Intelligent Monitoring for Adaptation in Grid
Applications • Design & Engineering of a Dynamic Binary Optimizer
• A Survey of Adaptive Optimization in Virtual Machines

plus:

Scanning Our Past: Electrical Engineering Hall of Fame: Alexander Graham Bell



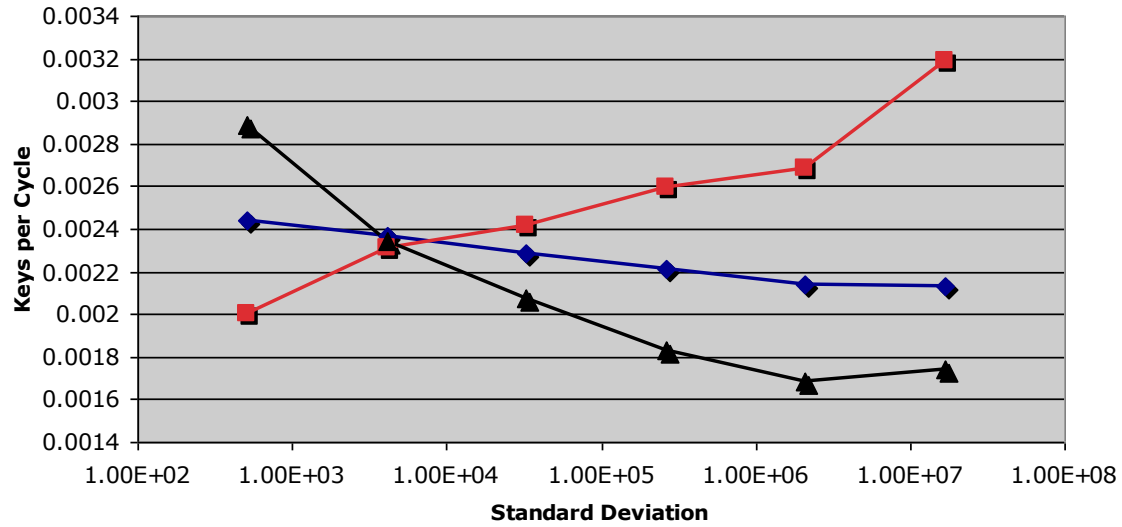
III.2 Program Synthesizers (5)

Sorting routine synthesis

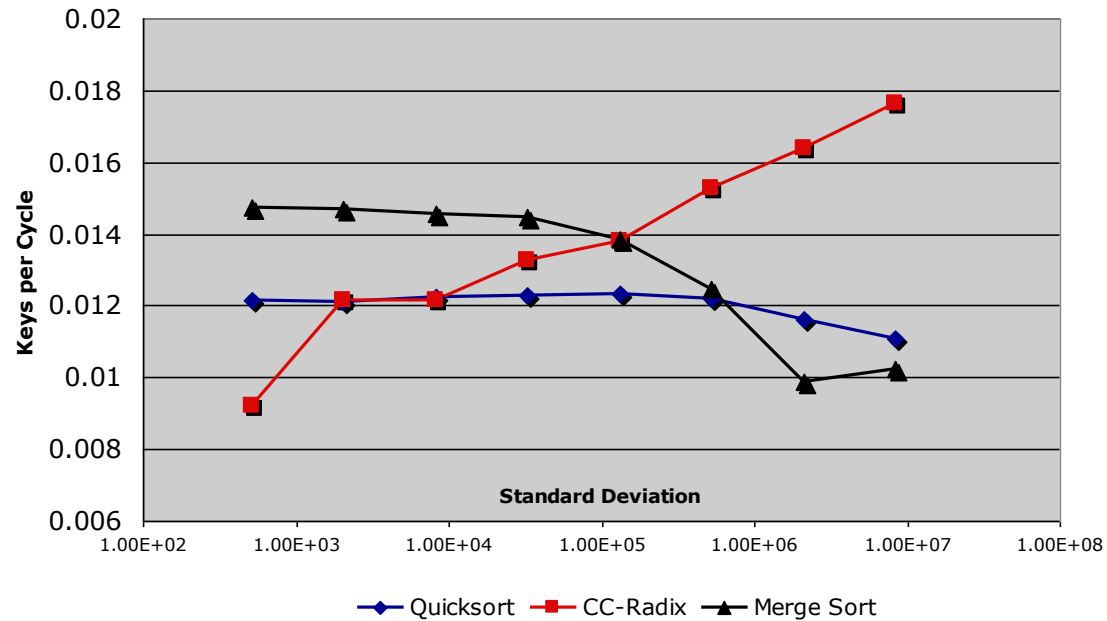
- During training several features are selected influenced by:
 - Architectural features
 - Different from platform to platform
 - Input characteristics
 - Only known at runtime
- Features such as: Radix for sorting, how to sort small segments, when is a segment small.

X. Li, M. Garzarán, and D. Padua. Optimizing Sorting with Genetic Algorithms. CGO2005

Intel Xeon



AMD Athlon MP

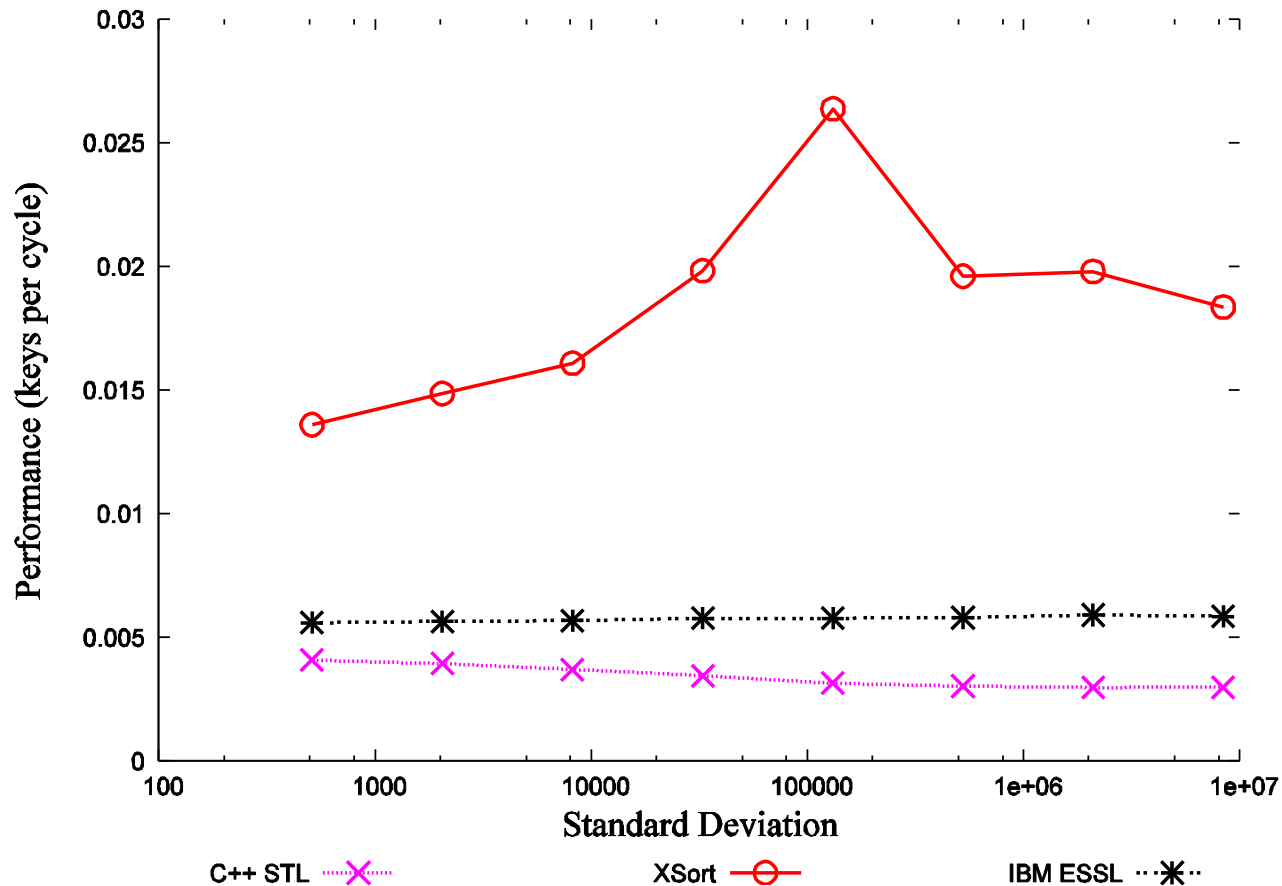


◆ Quicksort ■ CC-Radix ▲ Merge Sort

III.2 Program Synthesizers (6)

Sorting routine synthesis

Performance on Power4



III.2 Program Synthesizers (8)

Programming synthesizers

- Objective is to develop language extensions to implement parameterized programs.
- Values of the parameters are a function of the target machine and execution environment.
- Program synthesizers could be implemented using autotuning extensions.

Sebastien Donadio, James Brodman, Thomas Roeder, Kamen Yotov, Denis Barthou, Albert Cohen, María Jesús Garzarán, David Padua and Keshav Pingali. A Language for the Compact Representation of Multiple Program Versions. In the *Proc. of the International Workshop on Languages and Compilers for Parallel Computing*, October 2005.

III.2 Program Synthesizers (9)

Programming synthesizers

Example extensions.

```
#pragma search (1<=m<=10, a)
#pragma unroll m
    for (i=1; i<n; i++) { ... }
    %if (a) then { algorithm 1 }
                else { algorithm 2 }
```

IV. Conclusions

- Advances in languages and automatic optimization will probably be slow. Difficult problem.
- Advent of parallelism → Decrease in productivity. Higher costs.
- But progress must and will be made.
- Automatic optimization (including parallelization) is a difficult problem. At the same time is a core of computer science:

How much can we automate ?