Compilation Techniques for Automatic Extraction of Parallelism and Locality in Heterogeneous Architectures

José M. Andión

PhD Advisors: Gabriel Rodríguez and Manuel Arenaz

COMPUTER ARCHITECTURE GROUP
UNIVERSITY OF A CORUÑA
GAC.UDC.ES

UNIVERSIDADE DA CORUÑA

Outline

1. Introduction
2. A Novel Compiler Support for Multicore Systems
3. Locality-Aware Automatic Parallelization for GPGPU
4. Trace-Based Affine Reconstruction of Code
5. Conclusions

Outline

1. **Introduction**
2. A Novel Compiler Support for Multicore Systems
3. Locality-Aware Automatic Parallelization for GPGPU
4. Trace-Based Affine Reconstruction of Code
5. Conclusions

"HPC is a crucial asset for Europe's innovation capacity and is of strategic importance to its scientific and industrial capabilities, as well as to its citizens"

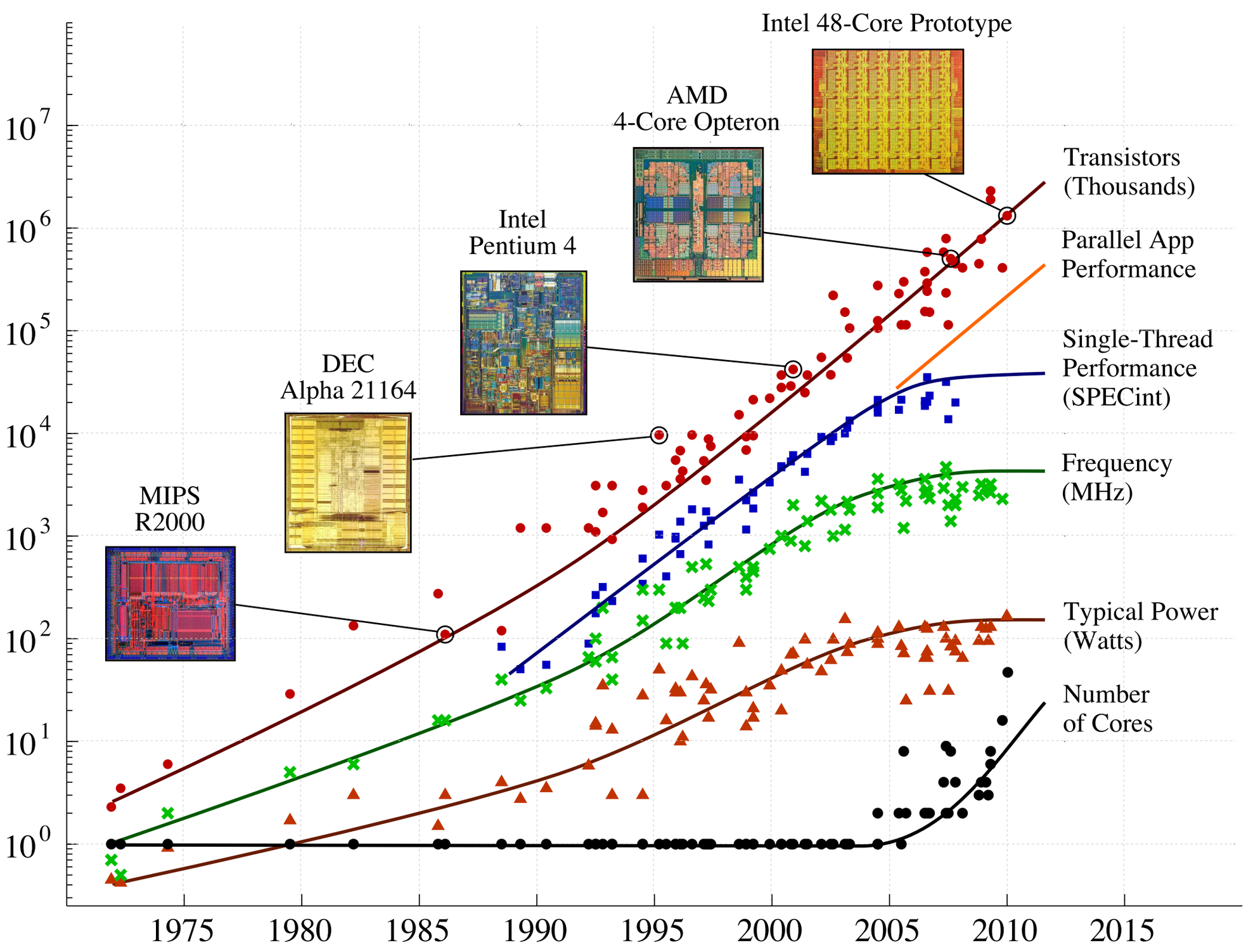
– H2020 Work Programme

"HPC has contributed substantially to national economic prosperity and rapidly accelerated scientific discovery"

– POTUS Executive Order for Creating a National Strategic Computing Initiative

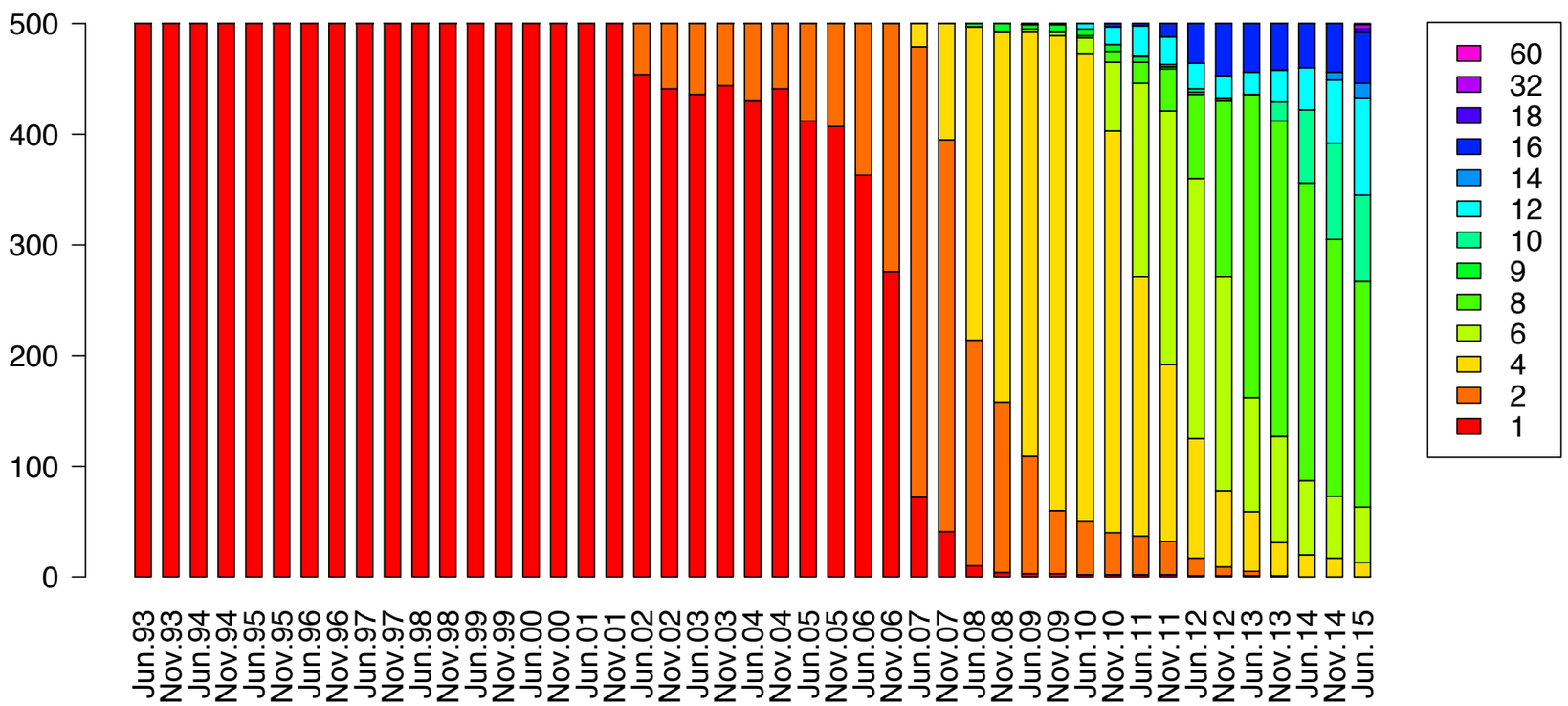
"To out-compute is to out-compete"

– US Council of Competitiveness



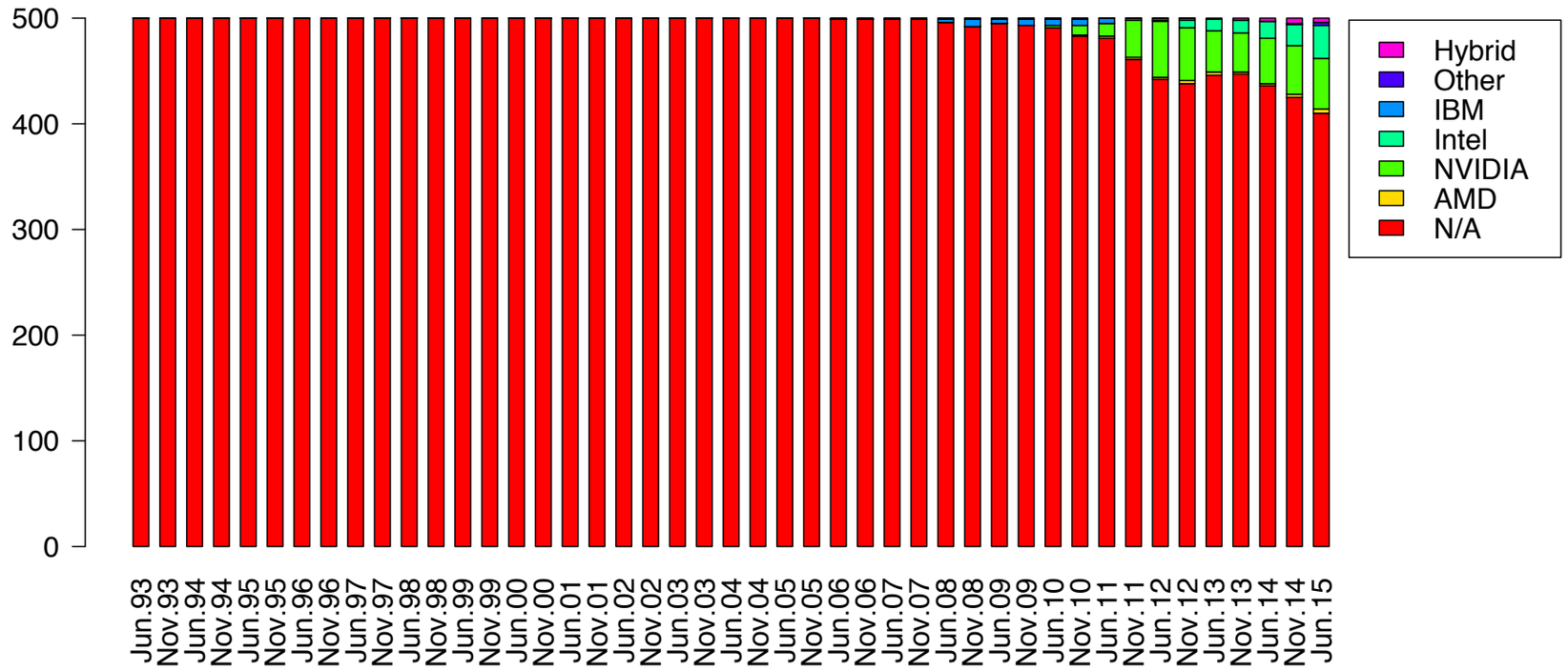
Trends in Transistors, Performance, and Power for General-Purpose Processors

C. F. Batten. Simplified vector-thread architectures for flexible and efficient data-parallel accelerators. PhD thesis, Massachusetts Institute of Technology, Department of Electrical Engineering and Computer Science, 2010.



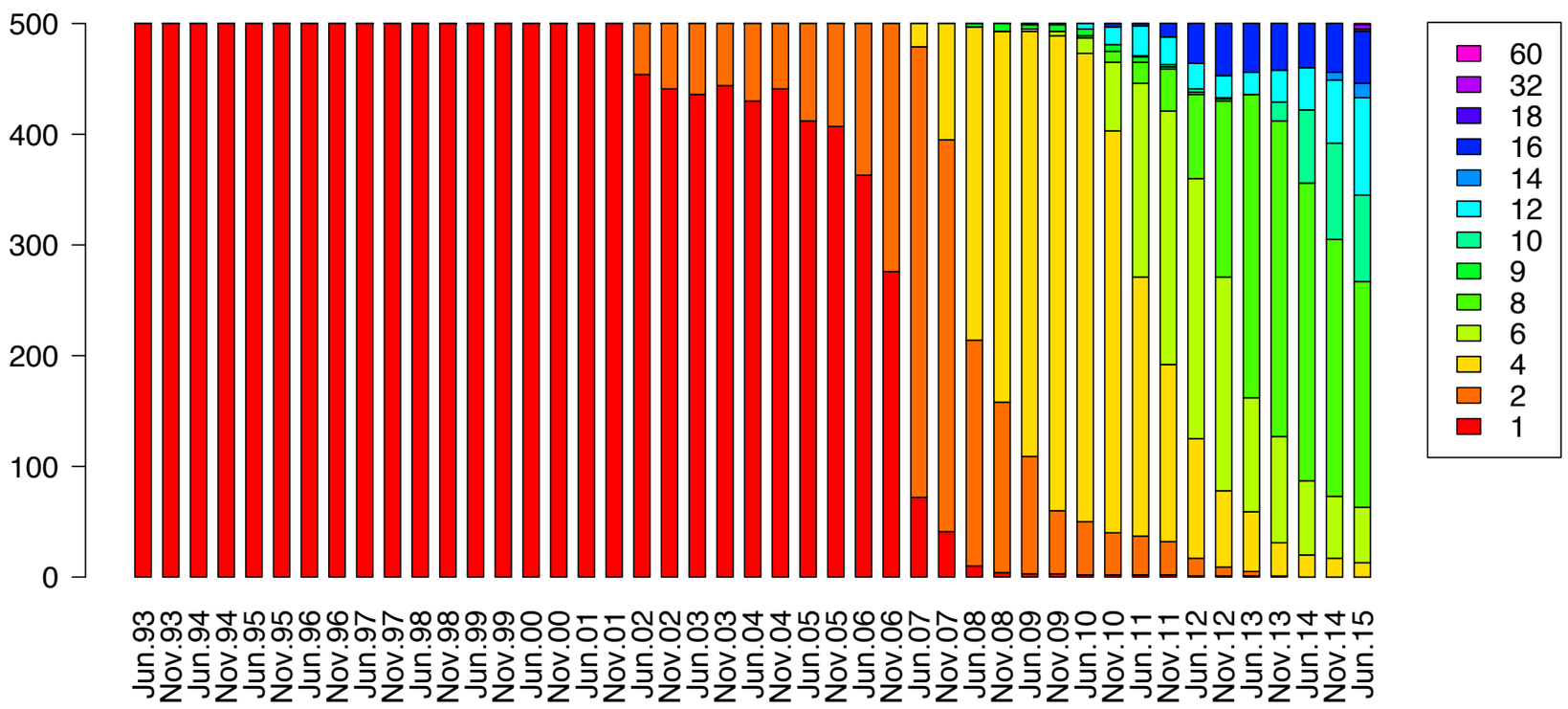
Cores per Socket

Accelerators



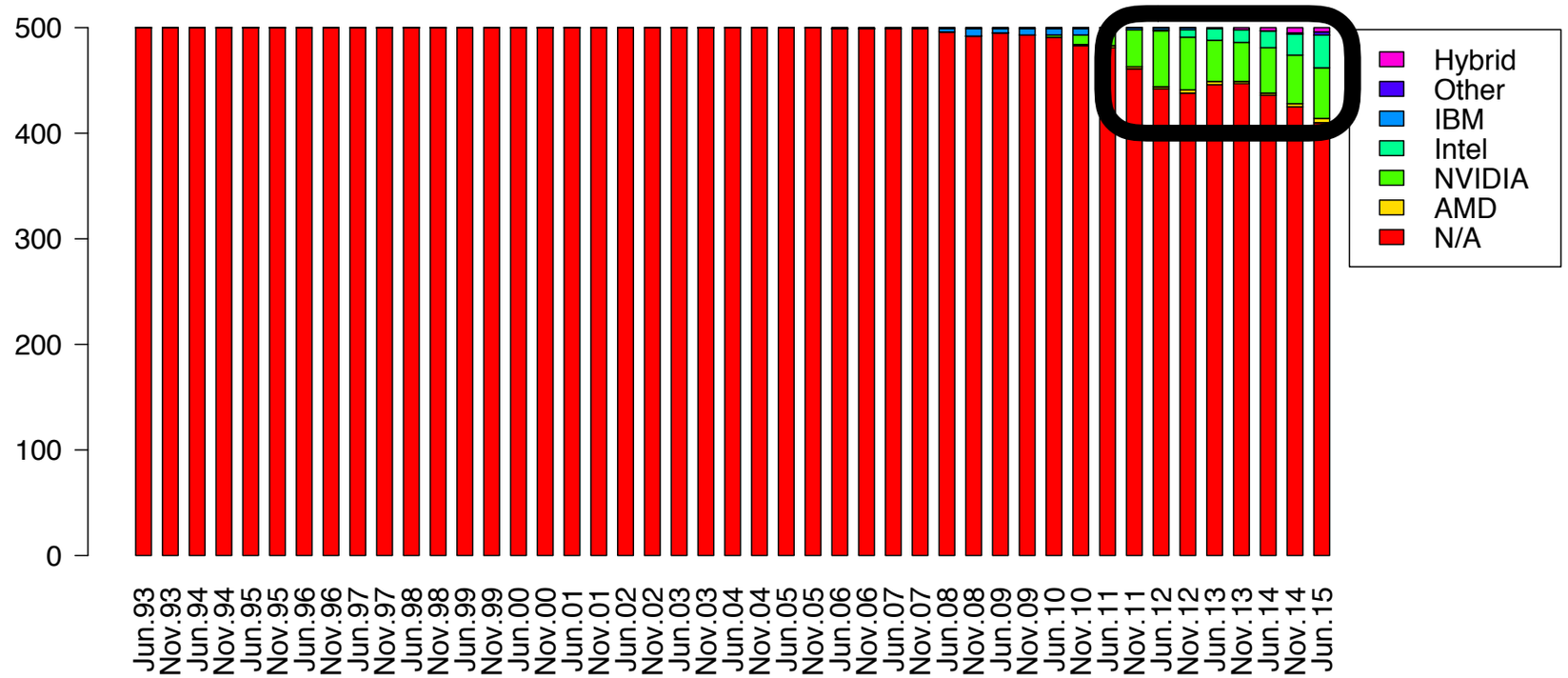
The TOP500 List

Development over Time



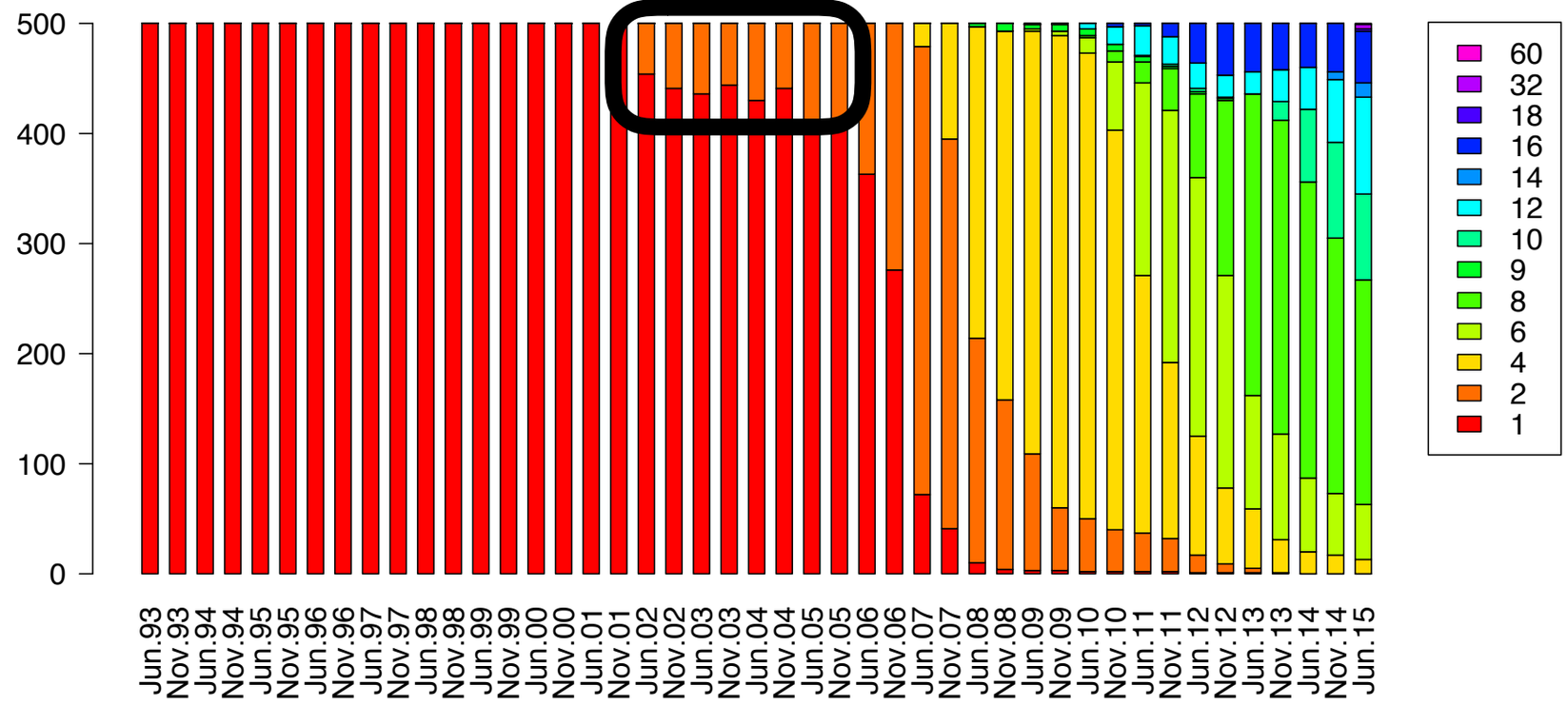
Cores per Socket

Accelerators



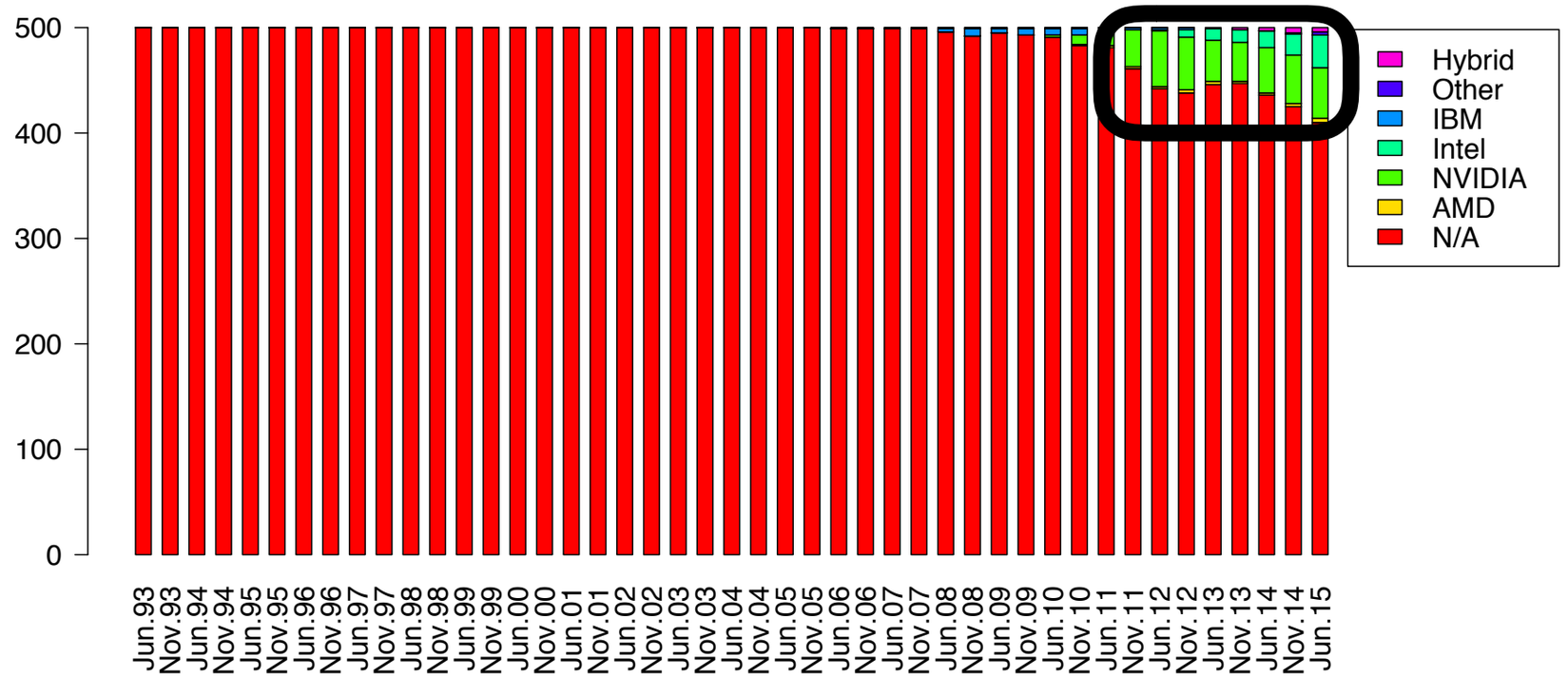
The TOP500 List

Development over Time



Cores per Socket

Accelerators



The TOP500 List

Development over Time

A New “Software Crisis”

- Modeling reality in software is already difficult
- Hardware architecture with multiple levels of increasing complexity
- How do we distribute computations?
- How do we distribute data?

Extraction of Parallelism

Productivity

\- +

- libraries

- compiler directives

- programming languages

- parallelizing compilers

Extraction of Parallelism

- libraries

- compiler directives

Our Proposal: Source-to-Source Parallelizing Compiler for CPUs and GPUs

- parallelizing compilers

+

Extraction of Locality

- The Locality Principle: temporal & spatial clustering
- Techniques to improve locality:
  - loop interchange, fission and fusion of loops and arrays, tiling…
  - hardware and software prefetching
  - data placement
  - design of ad-hoc memory systems
- Implemented in compiler frameworks

Extraction of Locality

- The Locality Principle: temporal & spatial clustering
- Techniques to improve locality:
  - loop interchange, fission and fusion of loops and arrays, tiling

Our Proposal: Affine Reconstruction of Code from a Trace of Memory Accesses

- design of ad-hoc memory systems
- Implemented in compiler frameworks

Outline

1. Introduction
2. **A Novel Compiler Support for Multicore Systems**
3. Locality-Aware Automatic Parallelization for GPGPU
4. Trace-Based Affine Reconstruction of Code
5. Conclusions

Outline

2. A Novel Compiler Support for Multicore Systems
   - KIR: A diKernel-based IR
   - Automatic Partitioning driven by the KIR
   - Automatic Parallelization of the Benchmark Suite
   - Experimental Evaluation

Outline

2. A Novel Compiler Support for Multicore Systems

   - **KIR: A diKernel-based IR**

   - Automatic Partitioning driven by the KIR

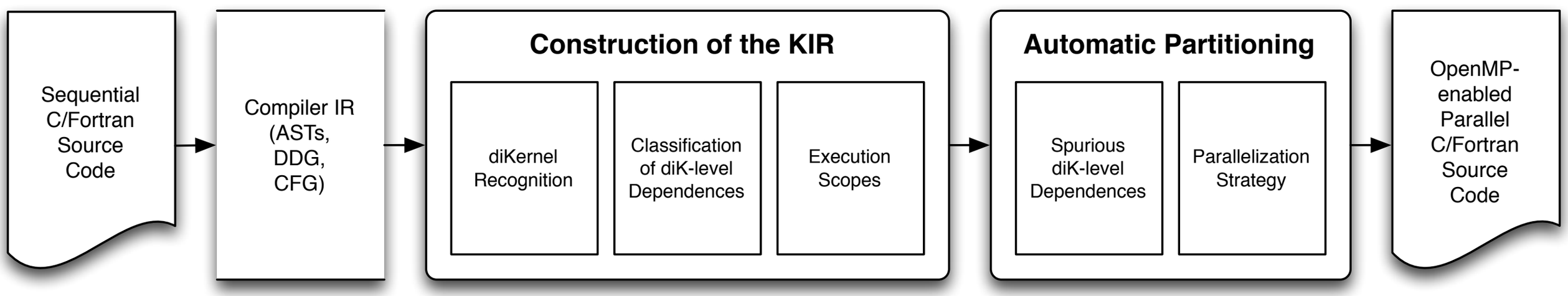
   - Automatic Parallelization of the Benchmark Suite

   - Experimental Evaluation

State-of-the-art vs. Our Approach

- Current parallelizing compilers are based on systems of equations that respect all dependences of statement-based IRs even if they are merely implementation artifacts

- Our approach:

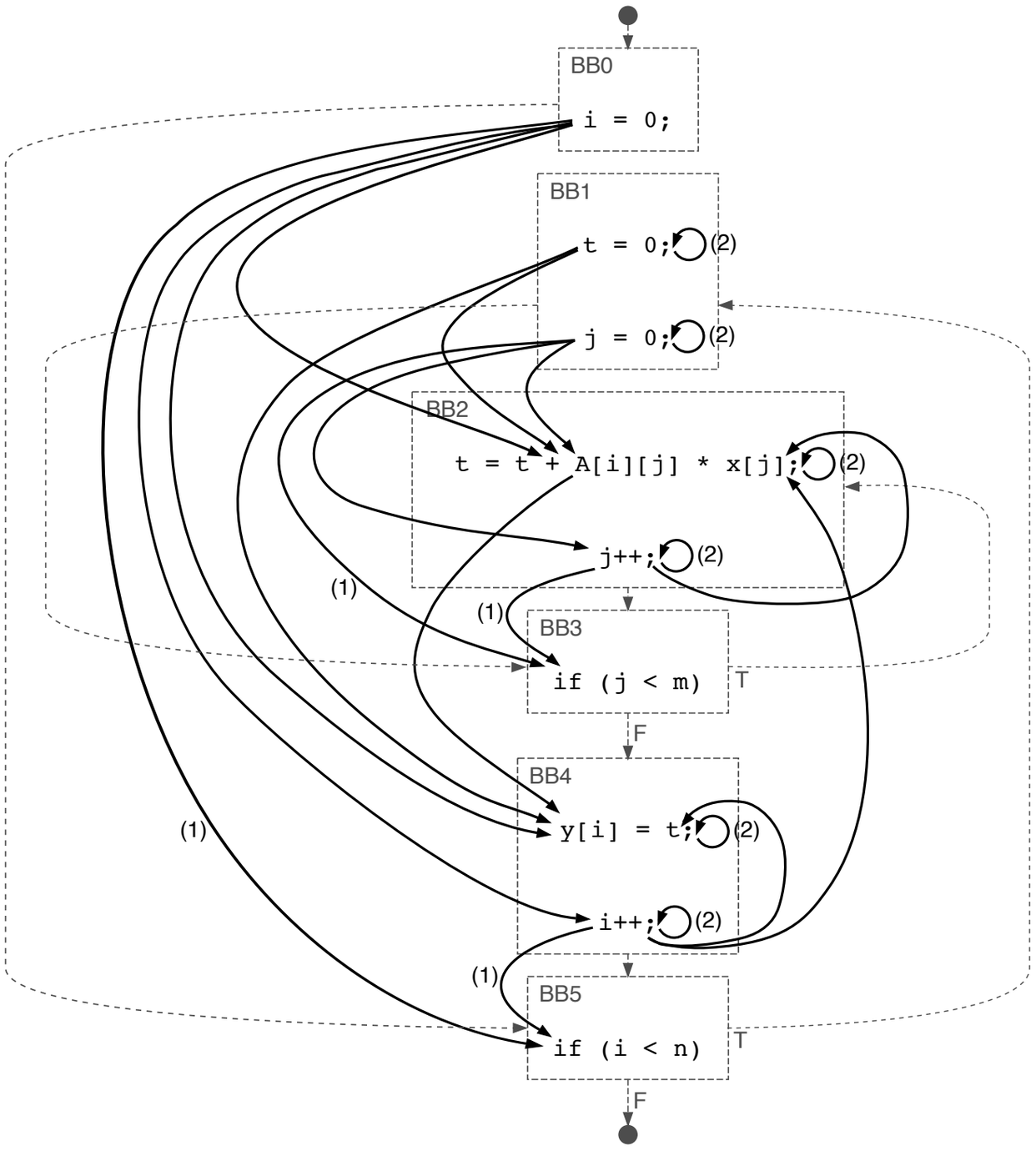


Standard Statement-based IR

BB0, BB5 & BB4

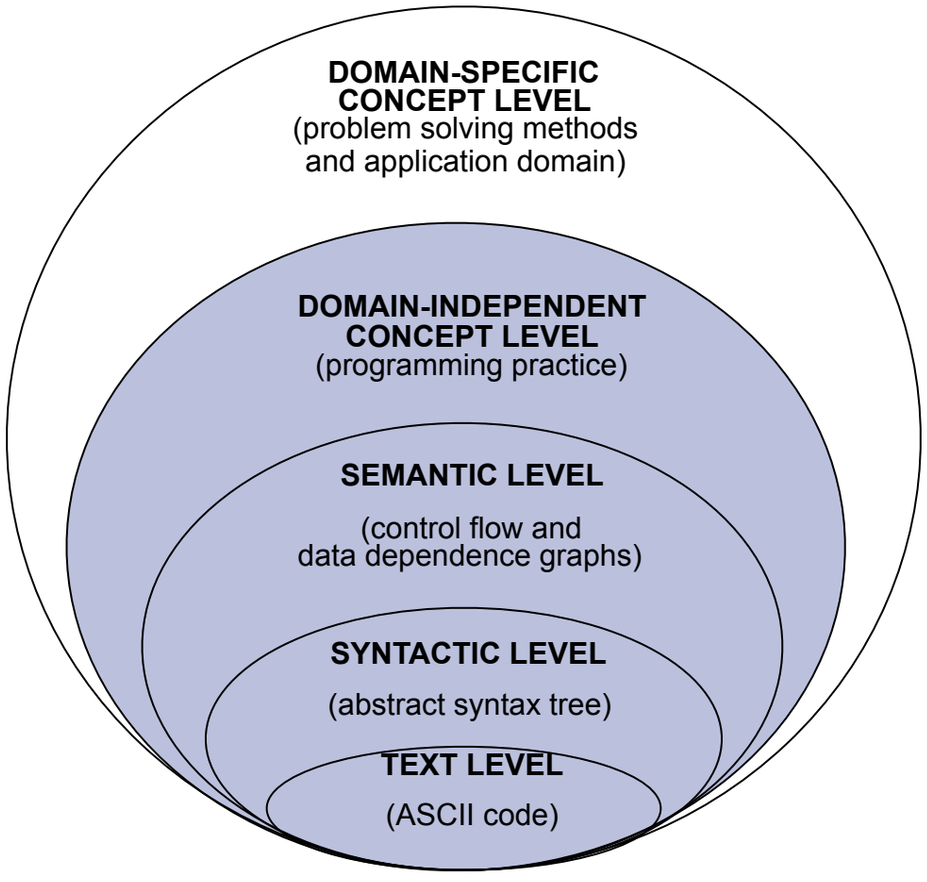
```
1 for (i = 0; i < n; i++) {
2   t = 0;
3   for (j = 0; j < m; j++) {
4     t = t + A[i][j] * x[j];
5   }
6   y[i] = t;
7 }
```

BB1, BB3 & BB2



diKernel: Domain-Independent Computational Kernel

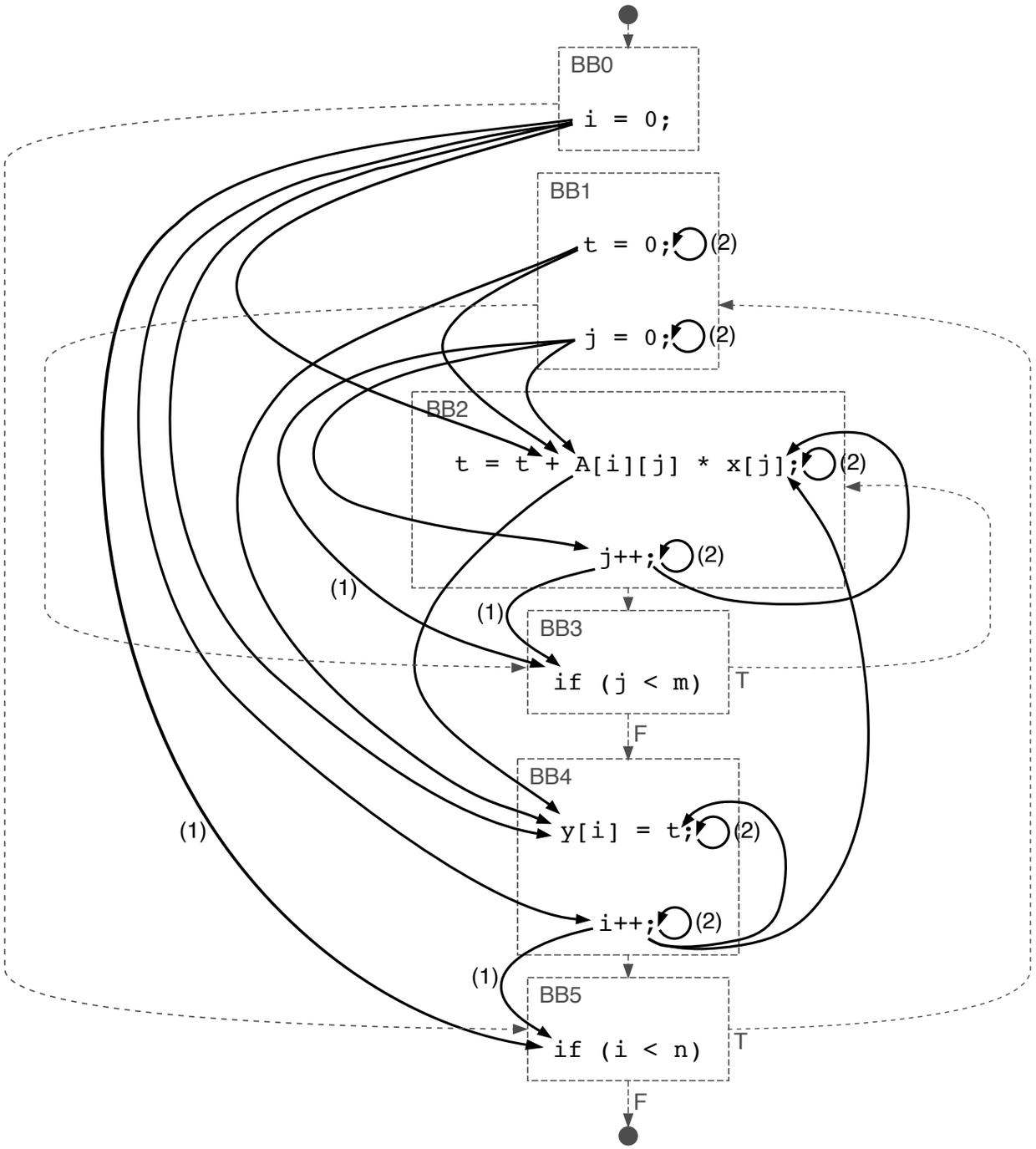
- Characterizes the computations carried out in a program without being affected by how they are coded

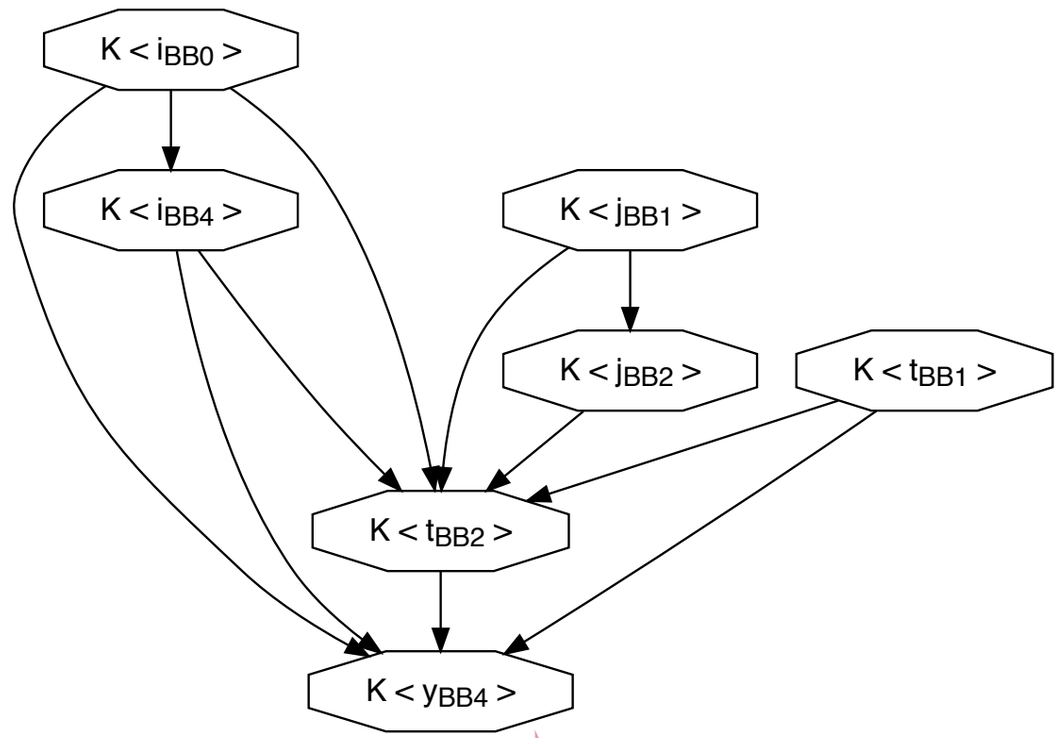


- SCC of the DDG ignoring flow-of-control statements

Building the KIR (I)

```
1 | for (i = 0; i < n; i++) {
2 |   t = 0;
3 |   for (j = 0; j < m; j++) {
4 |     t = t + A[i][j] * x[j];
5 |   }
6 |   y[i] = t;
7 | }
```





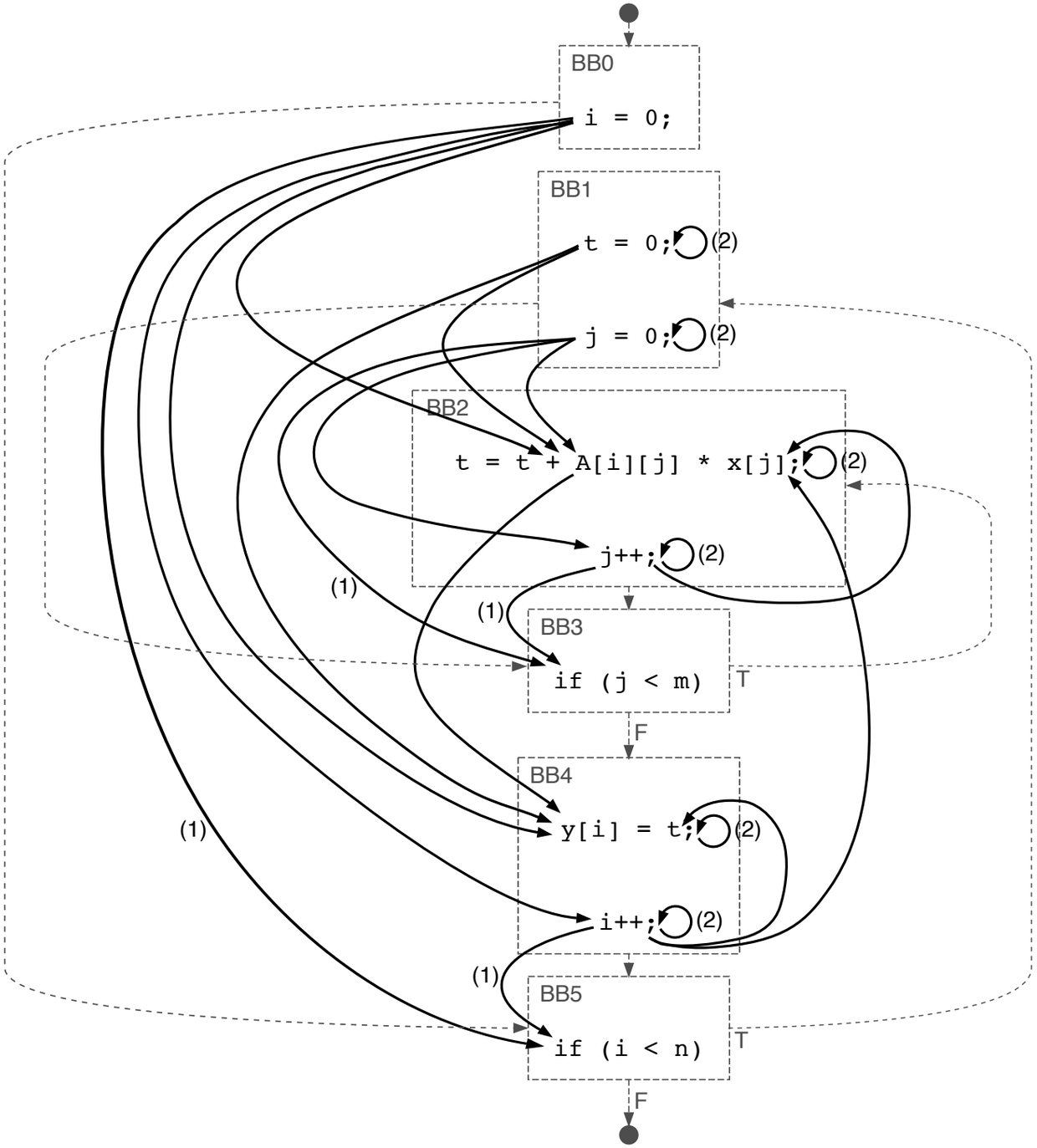
Edges (1), (2) are abstracted with diKernels

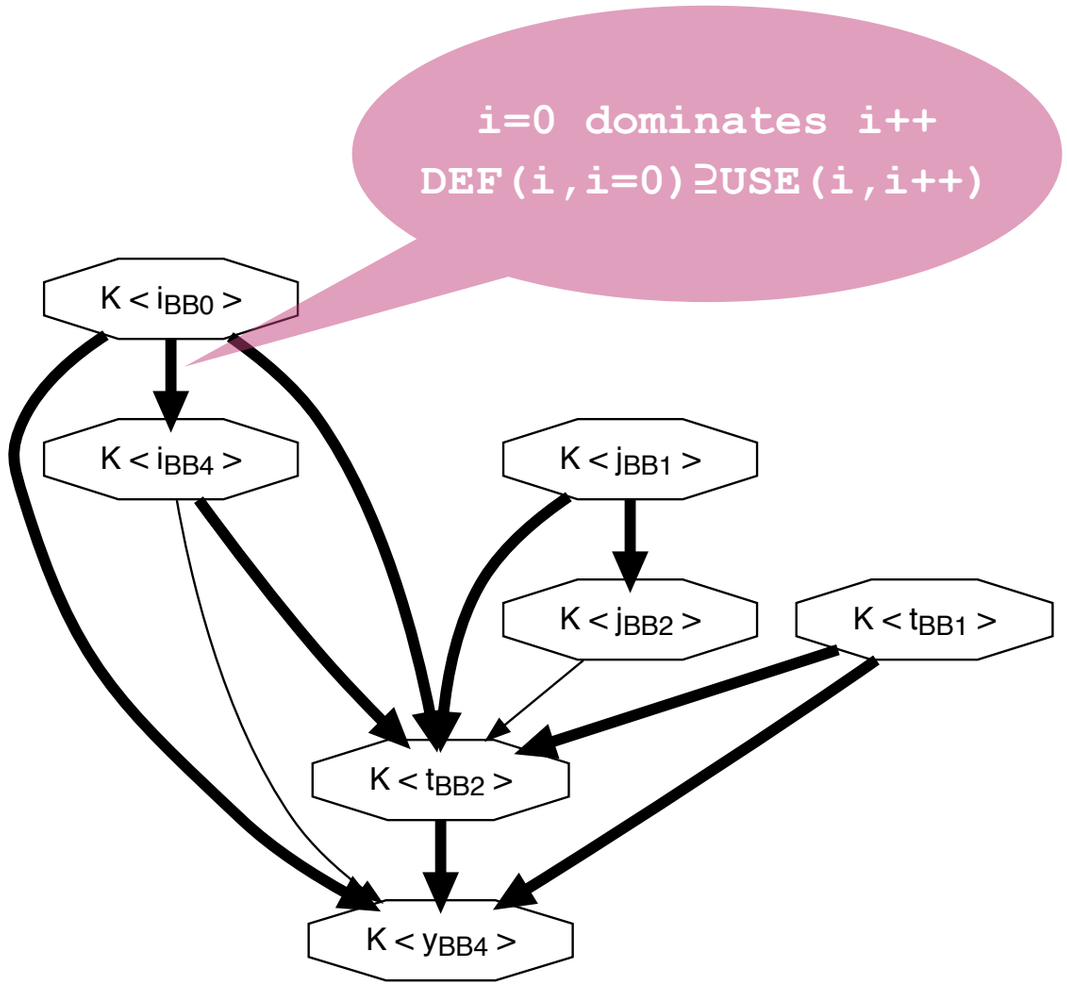
diKernel-level Flow Dependences

- Identification of the flow of information across the program
  - Statement-level dominance
    - Range of values of variable *x* produced and used throughout the execution of statements

$$DEF(x, x_i) \supseteq USE(x, y_j)$$

Building the KIR (II)



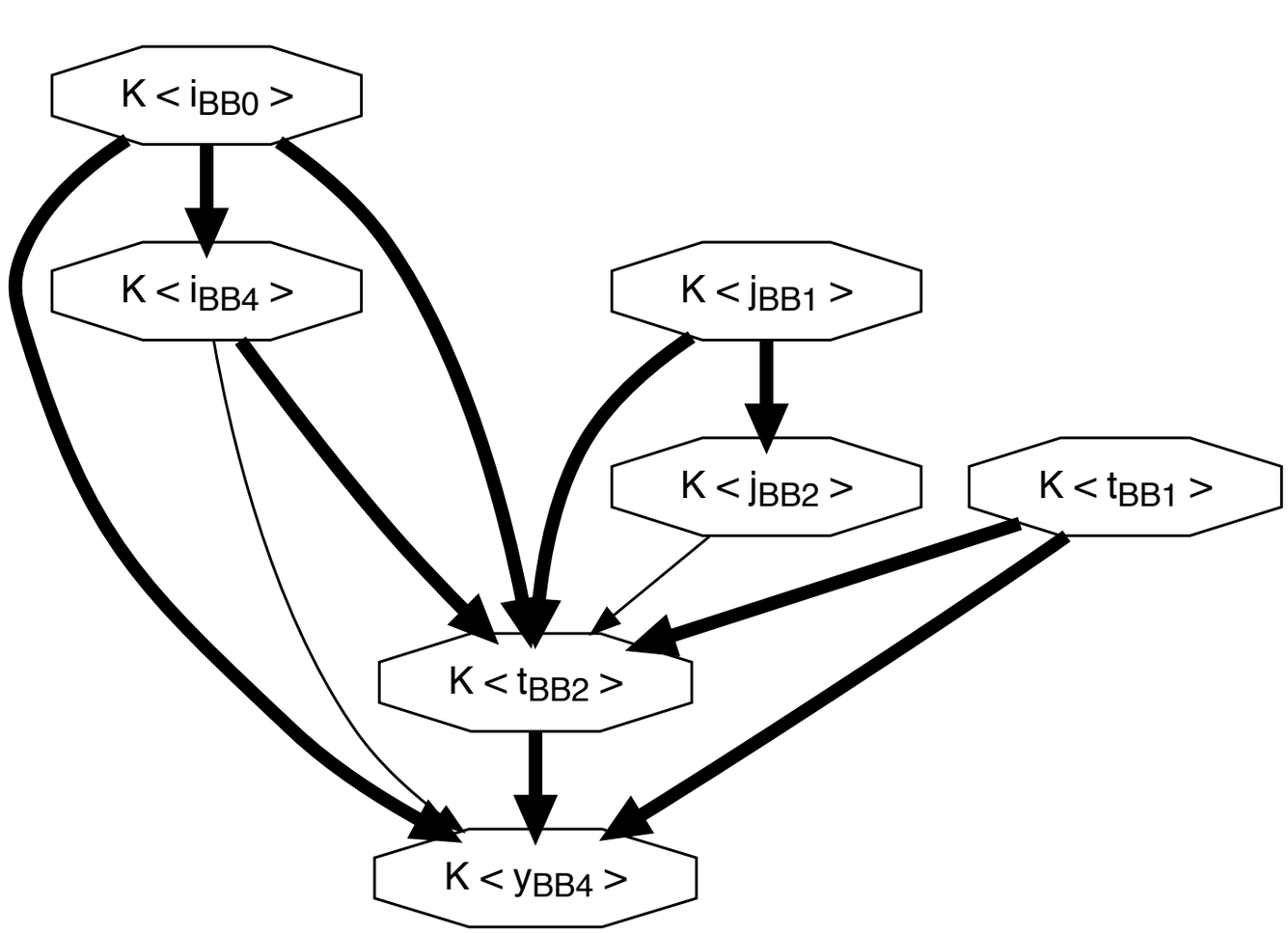


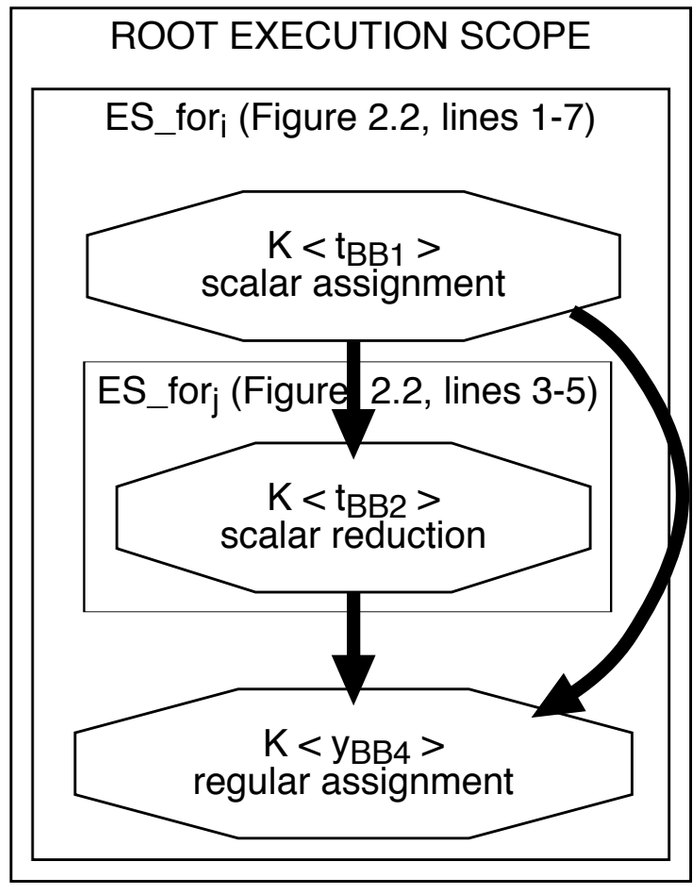
Hierarchy of Execution Scopes

- To expose the computational stages of the program
- Based on the hierarchy of loops: one execution scope for each perfect loop nest
- The root execution scope is a special node that represents the program as a whole.
- diKernels belong to the innermost execution scope that contains all of their statements
  - diKernels that compute the loop indices belong to the ES of the corresponding loop

Building the KIR (and III)

```
1 for (i = 0; i < n; i++) {
2   t = 0;
3   for (j = 0; j < m; j++) {
4     t = t + A[i][j] * x[j];
5   }
6   y[i] = t;
7 }
```





Outline

2. A Novel Compiler Support for Multicore Systems

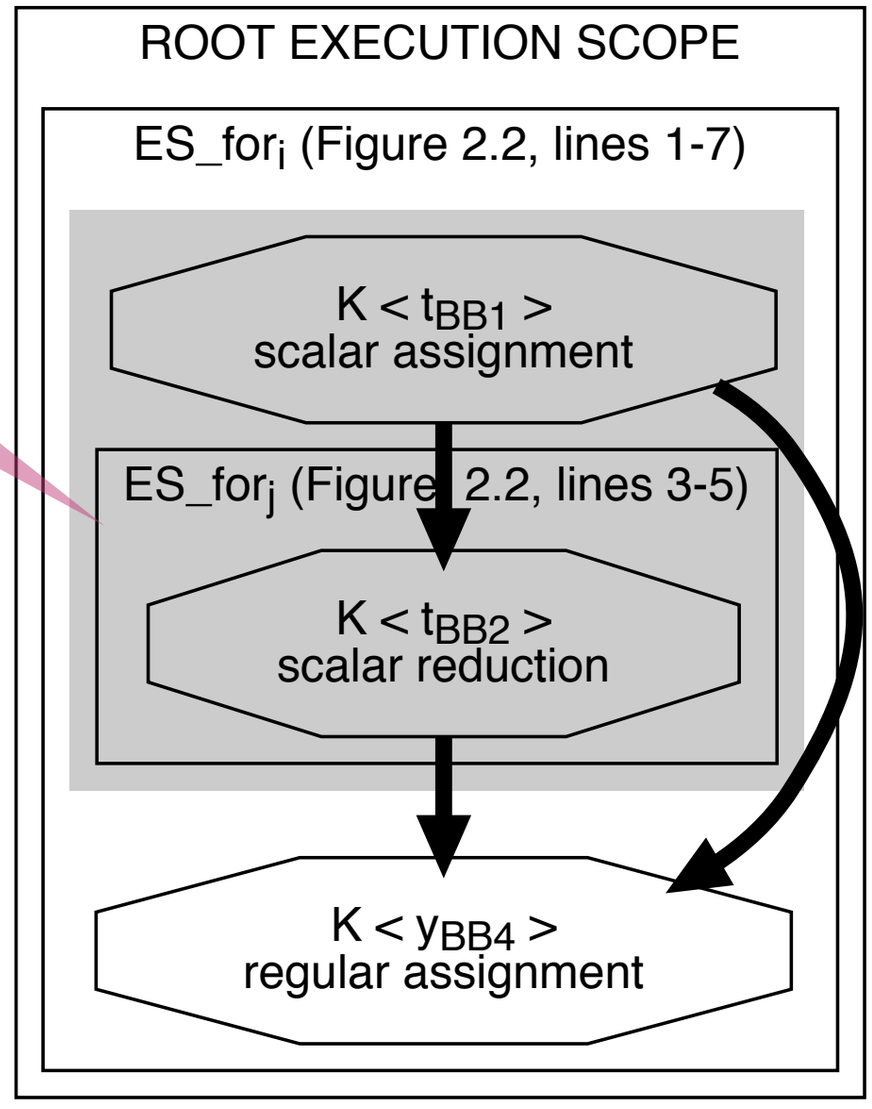
   - KIR: A diKernel-based IR
   - **Automatic Partitioning driven by the KIR**
   - Automatic Parallelization of the Benchmark Suite
   - Experimental Evaluation

Spurious diKernel-level Dependences

- They do not prevent the parallelization

t is a privatizable scalar variable

```
1 | for (i = 0; i < n; i++) {
2 |   t = 0;
3 |   for (j = 0; j < m; j++) {
4 |     t = t + A[i][j] * x[j];
5 |   }
6 |   y[i] = t;
7 | }
```



OpenMP-enabled Parallelization Strategy

- Find the critical path in the KIR
  - diKernel-level flow dependences
  - Parallelizing transformations for each type of diKernel
- Optimizations for the joint parallelization of loops
  - Minimize synchronization between diKernels
  - Minimize thread creation/destruction

Parallelizing Transformations

FULLY PARALLEL LOOP

```
1. #pragma omp parallel for
2. for (i = 0; i < n; i++) {
3.   A[i] = 2
4. }
```

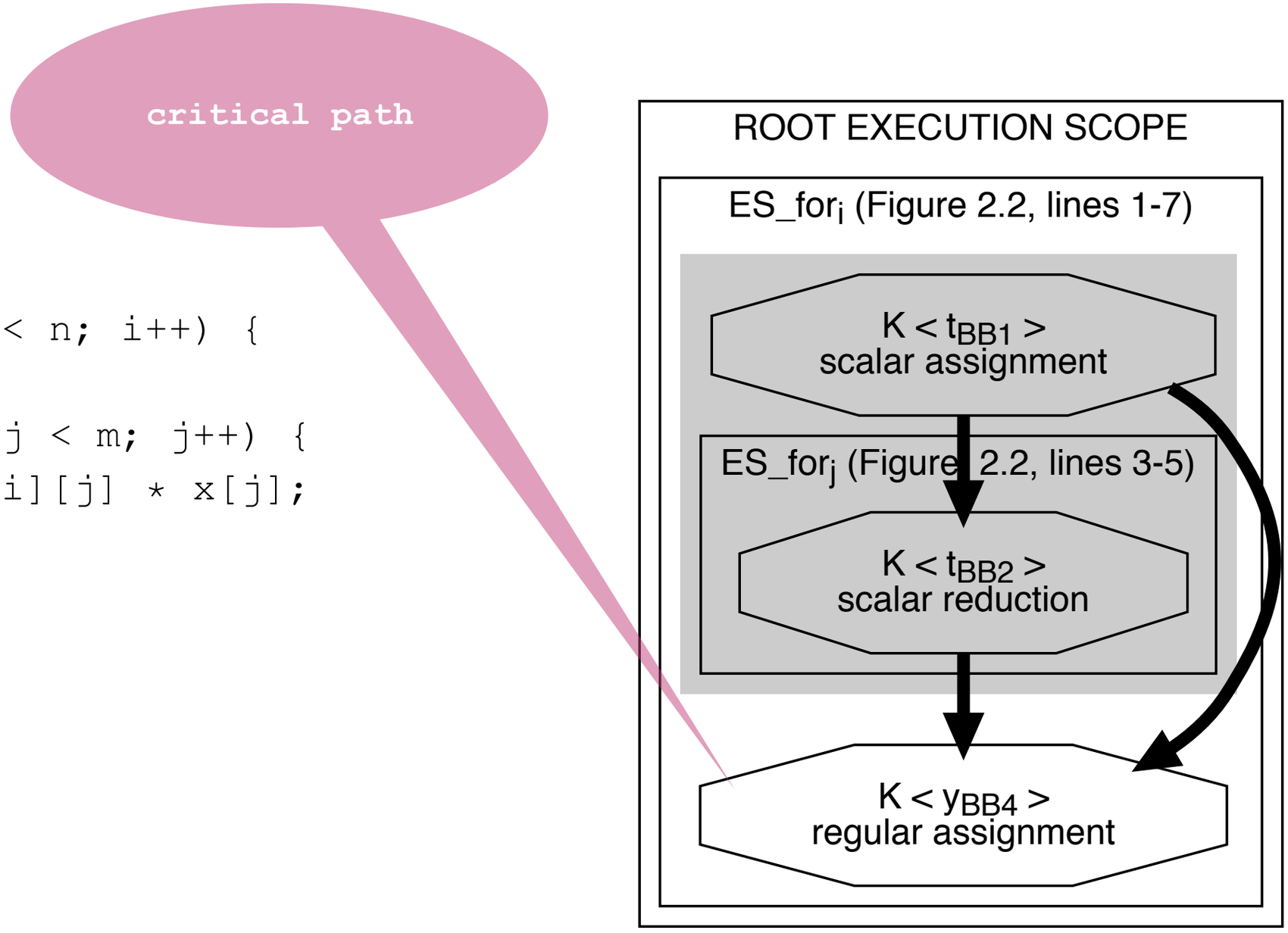
PARTIALLY PARALLEL LOOP

```
1. r = 0;
2. #pragma omp parallel for reduction(+:r)
3. for (i = 0; i < n; i++) {
4.   r = r + A[i];
5. }
```

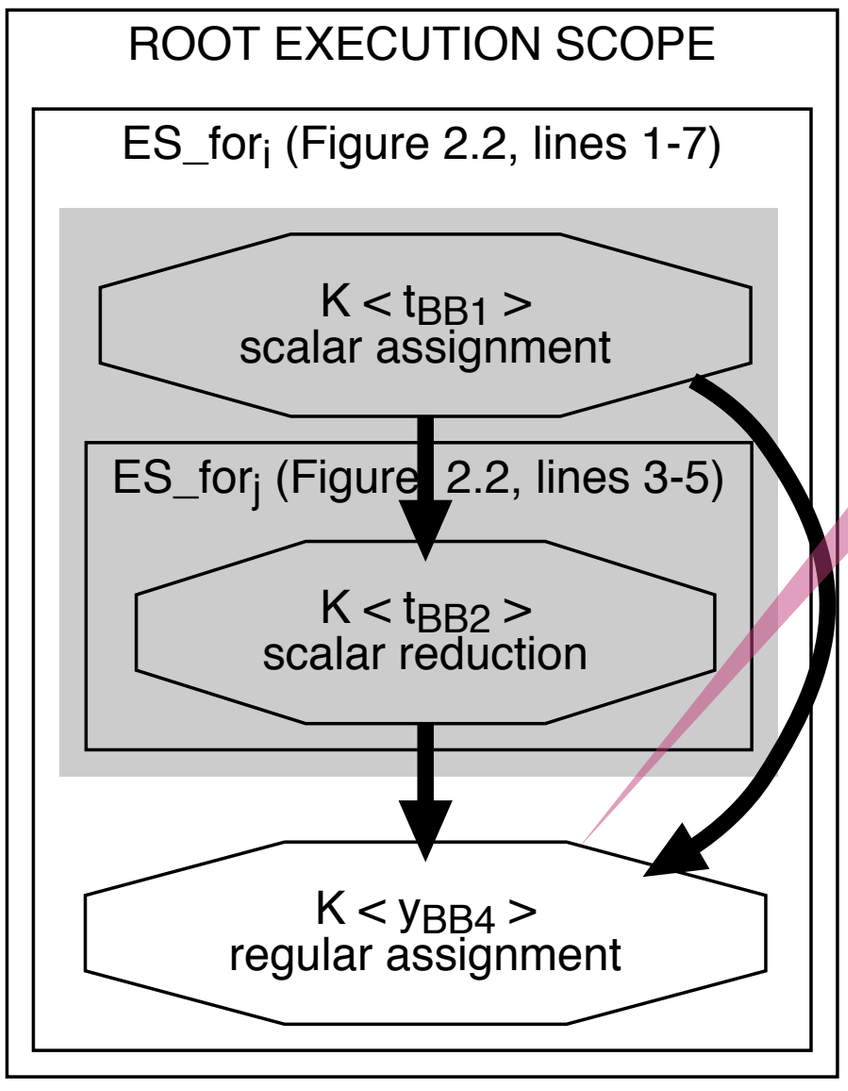
Array Expansion

Automatic Partitioning driven by the KIR (I)

```
1 for (i = 0; i < n; i++) {
2   t = 0;
3   for (j = 0; j < m; j++) {
4     t = t + A[i][j] * x[j];
5   }
6   y[i] = t;
7 }
```



Automatic Partitioning driven by the KIR (and II)



critical path

FULLY PARALLEL LOOP

```
1  #pragma omp parallel shared(A,x,y) private(i,j,t)
2  {
3  #pragma omp for schedule(static)
4    for (i = 0; i < n; i = i + 1) {
5      t = 0;
6      for (j = 0; j < m; j = j + 1) {
7        t = (t) + ((A[i][j]) * (x[j]));
8      }
9      y[i] = t;
10   }
11 }
```

Outline

2. A Novel Compiler Support for Multicore Systems

   - KIR: A diKernel-based IR
   - Automatic Partitioning driven by the KIR
   - **Automatic Parallelization of the Benchmark Suite**
   - Experimental Evaluation

Automatic Parallelization of the Benchmark Suite

- Synthetic Benchmarks
- Dense/Sparse Matrix-Vector Multiplication
- Sobel Edge Filter
- SWIM from SPEC CPU2000
- EQUAKE from SPEC CPU2000

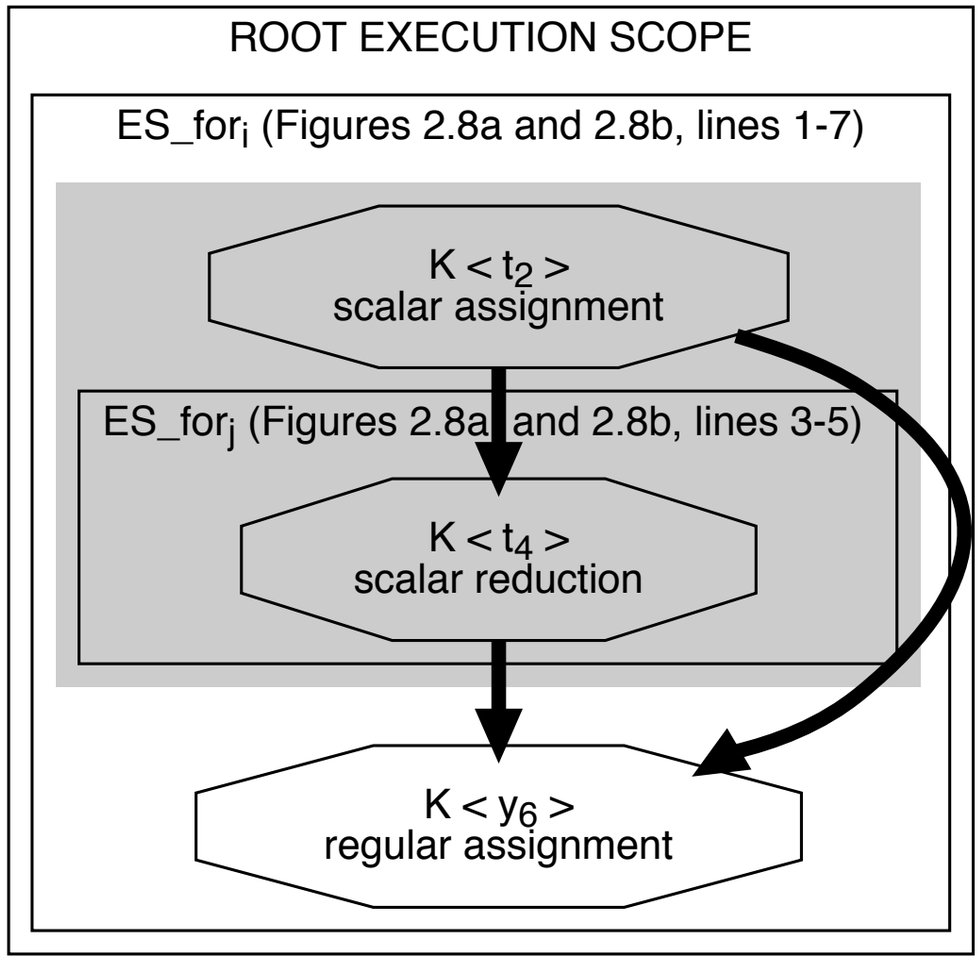
DenseAMUX & SparseAMUX

DenseAMUX

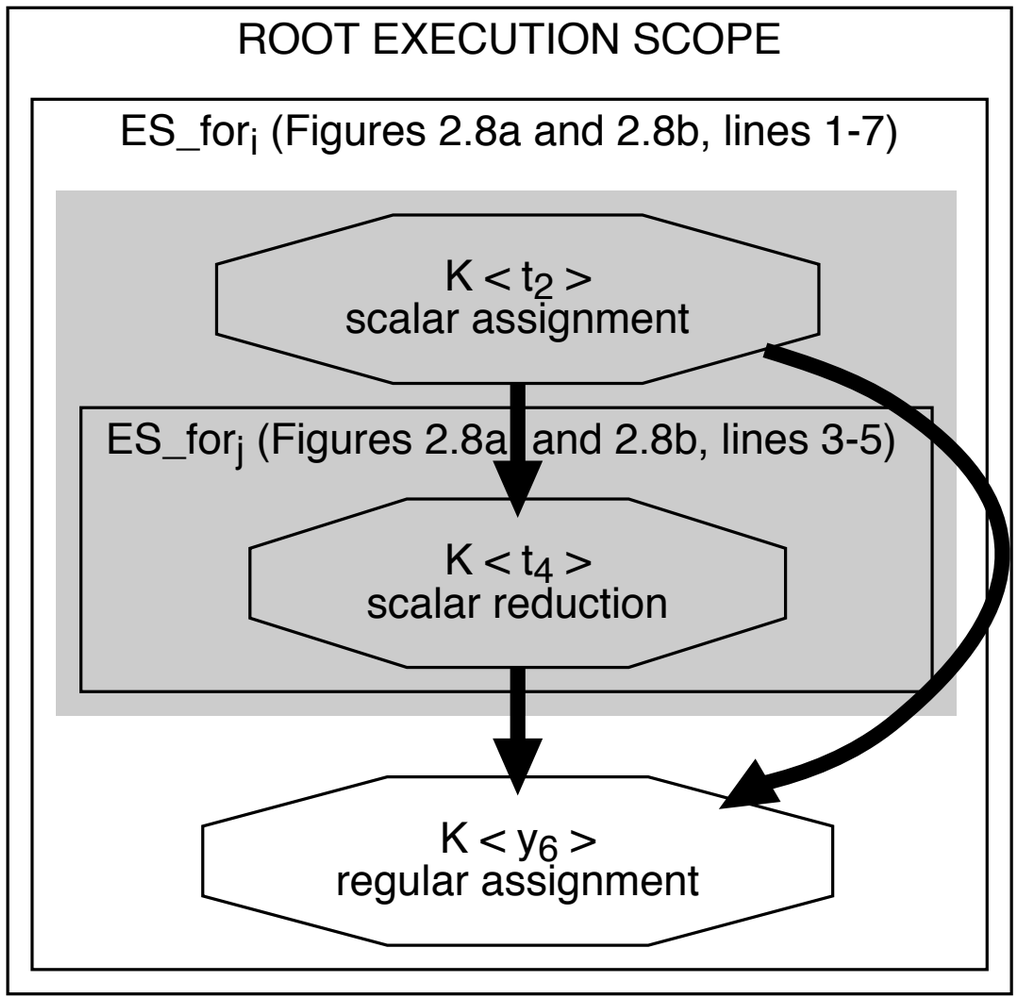
```
1 for (i = 0; i < n; i++) {
2   t = 0;
3   for (j = 0; j < m; j++) {
4     t = t + A[i][j] * x[j];
5   }
6   y[i] = t;
7 }
```

SparseAMUX

```
1 for (i = 0; i < n; i++) {
2   t = 0;
3   for (j = ia[i]; j < ia[i+1]-1; j++) {
4     t = t + A[j] * x[ja[j]];
5   }
6   y[i] = t;
7 }
```



SparseAMUX



FULLY PARALLEL LOOP

```
1  #pragma omp parallel shared(A,ia,ja,x,y) private(i,j,t)
2  {
3  #pragma omp for schedule(static)
4    for (i = 0; i < n; i++) {
5      t = 0;
6      for (j = ia[i]; j < (ia[i+1] - 1); j = j + 1) {
7        t = (t) + ((A[j]) * (x[ja[j]]));
8      }
9      y[i] = t;
0    }
1  }
```

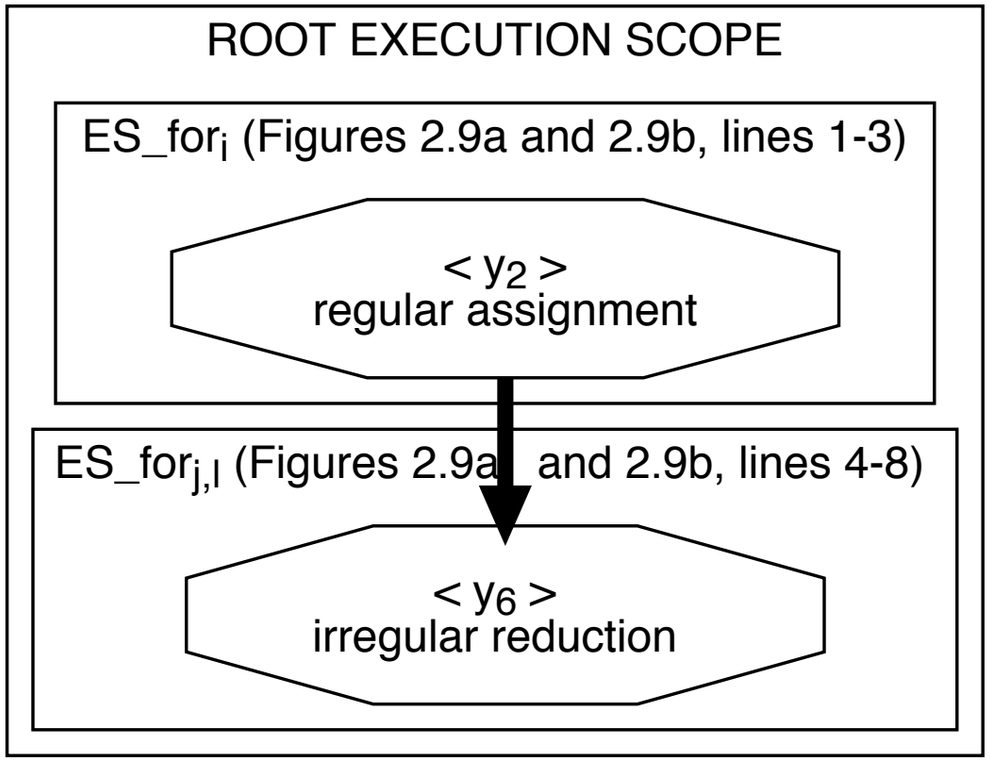
AMUXMS & ATMUX

AMUXMS

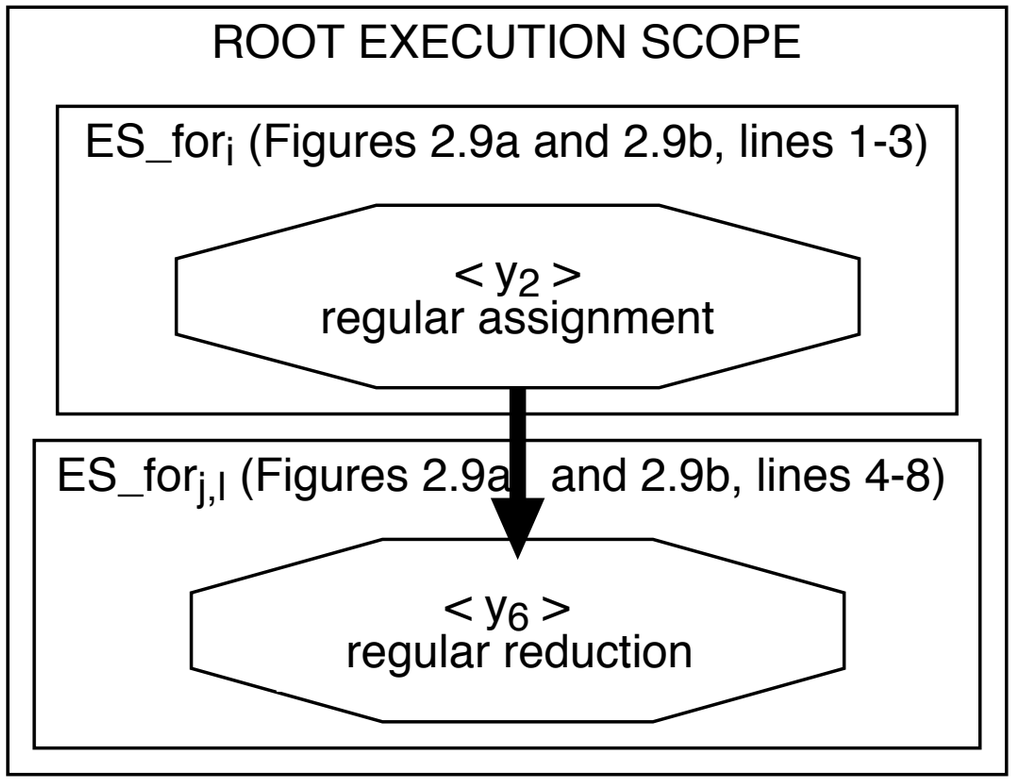
```
1 for (i = 0; i < n; i++) {
2   y[i] = A[i] * x[i];
3 }
4 for (j = 0; j < n; j++) {
5   for (l = ja[j]; l < ja[j+1]-1; l++) {
6     y[j] = y[j] + A[l] * x[ja[l]];
7   }
8 }
```

ATMUX

```
1 for (i = 0; i < n; i++) {
2   y[i] = 0;
3 }
4 for (j = 0; j < n; j++) {
5   for (l = ia[j]; l < ia[j+1]-1; l++) {
6     y[ja[l]] = y[ja[l]] + x[j] * A[l];
7   }
8 }
```



AMUXMS

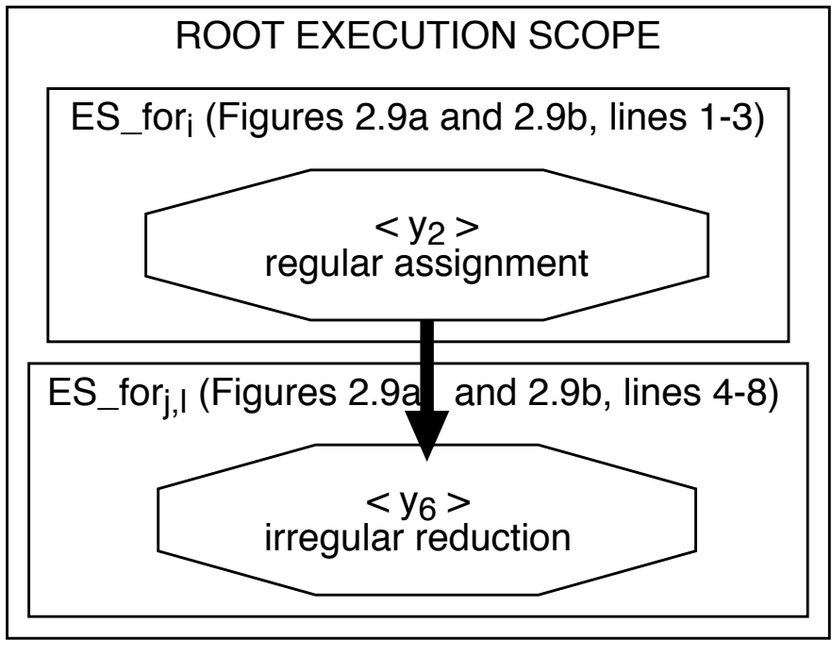


FULLY PARALLEL LOOP

```
1  #pragma omp parallel shared(A,x,ja,y) private(i,j,l,t)
2  {
3  #pragma omp for schedule(static) nowait
4    for (i = 0; i < n; i = i + 1) {
5      y[i] = (A[i]) * (x[i]);
6    }
7  #pragma omp for schedule(static)
8    for (j = 0; j < n; j = j + 1) {
9      for (l = ja[j]; l < (ja[j+1] - 1); l = l + 1) {
10       y[j] = (y[j]) + ((A[l]) * (x[ja[l]]));
11     }
12   }
13 }
```

FULLY PARALLEL LOOP

ATMUX



PARTIALLY PARALLEL LOOP

```
1  #pragma omp parallel shared(A,ia,ja,x,y) private(i,j,l,y___private)
2  {
3    if (omp_get_thread_num() == 0) {
4      y___private = y;
5    } else {
6      y___private = (float *) malloc(n * sizeof(float));
7    }
8    for (i = 0; i < n; i = i + 1) {
9      y___private[i] = 0;
10   }
11 #pragma omp for schedule(static)
12   for (j = 0; j < n; j = j + 1) {
13     for (l = ia[j]; l < (ia[j+1] - 1); l = l + 1) {
14       y___private[ja[l]] = (y___private[ja[l]]) + ((x[j]) * (A[l]));
15     }
16   }
17 #pragma omp critical
18   if (omp_get_thread_num() != 0) {
19     for (i = 0; i < n; i = i + 1) {
20       y[i] += y___private[i];
21     }
22   }
23   if (omp_get_thread_num() != 0) {
24     free(y___private);
25   }
26 }
```

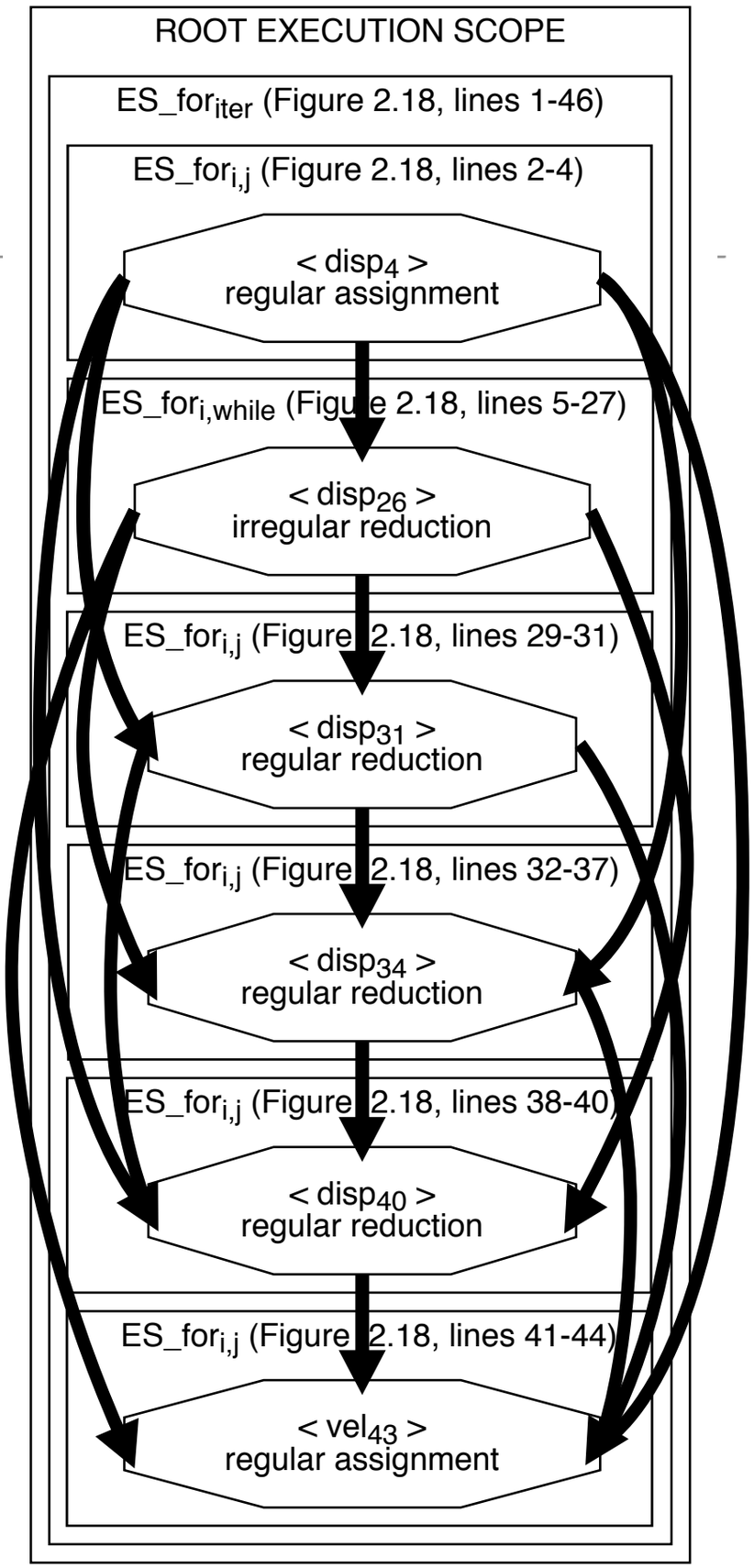
Initialization

Computation

Reduction

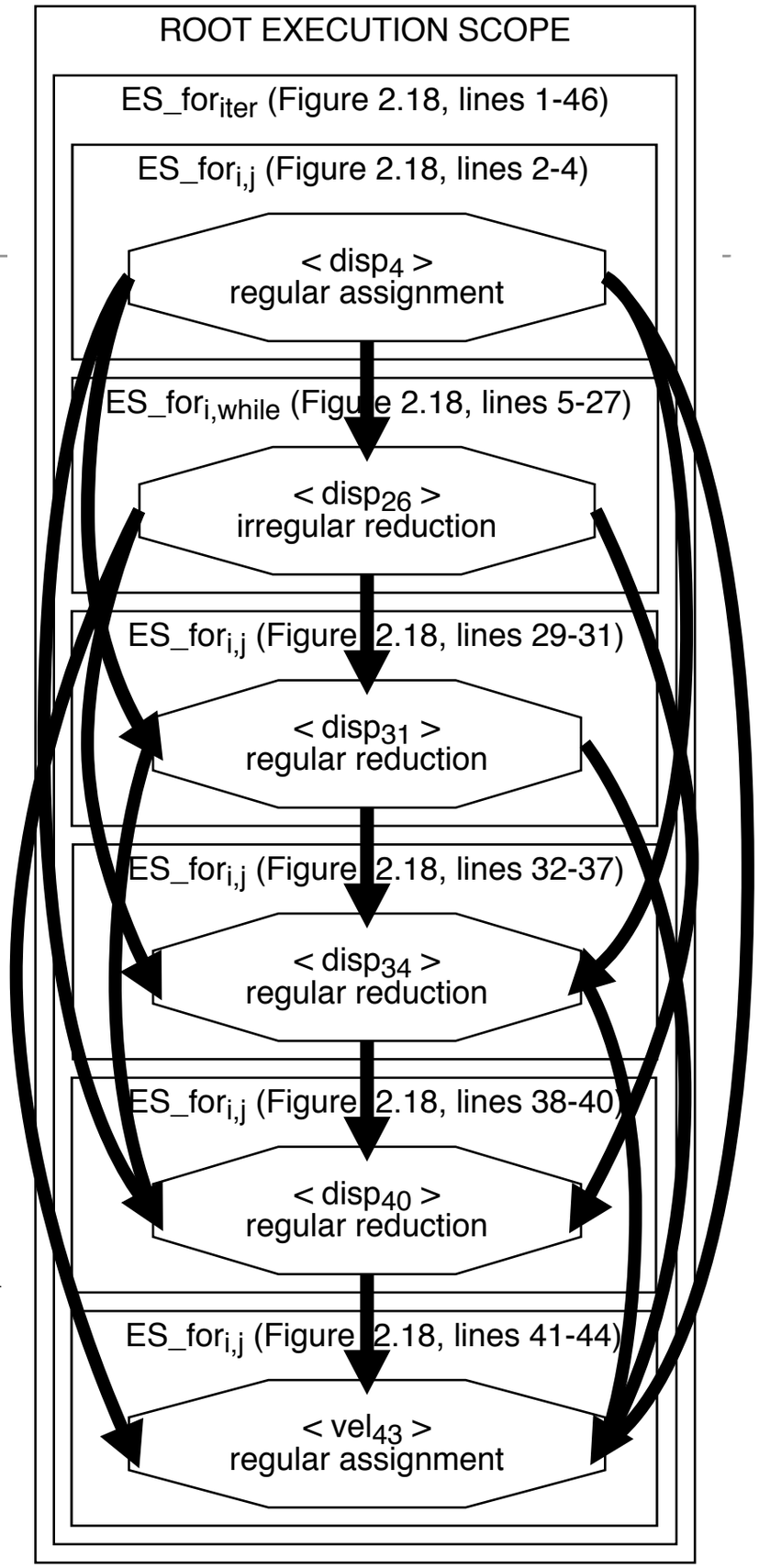
EQUAKE (I)

```
1  for (iter = 1; iter <= timesteps; iter++) {
2    for (i = 0; i < ARCHnodes; i++)
3      for (j = 0; j < 3; j++)
4        disp[disptplus][i][j] = 0.0;
5    for (i = 0; i < ARCHnodes; i++) {
6      Anext = ARCHmatrixindex[i]; Alast = ARCHmatrixindex[i+1];
7      sum0 = K[Anext][0][0] * disp[dispt][i][0]
8        + K[Anext][0][1] * disp[dispt][i][1]
9        + K[Anext][0][2] * disp[dispt][i][2];
10     sum1 = K[Anext][1][0] * ...; sum2 = K[Anext][2][0] * ...;
11     Anext++;
12     while (Anext < Alast) {
13       col = ARCHmatrixcol[Anext];
14       sum0 += K[Anext][0][0] * disp[dispt][col][0]
15         + K[Anext][0][1] * disp[dispt][col][1]
16         + K[Anext][0][2] * disp[dispt][col][2];
17       sum1 += K[Anext][1][0]*...; sum2 += K[Anext][2][0]*...;
18       disp[disptplus][col][0] +=
19         K[Anext][0][0] * disp[dispt][i][0]
20         + K[Anext][1][0] * disp[dispt][i][1]
21         + K[Anext][2][0] * disp[dispt][i][2];
22       disp[disptplus][col][1] += K[Anext][0][1] ...
23       disp[disptplus][col][2] += K[Anext][0][2] ...
24       Anext++;
25     }
26     disp[disptplus][i][0] += sum0; ...
27   }
```



EQUAKE (II)

```
28  time = iter * Exc.dt;
29  for (i = 0; i < ARCHnodes; i++)
30    for (j = 0; j < 3; j++)
31      disp[disptplus][i][j] *= - Exc.dt * Exc.dt;
32  for (i = 0; i < ARCHnodes; i++)
33    for (j = 0; j < 3; j++)
34      disp[disptplus][i][j] +=
35        2.0 * M[i][j] * disp[dispt][i][j]
36        - (M[i][j] - Exc.dt / 2.0 * C[i][j])
37        * disp[disptminus][i][j] - ...
38  for (i = 0; i < ARCHnodes; i++)
39    for (j = 0; j < 3; j++)
40      disp[disptplus][i][j] /= (M[i][j] + Exc.dt / 2.0 * C[i][j]);
41  for (i = 0; i < ARCHnodes; i++)
42    for (j = 0; j < 3; j++)
43      vel[i][j] = 0.5 / Exc.dt * (disp[disptplus][i][j]
44        - disp[disptminus][i][j]);
45  i = disptminus; disptminus = dispt; dispt = disptplus; disptplus = i
46 }
```



EQUAKE (III)

Minimization of thread creation/destruction

```
1  #pragma omp parallel shared(disp) private(disp___disptplus___private,...)
2  {
3    if (omp_get_thread_num() == 0) {
4      disp___disptplus___private = disp[disptplus];
5    } else {
6      disp___disptplus___private = (double **) malloc (ARCHnodes * sizeof(double *));
7      for (i = 0; i < ARCHnodes; i = i + 1)
8        disp___disptplus___private[i] = (double *) malloc(3 * sizeof(double));
9    }
10   for (iter = 1; iter < (timesteps + 1); iter = iter + 1) {
11 #pragma omp barrier
12     for (i = 0; i < ARCHnodes; i = i + 1)
13       for (j = 0; j < 3; j = j + 1)
14         disp___disptplus___private[i][j] = 0.0;
15 #pragma omp for schedule(static)
16     for (i = 0; i < ARCHnodes; i = i + 1) {
17       Anext = ARCHmatrixindex[i]; Alast = ARCHmatrixindex[i+1];
18       sum0 = K[Anext][0][0] * ...
19       Anext++;
20       while (Anext < Alast) {
21         col = ARCHmatrixcol[Anext];
22         sum0 += K[Anext][0][0] * ...
23         disp___disptplus___private[col][0] += K[Anext][0][0] * ...
24         Anext++;
25       }
26       disp___disptplus___private[i][0] += sum0; ...
27     }
28 #pragma omp critical
29     if (omp_get_thread_num() != 0)
30       for (i = 0; i < ARCHnodes; i = i + 1)
31         for (j = 0; j < 3; j = j + 1)
32           disp[disptplus][i][j] += disp___disptplus___private[i][j];
33 #pragma omp barrier
```

PARTIALLY PARALLEL LOOP

Initialization

Computation

Reduction

EQUAKE (and IV)

```
34      time = iter * Exc.dt;
35  #pragma omp for schedule(static) nowait
36      for (i = 0; i < ARCHnodes; i = i + 1)
37        for (j = 0; j < 3; j = j + 1)
38          disp[disptplus][i][j] *= - Exc.dt * Exc.dt;
39  #pragma omp for schedule(static) nowait
40      for (i = 0; i < ARCHnodes; i = i + 1)
41        for (j = 0; j < 3; j = j + 1)
42          disp[disptplus][i][j] += ...
43  #pragma omp for schedule(static) nowait
44      for (i = 0; i < ARCHnodes; i = i + 1)
45        for (j = 0; j < 3; j = j + 1)
46          disp[disptplus][i][j] /= ...
47  #pragma omp for schedule(static) nowait
48      for (i = 0; i < ARCHnodes; i = i + 1)
49        for (j = 0; j < 3; j = j + 1)
50          vel[i][j] = ...
51      i = disptminus; disptminus = dispt; dispt = disptplus; disptplus = i;
52    } /* for iter */
53    if (omp_get_thread_num() != 0) {
54      for (i = 0; i < ARCHnodes; i = i + 1)
55        free(disp___disptplus___private[i]);
56      free(disp___disptplus___private);
57    }
58  }
```

FULLY PARALLEL LOOP

FULLY PARALLEL LOOP

FULLY PARALLEL LOOP

FULLY PARALLEL LOOP

Outline

2. A Novel Compiler Support for Multicore Systems

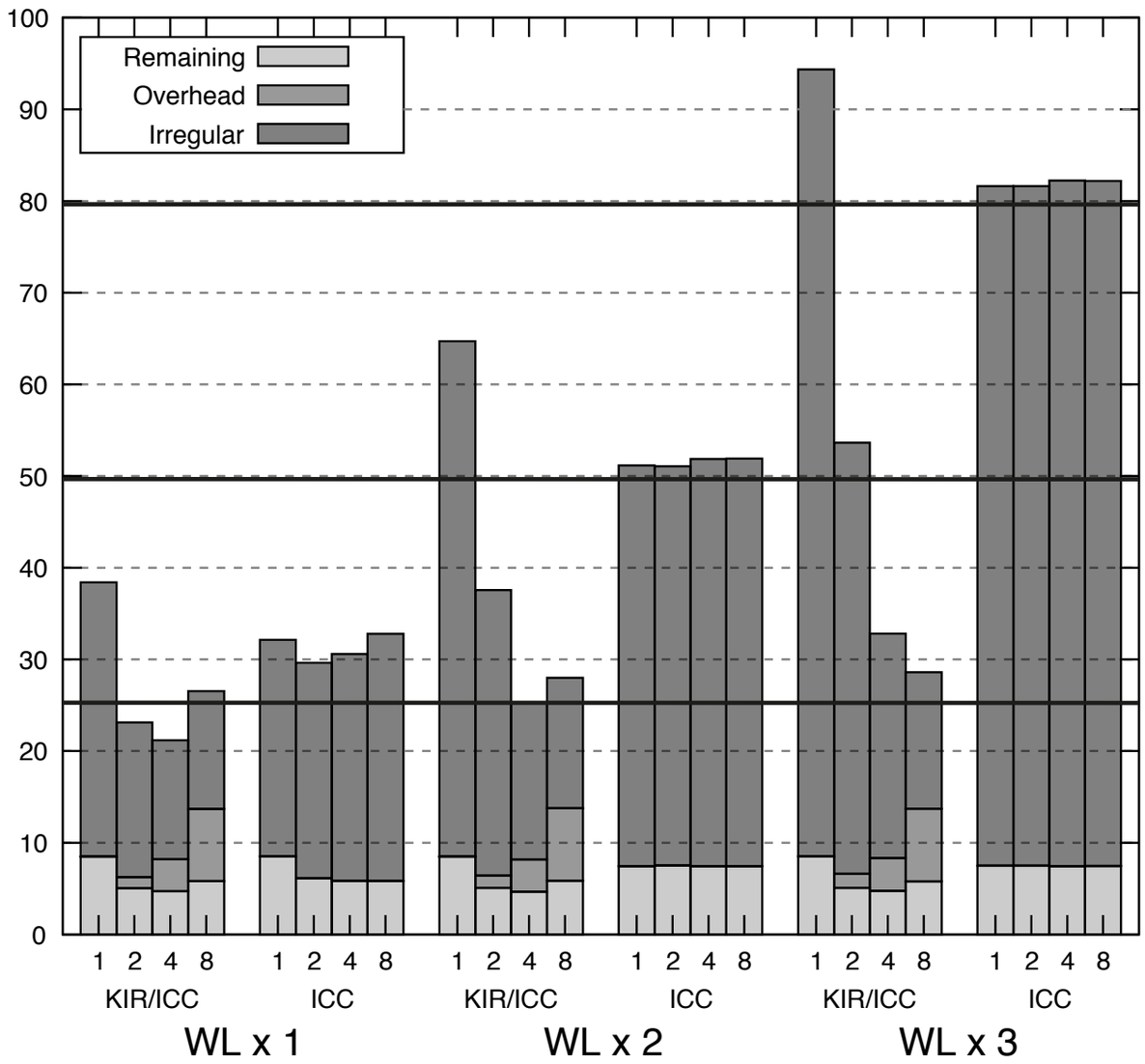
   - KIR: A diKernel-based IR
   - Automatic Partitioning driven by the KIR
   - Automatic Parallelization of the Benchmark Suite
   - **Experimental Evaluation**

Benchmark		Program Characteristics						Compilers			
		diKernel	Irreg. writes	Irreg. reads	Unknown LB	Complex CF	Temp. vars	GCC	ICC	PLUTO	KIR
Synthetic	reg. assig.	regular assignment						√	√	√	√
	irreg. assig.	irregular assignment	√		√						√
	sc. reduc. 1	scalar reduction						≈	√		√
	sc. reduc. 2	scalar reduction						≈	√		√
	sc. reduc. 3	scalar reduction				√		≈	√		√
	reg. reduc.	regular reduction						√	√	√	√
	irreg. reduc.	irregular reduction	√	√	√						√
	reg. recurr.	regular recurrence									√
Algebra	DenseAMUX	regular assignment					√		√	≈	√
	AMUX	regular assignment		√	√		√				√
	AMUXMS	regular reduction		√	√						√
	ATMUX	irregular reduction	√	√	√						√
Im.	sobel1	regular assignment				√	√				√
	sobel2	regular assignment					√	√			√
Apps	SWIM	regular recurrence				√			√	*U*	√
	EQUAKE	irregular reduction	√	√	√				≈		√

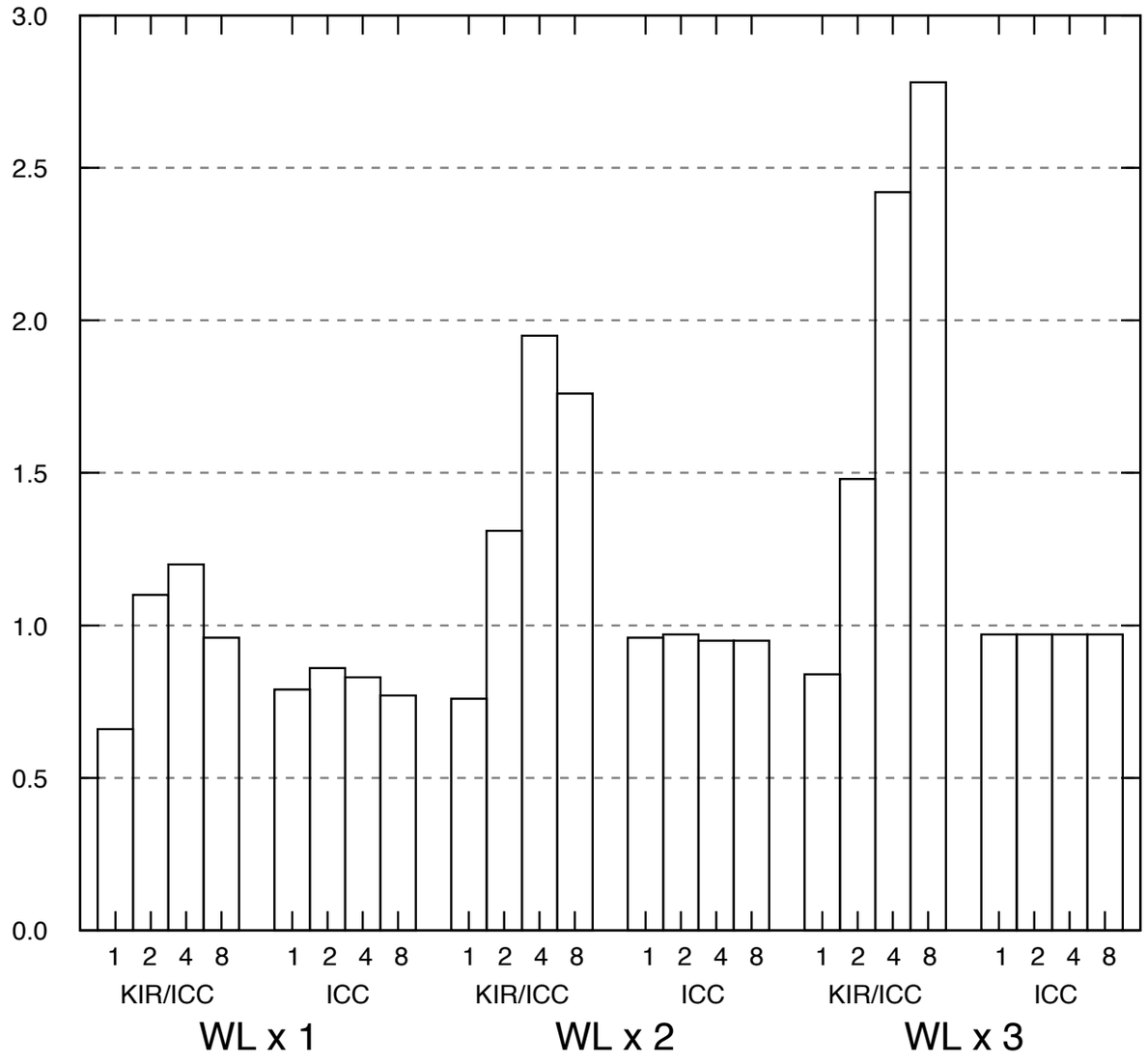
Effectiveness

2 Intel Xeon E5520 quad-core Nehalem processors at 2.26 GHz with 8 MB of cache memory per processor and 8 GB of RAM

Performance: EQUAKE (Execution Time)



Execution Time (s)



Speedup

Outline

1. Introduction
2. A Novel Compiler Support for Multicore Systems
3. **Locality-Aware Automatic Parallelization for GPGPU**
4. Trace-Based Affine Reconstruction of Code
5. Conclusions

Outline

3. Locality-Aware Automatic Parallelization for GPGPU
   - GPGPU with CUDA and OpenHMPP
   - Locality-Aware Generation of Efficient GPGPU Code
   - CONV3D & SGEMM
   - Experimental Evaluation

Outline

3. Locality-Aware Automatic Parallelization for GPGPU
   - **GPGPU with CUDA and OpenHMPP**
   - Locality-Aware Generation of Efficient GPGPU Code
   - CONV3D & SGEMM
   - Experimental Evaluation

GPGPU with CUDA

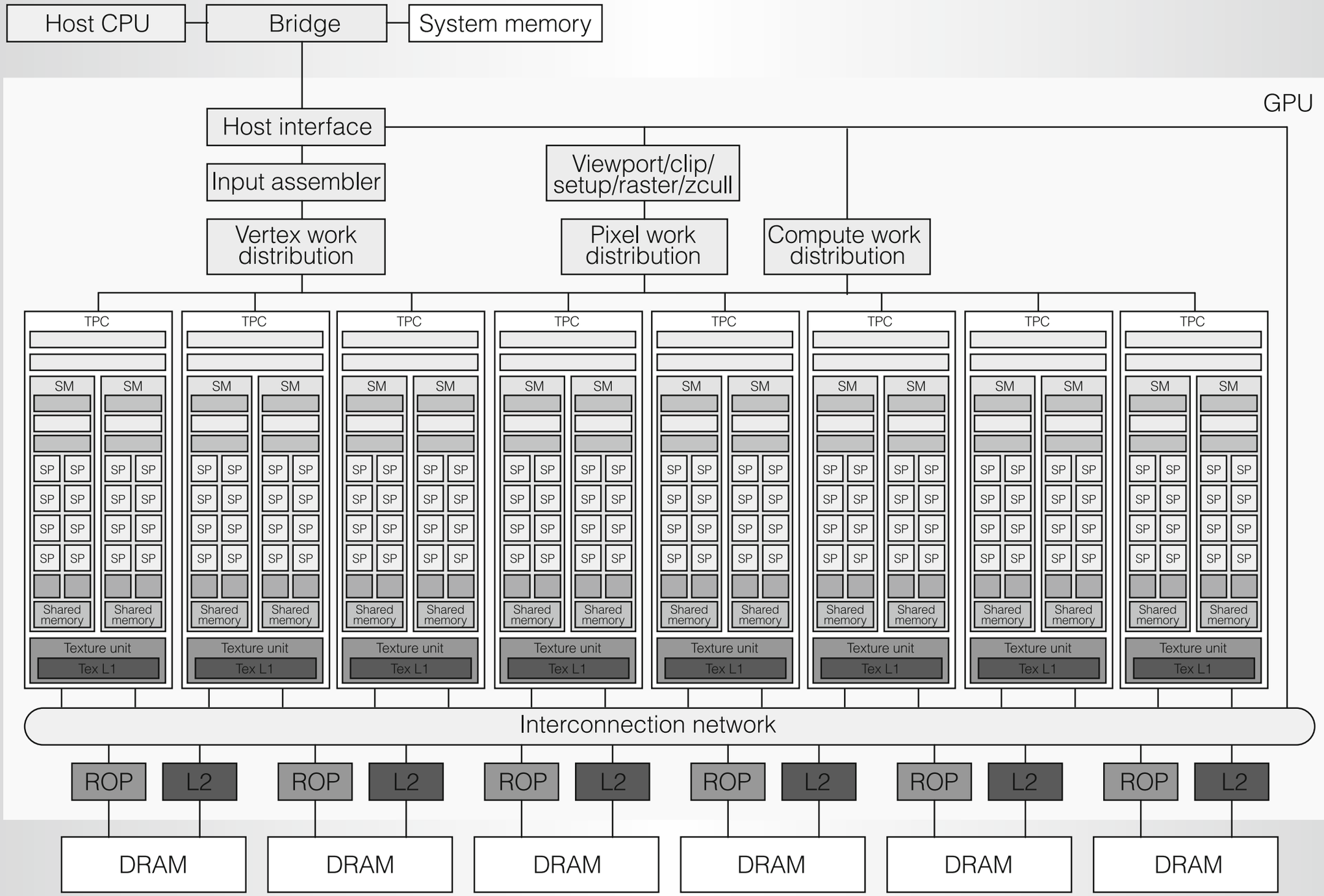
- First GPGPU programs look like graphics applications

- CUDA enables the use of C

CUDA kernel: specifies the operation of a single GPU thread

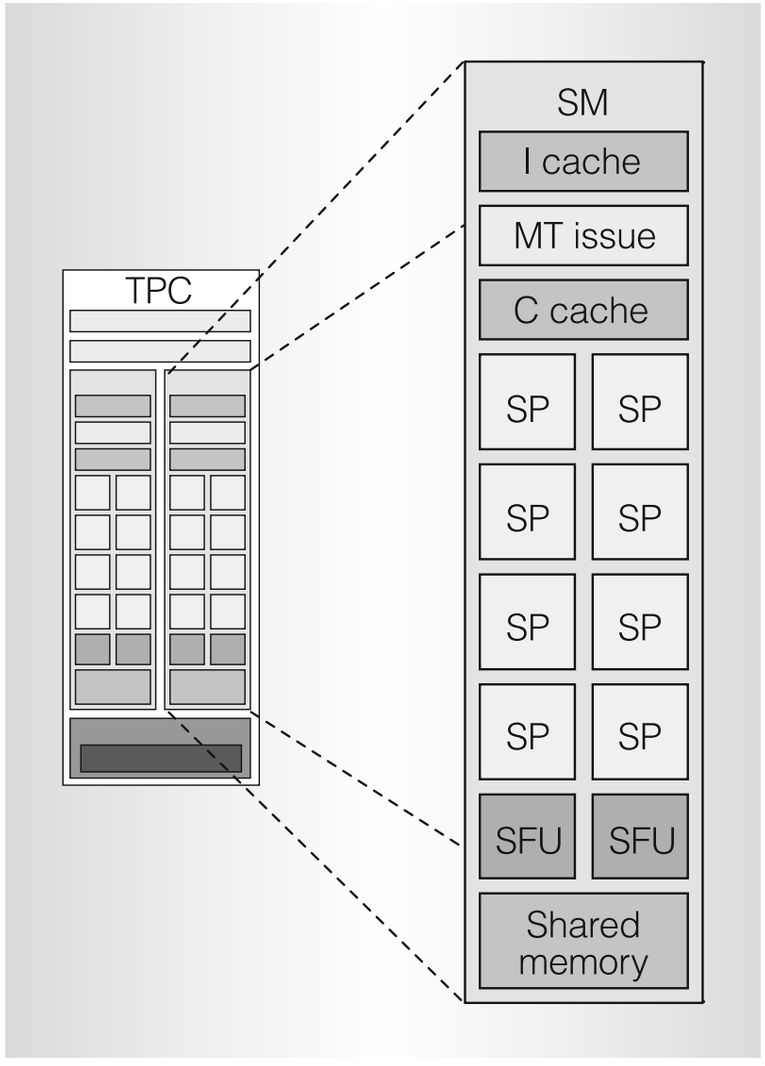
- Main ideas:

1. Lightweight parallel threads in hierarchy: grid, block
2. Shared memory
3. Barriers



Example of CUDA-enabled GPU architecture

GPU Memories



	Location	Access	Scope
registers	SM	read & write	one GPU thread
local memory	DRAM	read & write	one GPU thread
shared memory	SM	read & write	all GPU threads in a block
global memory	DRAM	read & write	all GPU threads & CPU

explicit allocations and transfers

GPU Programming Features in CUDA

Impact on performance: + (top) to - (bottom)

1	Threadification
2	Thread grouping: *warps*
3	Minimization of CPU-GPU data transfers
4	Coalescing
5	Maximization of the usage of registers and shared memory
6	Divergency
7	Occupancy
8	Threads per block

GPGPU with OpenHMPP

			
interaction	RPC	RPC	RPC
address spaces	disjoint	disjoint	disjoint
data transfers	automatic & manual	automatic & manual	automatic & manual
sw-managed caches	explicit handling	explicit handling	automatic handling
parallelism specification	loop iterations	gangs, workers, SIMD	loop iterations, tasks, SIMD
standard loop transformations	directives	no	no

Outline

3. Locality-Aware Automatic Parallelization for GPGPU

   - GPGPU with CUDA and OpenHMPP
   - **Locality-Aware Generation of Efficient GPGPU Code**
   - CONV3D & SGEMM
   - Experimental Evaluation

GPU Programming Features addressed by our Automatic Technique

1	Threadification
2	Thread grouping: *warps*
3	Minimization of CPU-GPU data transfers
4	Coalescing
5	Maximization of the usage of registers and shared memory
6	Divergency
7	Occupancy
8	Threads per block

Impact on performance: + / -

Chains of Recurrences (chrecs)

- Algebraic formalism

Definition 3.3.1. *Given a constant $\phi \in \mathbb{Z}$, a function $g : \mathbb{N}^0 \to \mathbb{Z}$, and the operator $+$, the **chrec** $f = \{\phi, +, g\}$ is defined as a function $f : \mathbb{N}^0 \to \mathbb{Z}$ such that:*

$$\{\phi, +, g\}(i) = \phi + \sum_{j=0}^{i-1} g(j)$$

- Useful for representing the iterations of a loop and array access patterns

```
2 for (i = 0; i <= N; i++) {
3    for (j = 0; j <= N; j++) {
4       ... x[i][j] ...
5    }
6 }
```

$CHRECS\_x_k$ = [{0,+,1}][{0,+,1}]

- We instantiate (particularize) them for each GPU thread

Detection of Coalesced Accesses to the GPU Global Memory

row major

CHRECS_x_k = [{0,+,1}][{0,+,1}]

column major

```
1 // only for_i is threadified
2 for (i = 0; i <= N; i++) {
3    for (j = 0; j <= N; j++) {
4       ... x[i][j] ...
5    }
6 }
```

(a) Source code S1.

		T0 (i=0)	*T1* (i=1)	*T2* (i=2)
	j=0	$x[0][0]$	$x[1][0]$	$x[2][0]$
	j=1	$x[0][1]$	$x[1][1]$	$x[2][1]$
	j=2	$x[0][2]$	$x[1][2]$	$x[2][2]$
	...	...	...	...
chrecs	$1^{st}dim$	$\{0\}$	$\{1\}$	$\{2\}$
	$2^{nd}dim$	$\{0,+,1\}$	$\{0,+,1\}$	$\{0,+,1\}$

(b) Non-coalesced accesses.

```
1 // only for_j is threadified
2 for (j = 0; j <= N; j++) {
3    for (i = 0; i <= N; i++) {
4       ... x[i][j] ...
5    }
6 }
```

(c) Source code S2.

		T0 (j=0)	*T1* (j=1)	*T2* (j=2)
	i=0	$x[0][0]$	$x[0][1]$	$x[0][2]$
	i=1	$x[1][0]$	$x[1][1]$	$x[1][2]$
	i=2	$x[2][0]$	$x[2][1]$	$x[2][2]$
	...	...	...	...
chrecs	$1^{st}dim$	$\{0,+,1\}$	$\{0,+,1\}$	$\{0,+,1\}$
	$2^{nd}dim$	$\{0\}$	$\{1\}$	$\{2\}$

(d) Coalesced accesses.

the same

convex set

Detection of whether an Access to the GPU Global Memory can be Coalesced

1: **FUNCTION** isCoalescedAccess
Input: access $x_k[i_{k,1}][i_{k,2}]\dots[i_{k,n}]$ to an n-dimensional array x stored in row-major order
Input: loop nest $L = L_1, L_2, \dots, L_l$ where L_1 is the threadified loop
Output: returns whether the given access x_k can be coalesced after threadifying the loop nest L
2: $CHRECS\_x_k \leftarrow [\{\phi_{k,1}, +, g_{k,1}\}][\{\phi_{k,2}, +, g_{k,2}\}]\dots[\{\phi_{k,n}, +, g_{k,n}\}]$
3: $W \leftarrow$ warp of GPU threads $\{T0, T1, T2\dots\}$
4: **for each** thread Ti in W **do**
5: $CHRECS\_x_k^{Ti} \leftarrow [\{\phi_{k,1}^{Ti}, +, g_{k,1}^{Ti}\}][\{\phi_{k,2}^{Ti}, +, g_{k,2}^{Ti}\}]\dots[\{\phi_{k,n}^{Ti}, +, g_{k,n}^{Ti}\}]$
6: **end for**
7: **if** $(\exists d \in \{1\dots n-1\}, Tj \in W - \{T0\} : \{\phi_{k,d}^{Tj}, +, g_{k,d}^{Tj}\} \neq \{\phi_{k,d}^{T0}, +, g_{k,d}^{T0}\})$ **then**
8: **return** false ▷ first $n-1$ chrecs differ
9: **end if**
10: $CHRECS\_RANGE\_x_{k,n} \leftarrow \bigcup^{Ti}\{\phi_{k,n}^{Ti}, +, g_{k,n}^{Ti}\}$
11: **if** $CHRECS\_RANGE\_x_{k,n}$ defines a convex set **then**
12: **return** true ▷ threads of the warp access consecutive locations
13: **else**
14: **return** $(\forall\, Tj \in W - \{T0\} : \{\phi_{k,n}^{Tj}, +, g_{k,n}^{Tj}\} = \{\phi_{k,n}^{T0}, +, g_{k,n}^{T0}\})$
▷ threads of the warp access the same location
15: **end if**
16: **end FUNCTION**

Usage of Registers to Store Reused Data within a GPU Thread

1: **PROCEDURE** StoreReusedDataInRegisters
Input: n-dimensional array $x[s_1][s_2]\dots[s_n]$
Input: loop nest $L = L_1, L_2, \dots, L_l$ where L_1 is the threadified loop
Output: a modified program that exploits reused data to maximize the usage of the GPU registers
2: collect accesses $x_k[i_{k,1}][i_{k,2}]\dots[i_{k,n}]$ with $k \in \{1,\dots,m\}$
3: $CHRECS\_x_k \leftarrow [\{\phi_{k,1}, +, g_{k,1}\}][\{\phi_{k,2}, +, g_{k,2}\}]\dots[\{\phi_{k,n}, +, g_{k,n}\}]$
4: **for each** thread Ti **do**
5: $CHRECS\_x_k^{Ti} \leftarrow [\{\phi_{k,1}^{Ti}, +, g_{k,1}^{Ti}\}][\{\phi_{k,2}^{Ti}, +, g_{k,2}^{Ti}\}]\dots[\{\phi_{k,n}^{Ti}, +, g_{k,n}^{Ti}\}]$
6: $REUSED\_DATA\_x^{Ti} \leftarrow \bigcap_{k=1}^{m} CHRECS\_x_k^{Ti}$
7: **if** $(REUSED\_DATA\_x^{Ti} \neq \emptyset)$ **then**
8: store reused data between the accesses made by Ti in its set of registers if data are private
9: **end if**
10: **end for**
11: **end PROCEDURE**

Usage of the GPU Shared Memory for Data Shared between the Threads of a Block

1: **PROCEDURE** StoreSharedDataInSharedMemory
Input: n-dimensional array $x[s_1][s_2]\dots[s_n]$
Input: loop nest $L = L_1, L_2, \dots, L_l$ where L_1 is the threadified loop
Output: a modified program using the GPU shared memory to share data between the threads of a block
2: collect accesses $x_k[i_{k,1}][i_{k,2}]\dots[i_{k,n}]$ with $k \in \{1, \dots, m\}$
3: $CHRECS\_x_k \leftarrow [\{\phi_{k,1}, +, g_{k,1}\}][\{\phi_{k,2}, +, g_{k,2}\}]\dots[\{\phi_{k,n}, +, g_{k,n}\}]$
4: **for each** block B **do**
5: **for each** thread Ti in B **do**
6: $CHRECS\_x_k^{Ti} \leftarrow [\{\phi_{k,1}^{Ti}, +, g_{k,1}^{Ti}\}][\{\phi_{k,2}^{Ti}, +, g_{k,2}^{Ti}\}]\dots[\{\phi_{k,n}^{Ti}, +, g_{k,n}^{Ti}\}]$
7: **end for**
8: $SHDATA\_x \leftarrow \bigcap^{Ti} CHRECS\_x_k^{Ti}$ with $k \in \{1, \dots, m\}$
9: **if** $(SHDATA\_x \neq \emptyset)$ **then**
10: store data shared between the threads of block B in the shared memory
11: **end if**
12: **end for**
13: **end PROCEDURE**

Increase the Computational Load of a GPU Thread

1: **PROCEDURE** INCREASELOAD

Input: access $x_k[i_{k,1}][i_{k,2}]\ldots[i_{k,n}]$ to an n-dimensional array x stored in row-major order

Input: loop nest $L = L_1, L_2, \ldots, L_l$ where both L_1, L_2 are threadified

Input: amount of data Δ to be processed by a GPU thread

Output: a modified program after applying loop tiling under the OpenHMPP programming model

2: increment the step of the outer loop L_1 to Δ

3: **for each** scalar variable s in L **do**

4: promote s to an array $s[\Delta]$

5: transform reads and writes to s into loops of Δ iterations

6: **end for**

7: **end PROCEDURE**

Use Scalar Variables to Enable GPU Compiler Optimizations

1: **PROCEDURE** INCREASELOAD

Input: loop nest $L = L_1, L_2, L_3 \ldots, L_l$ that results of Algorithm 3.4 where both L_1, L_2 are threadified, the step of L_1 is Δ, and L_3 is the created loop with Δ iterations

Output: a modified program that uses more scalar variables to enable GPU compiler optimizations

2: apply loop fission to L_3, the loop created in line 5 of Algorithm 3.4
3: **for each** loop L_3' resulting from the fission of L_3 **do**
4: interchange loops until L_3' is the innermost one
5: insert a `fullunroll` directive before L_3'
6: **end for**
7: **end PROCEDURE**

Outline

3. Locality-Aware Automatic Parallelization for GPGPU
   - GPGPU with CUDA and OpenHMPP
   - Locality-Aware Generation of Efficient GPGPU Code
   - **CONV3D & SGEMM**
   - Experimental Evaluation

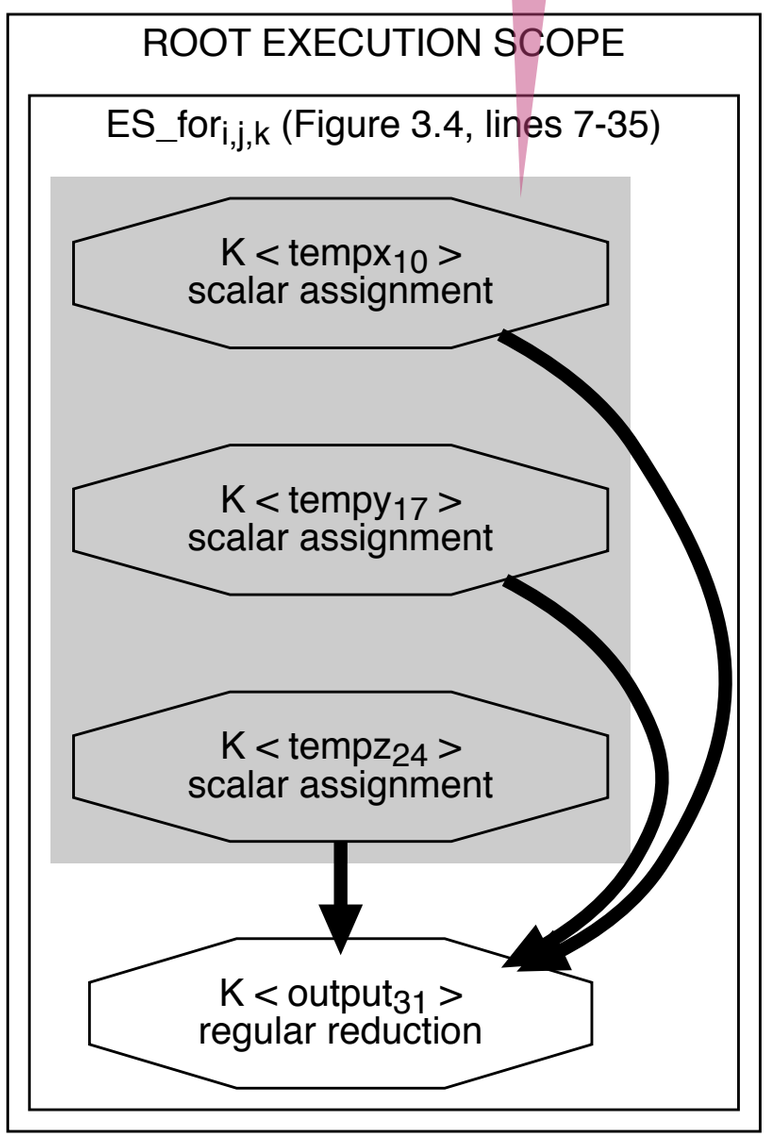
CONV3D & SGEMM

GPU Features	*conv3d-cpu*	*conv3d-hmpp1*	*conv3d-hmpp2*	*conv3d-hmpp3*	*sgemm-cpu*	*sgemm-mkl*	*sgemm-hmpp1*	*sgemm-hmpp2*	*sgemm-hmpp3*	*sgemm-hmpp4*	*sgemm-cublas*
Coalescing	-	√	√	√	-	-	√	√	√	√	-
Registers	-		√	√	-	-			√	√	-
Shared Memory	-			√	-	-				√	-

CONV3D (I)

```
1  int sizex, sizey, sizez, bound = 4;
2
3  void conv3d(float output[sizex][sizey][sizez],
4    float input[bound+sizex+bound][4+sizey+4][4+sizez+4],
5    float coefx, float coefy, float coefz) {
6
7    for (int i = 0; i < sizex; i++) {
8      for (int j = 0; j < sizey; j++) {
9        for (int k = 0; k < sizez; k++) {
10         float tempx = input[i][j][k] + coefx *
11           (
12             input[i-1][j][k] + input[i+1][j][k] +
13             input[i-2][j][k] + input[i+2][j][k] +
14             input[i-3][j][k] + input[i+3][j][k] +
15             input[i-4][j][k] + input[i+4][j][k]
16           );
17         float tempy = input[i][j][k] + coefy *
18           (
19             input[i][j-1][k] + input[i][j+1][k] +
20             input[i][j-2][k] + input[i][j+2][k] +
21             input[i][j-3][k] + input[i][j+3][k] +
22             input[i][j-4][k] + input[i][j+4][k]
23           );
24         float tempz = input[i][j][k] + coefz *
25           (
26             input[i][j][k-1] + input[i][j][k+1] +
27             input[i][j][k-2] + input[i][j][k+2] +
28             input[i][j][k-3] + input[i][j][k+3] +
29             input[i][j][k-4] + input[i][j][k+4]
30            );
31         output[i][j][k] =
32           output[i][j][k] + tempx + tempy + tempz;
33       }
34     }
35   }
36 }
```

shaded to be omitted in the discovering of parallelism



FULLY PARALLEL LOOP

CONV3D (II)

- conv3d-cpu: Sequential code
- conv3d-hmpp1: Coalescing

```
1. int i, j, k, size_x, size_y, size_z;
2. float coefx,coefy,coefz,*input,*output;
3.
4. for (i = 0; i < size_x; i++) {
5.  for (j = 0; j < size y; j++) {
6.   for (k = 0; k < size_z; k++) {
7.    float tempx = input[i][j][k]+coefx*
8. …
```

$CHRECS\_input_1$ = [{0,+,1}][{0,+,1}][{0,+,1}]

- Default OpenHMPP policy

$CHRECS\_input_1^{T0}$ = [{0}][{0}][{0,+,1}]

$CHRECS\_input_1^{T1}$ = [{0}][{1}][{0,+,1}]

- Loop nest is permuted to ***forj, fork, fori***

$CHRECS\_input_1^{T0}$ = [{0,+,1}][{0}][{0}]

$CHRECS\_input_1^{T1}$ = [{0,+,1}][{0}][{1}]

CONV3D (III)

- conv3d-hmpp2: Registers

```
4. for (j = 0; j < size_y; j++) {
5.   for (k = 0; k < size_z; k++) {
6.     for (i = 0; i < size_x; i++) {
7.       float tempx = input[i][j][k]+coefx*
8.       (
9.       input[i-1][j][k]+input[i+1][j][k]+
10.…
```

$CHRECS\_input_1 = [\{0,+,1\}][\{0,+,1\}][\{0,+,1\}]$

$CHRECS\_input_2 = [\{-1,+,1\}][\{0,+,1\}][\{0,+,1\}]$

$CHRECS\_input_3 = [\{1,+,1\}][\{0,+,1\}][\{0,+,1\}]$

$CHRECS\_input_1^{T0} = [\{0,+,1\}][\{0\}][\{0\}]$

$CHRECS\_input_2^{T0} = [\{-1,+,1\}][\{0\}][\{0\}]$

$CHRECS\_input_3^{T0} = [\{1,+,1\}][\{0\}][\{0\}]$

$\cap \neq \emptyset$

CONV3D (IV)

```
1  #pragma hmpp conv3d___hmpp2 codelet
2  void conv3d___hmpp2(float output[sizex][sizey][sizez],
3    float input[bound+sizex+bound][4+sizey+4][4+sizez+4],
4    float coefx, float coefy, float coefz) {
5
6  #pragma hmppcg gridify (j, k)
7    for (int j = 0; j < sizey; j++) {
8      for (int k = 0; k < sizez; k++) {
9        float i___minus4 = 0;
10       float i___minus3 = input[-4][j][k];
11       float i___minus2 = input[-3][j][k];
12       float i___minus1 = input[-2][j][k];
13       float i___plus0 = input[-1][j][k];
14       float i___plus1 = input[0][j][k];
15       float i___plus2 = input[1][j][k];
16       float i___plus3 = input[2][j][k];
17       float i___plus4 = input[3][j][k];
18       for (int i = 0; i < sizex; i++) {
19         i___minus4 = i___minus3;
20         i___minus3 = i___minus2;
21         i___minus2 = i___minus1;
22         i___minus1 = i___plus0;
23         i___plus0 = i___plus1;
24         i___plus1 = i___plus2;
25         i___plus2 = i___plus3;
26         i___plus3 = i___plus4;
27         i___plus4 = input[i+4][j][k];
28         float tempx = i___plus0 + coefx *
29           (
30             i___minus1 + i___plus1 +
31             i___minus2 + i___plus2 +
32             i___minus3 + i___plus3 +
33             i___minus4 + i___plus4
34           );
35         float tempy = ...
36         float tempz = ...
37         output[i][j][k] =
38           output[i][j][k] + tempx + tempy + tempz;
39       }
40     }
41   }
42 }
```

CONV3D (V)

- conv3d-hmpp3: Shared memory

```
4.for (j = 0; j < size_y; j++) {
5.   for (k = 0; k < size_z; k++) {
6.     for (i = 0; i < size_x; i++) {
…
21.    float tempz = input[i][j][k]+coefz*
22.    (
23.      input[i][j][k-1]+input[i][j][k+1]+
24.      input[i][j][k-2]+input[i][j][k+2]+
25.      input[i][j][k-3]+input[i][j][k+3]+
26.      input[i][j][k-4]+input[i][j][k+4]
27.    );
…
```

CHRECS	*T0*			*T1*		
	$1^{st}dim$	$2^{nd}dim$	$3^{rd}dim$	$1^{st}dim$	$2^{nd}dim$	$3^{rd}dim$
$CHRECS\_input_{19}$	$\{0,+,1\}$	$\{0\}$	$\{0\}$	$\{0,+,1\}$	$\{0\}$	$\{1\}$
$CHRECS\_input_{20}$	$\{0,+,1\}$	$\{0\}$	$\{-1\}$	$\{0,+,1\}$	$\{0\}$	$\{0\}$
$CHRECS\_input_{21}$	$\{0,+,1\}$	$\{0\}$	$\{1\}$	$\{0,+,1\}$	$\{0\}$	$\{2\}$
$CHRECS\_input_{22}$	$\{0,+,1\}$	$\{0\}$	$\{-2\}$	$\{0,+,1\}$	$\{0\}$	$\{-1\}$
$CHRECS\_input_{23}$	$\{0,+,1\}$	$\{0\}$	$\{2\}$	$\{0,+,1\}$	$\{0\}$	$\{3\}$
$CHRECS\_input_{24}$	$\{0,+,1\}$	$\{0\}$	$\{-3\}$	$\{0,+,1\}$	$\{0\}$	$\{-2\}$
$CHRECS\_input_{25}$	$\{0,+,1\}$	$\{0\}$	$\{3\}$	$\{0,+,1\}$	$\{0\}$	$\{4\}$
$CHRECS\_input_{26}$	$\{0,+,1\}$	$\{0\}$	$\{-4\}$	$\{0,+,1\}$	$\{0\}$	$\{-3\}$
$CHRECS\_input_{27}$	$\{0,+,1\}$	$\{0\}$	$\{4\}$	$\{0,+,1\}$	$\{0\}$	$\{5\}$

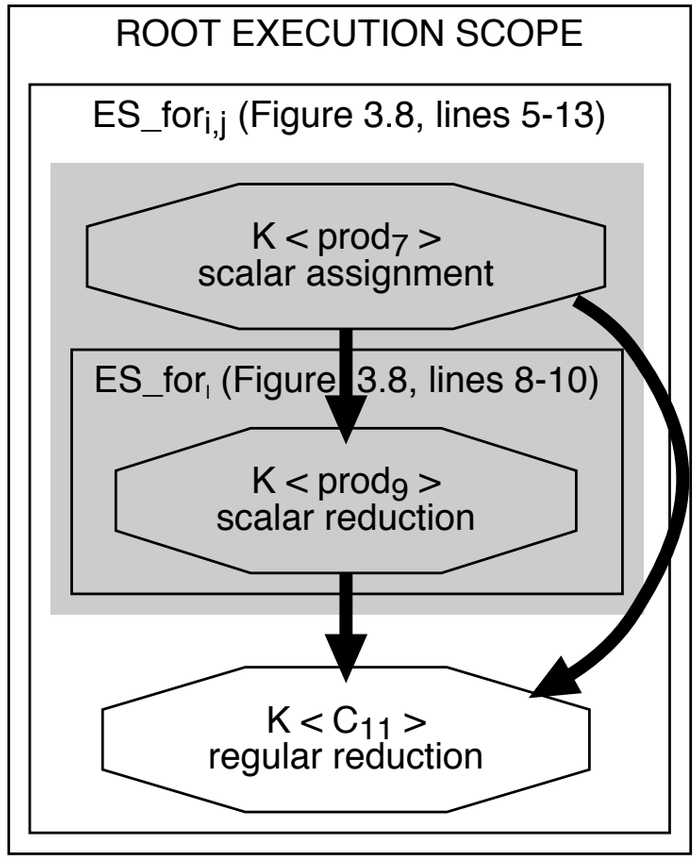
`shared` clause of the `gridify` directive

CONV3D (and VI)

```
1  #pragma hmpp conv3d___hmpp3 codelet
2  void conv3d___hmpp3(float output[sizex][sizey][sizez],
3    float input[bound+sizex+bound][4+sizey+4][4+sizez+4],
4    float coefx, float coefy, float coefz) {
5    float input___shared[bound+8+bound][bound+32+bound];
6  #pragma hmppcg gridify(j,k),blocksize(32x8),shared(input___shared),unguarded
7    for (int j = 0; j < sizey; j++) {
8      for (int k = 0; k < sizez; k++) {
9        int tx = 0;
10       int ty = 0;
11 #pragma hmppcg set tx = RankInBlockX()
12 #pragma hmppcg set ty = RankInBlockY()
13       int rk = tx + bound;
14       int rj = ty + bound;
15       float i___minus4 = ...
16       for (int i = 0; i < sizex; i++) {
17         i___minus4 = ...
18 #pragma hmppcg grid barrier
19         input___shared[rj-bound][rk-bound] = input[i][j-bound][k-bound];
20         input___shared[rj+bound][rk-bound] = input[i][j+bound][k-bound];
21         input___shared[rj-bound][rk+bound] = input[i][j-bound][k+bound];
22         input___shared[rj+bound][rk+bound] = input[i][j+bound][k+bound];
23 #pragma hmppcg grid barrier
24         float tempx = ...
25         float tempy = i___plus0 + coefy *
26           (
27             input___shared[rj-1][rk] + input___shared[rj+1][rk] +
28             input___shared[rj-2][rk] + input___shared[rj+2][rk] +
29             input___shared[rj-3][rk] + input___shared[rj+3][rk] +
30             input___shared[rj-4][rk] + input___shared[rj+4][rk]
31           );
32         float tempz = i___plus0 + coefz *
33           (
34             input___shared[rj][rk-1] + input___shared[rj][rk+1] +
35             input___shared[rj][rk-2] + input___shared[rj][rk+2] +
36             input___shared[rj][rk-3] + input___shared[rj][rk+3] +
37             input___shared[rj][rk-4] + input___shared[rj][rk+4]
38            );
39         output[i][j][k] =
40           output[i][j][k] + tempx + tempy + tempz;
41       }
42     }
43   }
44 }
```

SGEMM (I)

```
1  int m, n, k;
2  void sgemm(float C[m][n], float alpha, float A[m][k],
3    float B[k][n], float beta) {
4
5    for (int i = 0; i < m; i++) {
6      for (int j = 0; j < n; j++) {
7        float prod = 0;
8        for (int l = 0; l < k; l++) {
9          prod += A[i][l] * B[l][j];
10       }
11       C[i][j] = alpha * prod + beta * C[i][j];
12     }
13   }
14 }
```



FULLY PARALLEL LOOP

SGEMM (II)

- sgemm-cpu: Sequential code
- sgemm-mkl: Intel MKL
- sgemm-hmpp1: Offloading (and check coalescing)

```
1  int m, n, k;
2  void sgemm(float C[m][n], float alpha, float A[m][k],
3    float B[k][n], float beta) {
4
5    for (int i = 0; i < m; i++) {
6      for (int j = 0; j < n; j++) {
7        float prod = 0;
8        for (int l = 0; l < k; l++) {
9          prod += A[i][l] * B[l][j];
10       }
11       C[i][j] = alpha * prod + beta * C[i][j];
12     }
13   }
14 }
```

CHRECS	not instantiated		*T0*		*T1*	
	$1^{st}dim$	$2^{nd}dim$	$1^{st}dim$	$2^{nd}dim$	$1^{st}dim$	$2^{nd}dim$
CHRECS_A	$\{0,+,1\}$	$\{0,+,1\}$	$\{0\}$	$\{0,+,1\}$	$\{0\}$	$\{0,+,1\}$
CHRECS_B	$\{0,+,1\}$	$\{0,+,1\}$	$\{0,+,1\}$	$\{0\}$	$\{0,+,1\}$	$\{1\}$
CHRECS_C	$\{0,+,1\}$	$\{0,+,1\}$	$\{0\}$	$\{0\}$	$\{0\}$	$\{1\}$

SGEMM (III)

- sgemm-hmpp2: Tiling preserving coalescing

```
1  int m, n, k;
2  #define DELTA 16
3
4  #pragma hmpp sgemm___hmpp2 codelet
5  void sgemm___hmpp2(float C[m][n], float alpha, float A[m][k],
6    float B[k][n], float beta) {
7
8  #pragma hmppcg gridify (i,j), blocksize(128x1)
9    for (int i = 0; i < m; i = i + DELTA) {
10     for (int j = 0; j < n; j++) {
11       float prod[DELTA];
12       for (int t = 0; t < DELTA; t++) {
13         prod[t] = 0;
14         for (int l = 0; l < k; l++) {
15           prod[t] += A[i+t][l] * B[l][j];
16         }
17         C[i+t][j] = alpha * prod[t] + beta * C[i+t][j];
18       }
19     }
20   }
21 }
```

SGEMM (and IV)

- sgemm-hmpp3: Let the compiler use the registers (`fullunroll`)

- sgemm-hmpp4: Use the shared memory for *B*

- sgemm-cublas: NVIDIA CUBLAS library

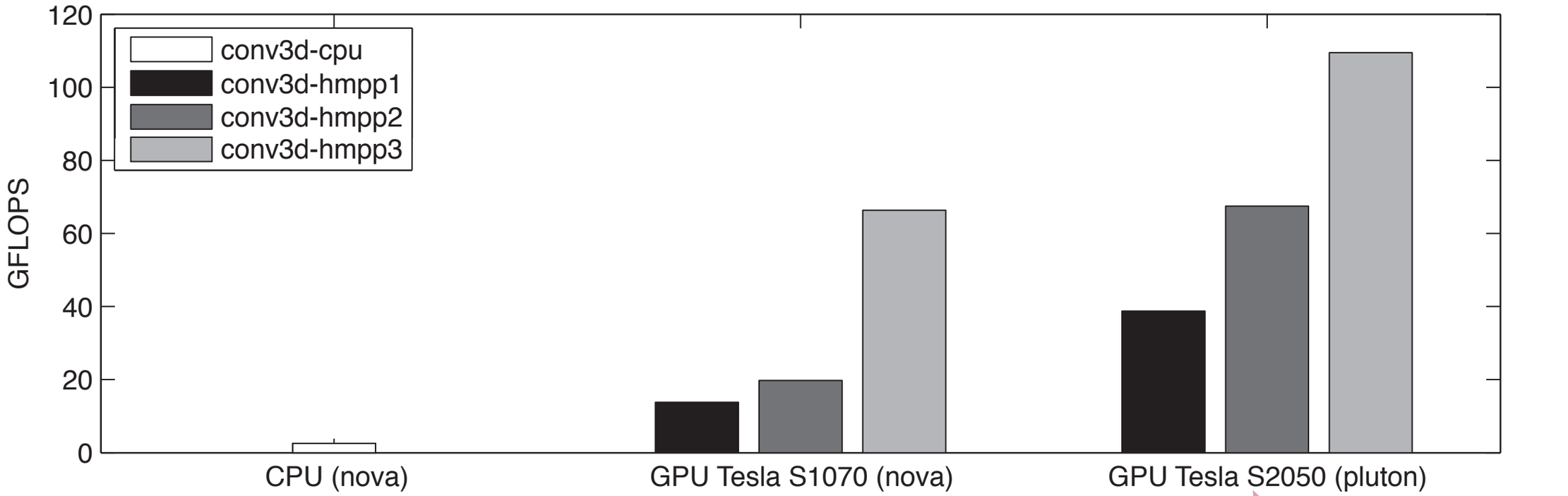
```
1  int m, n, k;
2  #define DELTA 16
3
4  #pragma hmpp sgemm___hmpp3 codelet
5  void sgemm___hmpp3(float C[m][n], float alpha, float A[m][k],
6    float B[k][n], float beta) {
7
8  #pragma hmppcg gridify (i,j), blocksize(128x1)
9    for (int i = 0; i < m; i = i + DELTA) {
10     for (int j = 0; j < n; j++) {
11       float prod[DELTA];
12 #pragma hmppcg fullunroll
13       for (int t = 0; t < DELTA; t++) {
14         prod[t] = 0;
15       }
16       for (int l = 0; l < k; l++) {
17 #pragma hmppcg fullunroll
18         for (int t = 0; t < DELTA; t++) {
19           prod[t] += A[i+t][l] * B[l][j];
20         }
21       }
22 #pragma hmppcg fullunroll
23       for (int t = 0; t < DELTA; t++) {
24         C[i+t][j] = alpha * prod[t] + beta * C[i+t][j];
25       }
26     }
27   }
28 }
```

Outline

3. Locality-Aware Automatic Parallelization for GPGPU
   - GPGPU with CUDA and OpenHMPP
   - Locality-Aware Generation of Efficient GPGPU Code
   - CONV3D & SGEMM
   - **Experimental Evaluation**

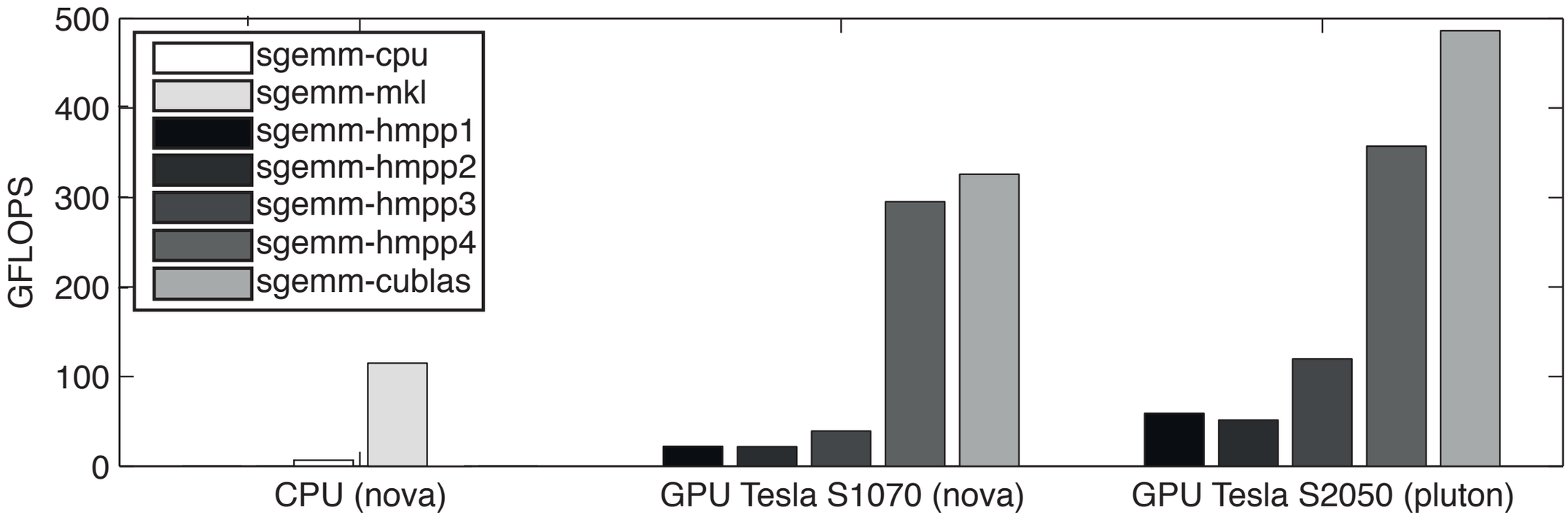
Performance Evaluation: CONV3D

$size_X$, $size_Y$ and $size_Z$ in 128, 256, 384, 512, 640 and 768



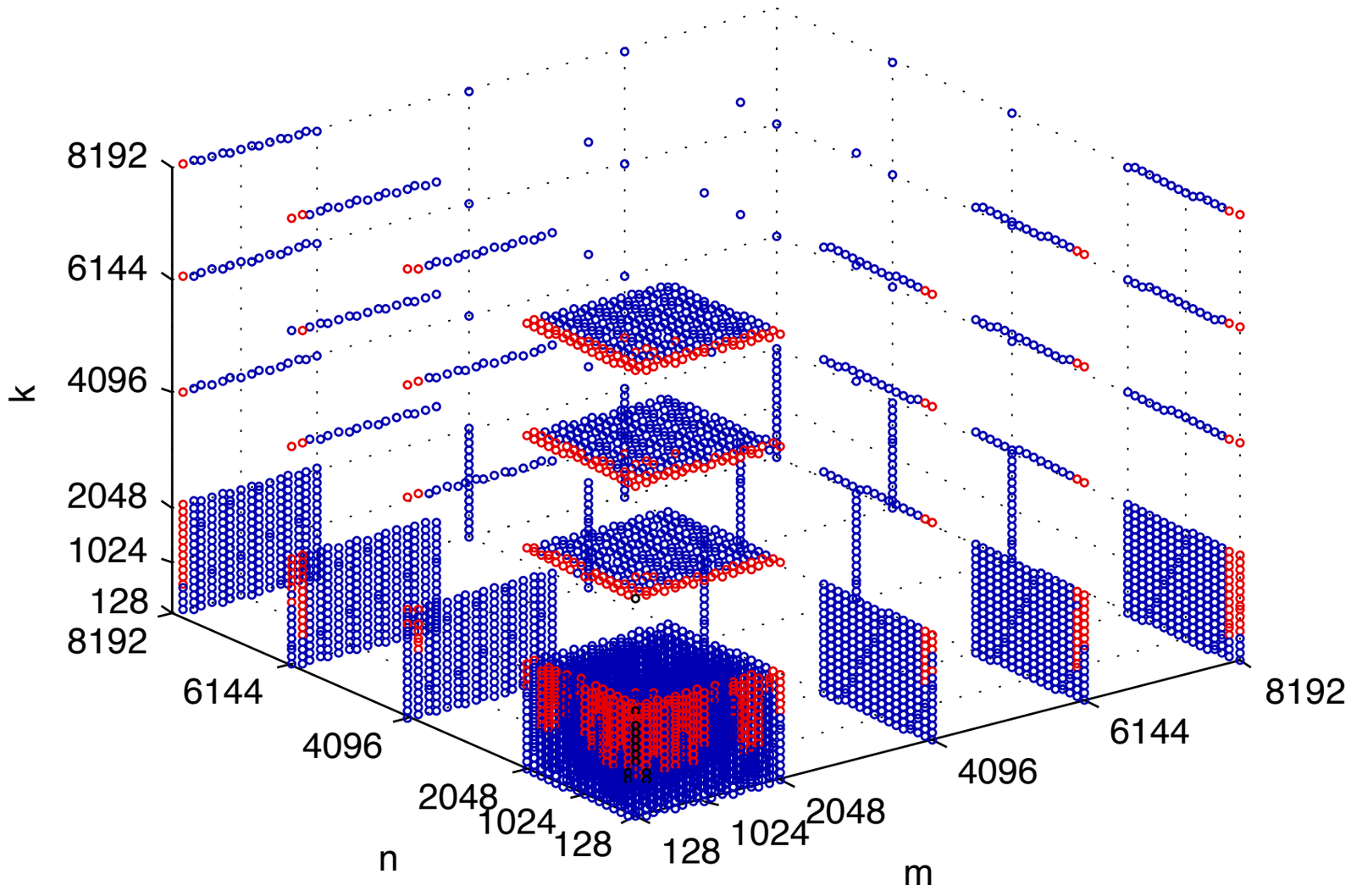
Performance Evaluation: SGEMM (I)

m, n and k in 128, 256, 384, 512, 640, 768, 896, 1024, 1152, 1280, 1408, 1536, 1664, 1792, 1920, 2048, 4096, 6144 and 8192



the biggest improvement factor is the usage of the GPU shared memory

Performance Evaluation: SGEMM (and II)



GPU Tesla S2050 (pluton)

blue: sgemm-cublas
red: sgemm-hmpp4
black: sgemm-mkl

Outline

1. Introduction
2. A Novel Compiler Support for Multicore Systems
3. Locality-Aware Automatic Parallelization for GPGPU
4. **Trace-Based Affine Reconstruction of Code**
5. Main Contributions and Future Research Lines

Outline

4. Trace-Based Affine Reconstruction of Code

   - Problem Formulation
   - Problem Resolution with CHOLESKY
   - Extensions for Supporting Nonlinear Traces
   - Experimental Evaluation

Outline

4. Trace-Based Affine Reconstruction of Code

   - **Problem Formulation**

   - Problem Resolution with CHOLESKY

   - Extensions for Supporting Nonlinear Traces

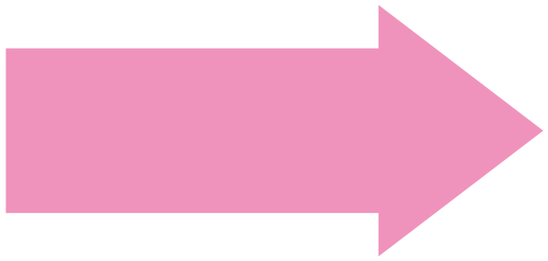
   - Experimental Evaluation

Problem Statement

```
1  | 0x1e2d140
2  | 0x1e2d140
   ⋮

30 | 0x1e2d140
31 | 0x1e2d240
32 | 0x1e2d248
33 | 0x1e2d240
34 | 0x1e2d248
   ⋮
```

```
88 | 0x1e2d248
89 | 0x1e2d340
90 | 0x1e2d348
91 | 0x1e2d350
92 | 0x1e2d340
93 | 0x1e2d348
94 | 0x1e2d350
   ⋮
```



```
1. for (i = 0; i <= 29; i++) {
2.   for (j = 0; j <= 29-i; j++) {
3.     for (k = 0; k < i; k++) {
4.       … A[i][k] …
5.     }
6.   }
7. }
```

- We assume that:
  - Addresses are generated by a single instruction
  - Instruction is enclosed in an affine loop nest
- Existing memory optimization techniques based on the polyhedral model, and any other static or dynamic optimization technique in the absence of source and/or binary code, can be applied.

Problem Formulation (I)

$$\begin{array}{l}\textbf{DO}\ \ i_1 = 0,\ \ u_1(\overrightarrow{\imath})\\ \quad \textbf{DO}\ \ i_2 = 0,\ \ u_2(\overrightarrow{\imath})\\ \qquad \vdots\\ \qquad \textbf{DO}\ \ i_n = 0,\ \ u_n(\overrightarrow{\imath})\\ \qquad\quad V[f_1(\overrightarrow{\imath})]\ldots[f_m(\overrightarrow{\imath})]\end{array}$$

$$V[f_1(\overrightarrow{\imath})]\ldots[f_m(\overrightarrow{\imath})] = V[c_0 + i_1c_1 + \ldots + i_nc_n]$$

Problem Formulation (II)

- In our model, only three possible variations of a loop index between two consecutive iterations are allowed

1. i_j does not change $\Rightarrow \delta_j^k = 0$
2. i_j is increased by one $\Rightarrow \delta_j^k = 1$
3. i_j is reset to 0 $\Rightarrow \delta_j^k = -i_j^k$

$$(\overrightarrow{\imath}^{k+1} - \overrightarrow{\imath}^{k}) = \begin{bmatrix} i_1^{k+1} - i_1^k \\ i_2^{k+1} - i_2^k \\ \vdots \\ i_n^{k+1} - i_n^k \end{bmatrix} = \begin{bmatrix} \delta_1^k \\ \delta_2^k \\ \vdots \\ \delta_n^k \end{bmatrix} = \overrightarrow{\delta}^k$$

Problem Formulation (and III)

- The stride between two consecutive accesses is a linear combination of the coefficients of the loop indices

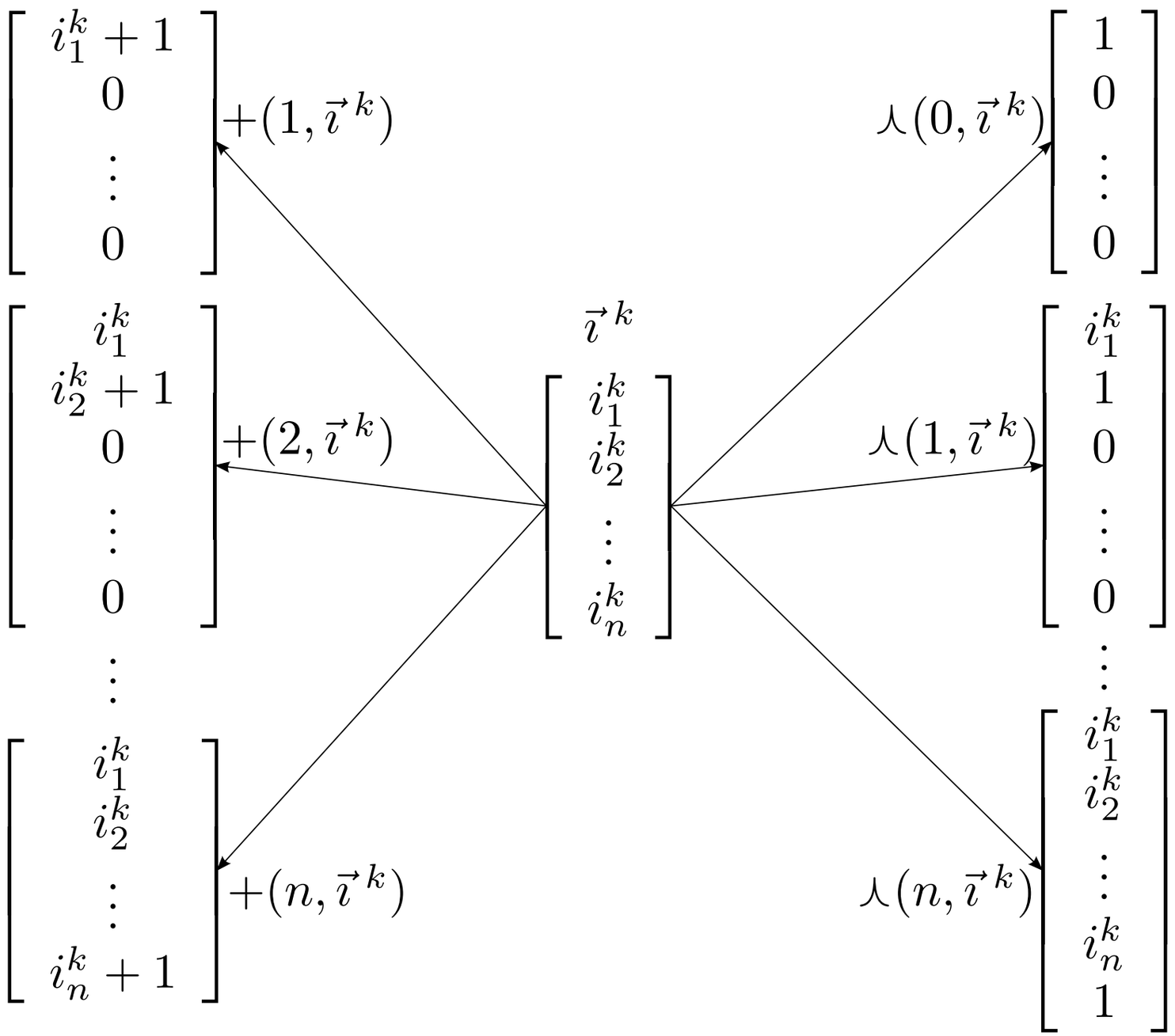
$$\sigma^k = V(\overrightarrow{i}^{k+1}) - V(\overrightarrow{i}^{k})$$

$$\begin{aligned}
\sigma^k &= V + (c_0 + c_1 i_1^{k+1} + \ldots + c_n i_n^{k+1}) - \\
&\quad V + (c_0 + c_1 i_1^{k} + \ldots + c_n i_n^{k}) = \\
&= c_1 \delta_1^k + \ldots + c_n \delta_n^k = \\
&= \overrightarrow{c}\,\overrightarrow{\delta}^k
\end{aligned}$$

Outline

4. Trace-Based Affine Reconstruction of Code

   - Problem Formulation
   - **Problem Resolution with CHOLESKY**
   - Extensions for Supporting Nonlinear Traces
   - Experimental Evaluation



Solution Space

2n + 1 candidates

Problem Resolution (I)

$$\{a_1, \ldots, a_k\}$$

$$\mathcal{S}_n^k = \{\overrightarrow{c}, \mathbf{I}^k, \mathbf{U}, \overrightarrow{w}\}$$

- Coefficients of the Loop Indices

$$\overrightarrow{c} \in \mathbb{Z}^n$$

$$u_j(\overrightarrow{\imath}) = w_j + u_{j,1}i_1 + \ldots + u_{j,(j-1)}i_{(j-1)}$$

- Iteration Indices

$$\mathbf{I}^k = [\overrightarrow{\imath}^1 | \ldots | \overrightarrow{\imath}^k] \in \mathbb{Z}^{n \times k}$$

- Bounds

$$\mathbf{U} = \begin{bmatrix} -1 & 0 & 0 & \ldots & 0 \\ u_{2,1} & -1 & 0 & \ldots & 0 \\ u_{3,1} & u_{3,2} & -1 & \ldots & 0 \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ u_{n,1} & u_{n,2} & u_{n,3} & \ldots & -1 \end{bmatrix} \qquad \overrightarrow{w} = \begin{bmatrix} w_1 \\ w_2 \\ \vdots \\ w_n \end{bmatrix}$$

$$\mathbf{U} \in \mathbb{Z}^{n \times n}$$

$$\overrightarrow{w} \in \mathbb{Z}^n$$

$$\mathbf{U}\overrightarrow{\imath} + \overrightarrow{w} \geq \overrightarrow{0}^T$$

Problem Resolution (II)

- To be a valid solution:
  - Each consecutive pair of indices must be sequential

$$\mathbf{UI}^k + \overrightarrow{w}\mathbf{1}^{1\times k} \geq \mathbf{0}^{n\times k}$$

- The observed strides are coherent with the reconstructed ones

$$\overrightarrow{c}(\overrightarrow{i}^{k+1} - \overrightarrow{i}^{k}) = \overrightarrow{c}\,\overrightarrow{\delta}^{k} = \sigma^k$$

Problem Resolution: CHOLESKY (I)

```
1  #define N 32;
2  double p[N], A[N][N], x;
3  int i, j, k;
4
5  #pragma scop
6  for (i = 0; i < N; ++i) {
7     x = A[i][i];
8     for (j = 0; j <= i - 1; ++j)
9        x = x - A[i][j] * A[i][j];
10    p[i] = 1.0 / sqrt(x);
11    for (j = i + 1; j < N; ++j) {
12       x = A[i][j];
13       for (k = 0; k <= i - 1; ++k)
14          x = x - A[j][k] * A[i][k] ;
15       A[j][i] = x * p[i];
16    }
17 }
18 #pragma endscop
```

```
1  0x1e2d140
2  0x1e2d140
   ⋮

30 0x1e2d140
31 0x1e2d240
32 0x1e2d248
33 0x1e2d240
34 0x1e2d248
   ⋮
```

```
88 0x1e2d248
89 0x1e2d340
90 0x1e2d348
91 0x1e2d350
92 0x1e2d340
93 0x1e2d348
94 0x1e2d350
   ⋮
```

$$\begin{cases} \overrightarrow{c} = [\sigma^1] = [a_2 - a_1] = [0] \\ \mathbf{I}^2 = [\overrightarrow{\imath}^1 | \overrightarrow{\imath}^2] = [0, 1] \\ \mathbf{U} = [-1] \\ \overrightarrow{w} = [1]^T \end{cases}$$

Problem Resolution: Building the System

- Calculate the observed stride

$$\sigma^k = a_{k+1} - a_k$$

- Build a diophantine linear equation system

$$\overrightarrow{c}(\overrightarrow{\imath}^{k+1} - \overrightarrow{\imath}^{k}) = \sigma^k \Rightarrow (\overrightarrow{c}^T \overrightarrow{c})\overrightarrow{\delta}^k = \overrightarrow{c}^T \sigma^k$$

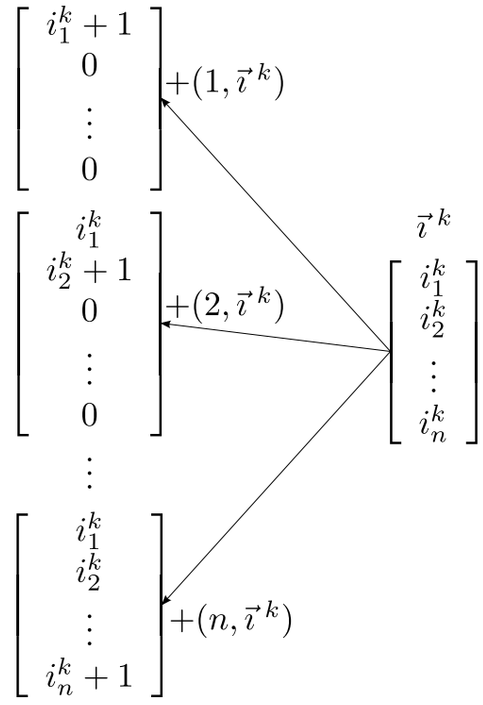
- One or more solutions: Explore them independently
- No solution under current boundaries
  - Increase dimensionality adding a new loop
  - Modify boundaries
  - Discard this branch

Problem Resolution: Solving the System

$$\vec{c}\,(\vec{\imath}^{\,k+1} - \vec{\imath}^{\,k}) = \sigma^k \Rightarrow (\vec{c}^{\,T}\vec{c}\,)\,\vec{\delta}^{\,k} = \vec{c}^{\,T}\sigma^k$$

- As indices must be sequential, there are at most *n* solutions

$$\{\vec{\imath}^{\,k+1}_l = +(l, \vec{\imath}^{\,k}), 0 < l \leq n\}$$



- We only need to calculate the predicted stride for each valid index and compare with the observed stride

$$\hat{\sigma}^k_l = \vec{c}\,\vec{\delta}^{\,k}_l$$

Problem Resolution: CHOLESKY (II)

- Solution for the first two accesses:

$$\begin{cases} \overrightarrow{c} = \left[\sigma^1\right] = [a_2 - a_1] = [0] \\ \mathbf{I}^2 = \left[\overrightarrow{\imath}^1 | \overrightarrow{\imath}^2\right] = [0, 1] \\ \mathbf{U} = [-1] \\ \overrightarrow{w} = [1]^T \end{cases}$$

- Processing the third access:

$$a_3 = \texttt{0x1e2d140}$$

$$\sigma^2 = a_3 - a_2 = \texttt{0x1e2d140} - \texttt{0x1e2d140} = 0$$

$$\hat{\sigma}_1^2 = \overrightarrow{c}\,\overrightarrow{\delta}_1^2 = [0]\,[1]^T = 0$$

$$\mathbf{I} = [\mathbf{I}| + (\mathbf{1}, \overrightarrow{\boldsymbol{\imath}}^{\mathbf{2}})] = \begin{bmatrix} 0 & 1 & \mathbf{2} \end{bmatrix}$$

- The reconstruction continues until

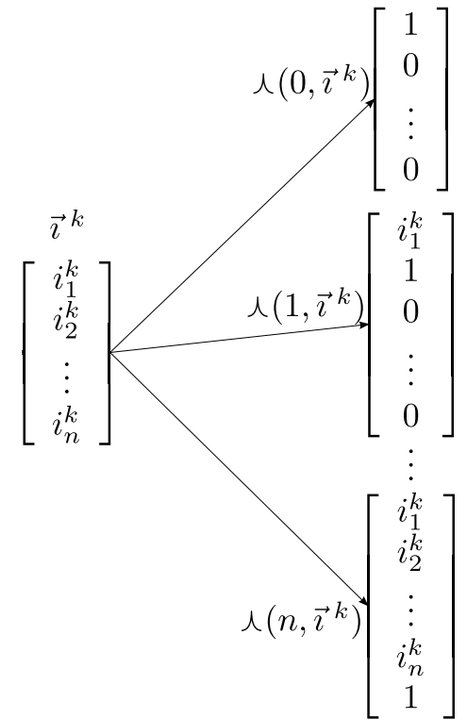
$$\sigma^{30} = a_{31} - a_{30} = \texttt{0x1e2d240} - \texttt{0x1e2d140} \Rightarrow \sigma^{30} = \texttt{0x100} = 256$$

Problem Resolution: Increasing the Solution Dimensionality (I)

- Add a new loop

$$\overrightarrow{\imath}^{k+1} = \lambda(p, \overrightarrow{\imath}^{k})$$

$$\mathbf{I}^{k+1} = \left[\begin{array}{c|c} \mathbf{I}^{k}_{(1:p,:)} & \\ \mathbf{0}^{1\times k} & \overrightarrow{\imath}^{k+1} \\ \mathbf{I}^{k}_{(p+1:n,:)} & \end{array}\right]$$



- In CHOLESKY

$$\mathbf{I} = \left[\begin{array}{ccc|c} \mathbf{0} & \dots & \mathbf{0} & \mathbf{1} \\ & \mathbf{I}_{(1:30)} & & \mathbf{0} \end{array}\right] = \left[\begin{array}{ccc|c} \mathbf{0} & \dots & \mathbf{0} & \mathbf{1} \\ 0 & \dots & 29 & \mathbf{0} \end{array}\right]$$

Problem Resolution: Increasing the Solution Dimensionality (and II)

- The coefficient for the new loop can be derived from the observed stride

$$\overrightarrow{c}(\overrightarrow{i}^{k+1} - \overrightarrow{i}^{k}) = \sigma^k \Rightarrow \left[c_1, \ldots, c_p, \boldsymbol{c'_p}, c_{p+1}, \ldots, c_n\right] \begin{bmatrix} 0 \\ \vdots \\ 0 \\ 1 \\ -i_p^k \\ \vdots \\ -i_n^k \end{bmatrix} = \sigma^k \Rightarrow \; c'_p = \sigma^k + \sum_{r=p+1}^{n} i_r^k c_r$$

- In CHOLESKY

$$c'_0 = \sigma^{30} + i_1^{30} c_1 = 256 + 0 \cdot 29 \Rightarrow \overrightarrow{c} = \begin{bmatrix} \mathbf{256} & 0 \end{bmatrix}$$

Problem Resolution: Updating the Loop Bounds

- Loop indices must be sequential and stay into loop bounds

$$\mathbf{U}'\mathbf{I}^{k+1} + \overrightarrow{w}'\mathbf{1}^{1\times(k+1)} \geq \mathbf{0}^{n\times(k+1)}$$

  - Inconsistent system ⟶ The branch is discarded

  - System with solutions ⟶ Overdetermined

$\overrightarrow{w}'$	$O(1)$
$\mathbf{U}'$	$O(n^2)$

Problem Resolution: Accelerating the Traversal

- In the general case, the complexity of exploring the solution space for a trace with *A* addresses generated by *n* loops is $O(n^A)$

$$\overrightarrow{\gamma}^k = \mathbf{U}\,\overrightarrow{\imath}^k + \overrightarrow{w}$$

- Each element indicates how many more iterations of each index are left before it resets under the bounds

- The most plausible value for the next index is $\overrightarrow{\imath}_l^{k+1} = +(l, \overrightarrow{\imath}^k)$ where *l* is the position of the innermost positive element

- Several accesses are recognized in block

Problem Resolution: CHOLESKY (III)

$$\mathbf{I} = \left[\begin{array}{ccc|c} \mathbf{0} & \dots & \mathbf{0} & \mathbf{1} \\ \hline \mathbf{I}_{(1:30)} & & & \mathbf{0} \end{array}\right] = \left[\begin{array}{ccc|c} \mathbf{0} & \dots & \mathbf{0} & \mathbf{1} \\ \hline 0 & \dots & 29 & \mathbf{0} \end{array}\right]$$

$$\mathbf{U} = \left[\begin{array}{c|c} \mathbf{-1} & \mathbf{0} \\ \hline \mathbf{0} & \mathbf{U}_{(1:1,1:1)} \end{array}\right] = \left[\begin{array}{c|c} \mathbf{-1} & \mathbf{0} \\ \hline \mathbf{0} & -1 \end{array}\right] \qquad \overrightarrow{w} = \left[\mathbf{1} | \overrightarrow{w}_{(1:1)}\right]^T = [\mathbf{1}|29]^T$$

$$\mathbf{UI} + \overrightarrow{w}\mathbf{1}^{1\times(31)} \geq \mathbf{0}^{2\times(31)}$$

$$\left[\begin{array}{cc} -1 & 0 \\ 0 & -1 \end{array}\right]\left[\begin{array}{cccc} 0 & \dots & 0 & 1 \\ 0 & \dots & 29 & 0 \end{array}\right] + \left[\begin{array}{c} 1 \\ 29 \end{array}\right]\left[\begin{array}{ccc} 1 & \dots & 1 \end{array}\right] =$$

$$\left[\begin{array}{cccccc} 0 & 0 & 0 & \dots & 0 & -1 \\ 0 & -1 & -2 & \dots & -29 & 0 \end{array}\right] + \left[\begin{array}{ccc} 1 & \dots & 1 \\ 29 & \dots & 29 \end{array}\right] =$$

$$\left[\begin{array}{cccccc} 1 & 1 & 1 & \dots & 1 & 0 \\ 29 & 28 & 27 & \dots & 0 & 29 \end{array}\right] \geq \mathbf{0}^{2\times(31)}$$

Problem Resolution: CHOLESKY (and IV)

```
1  #define N 32;
2  double p[N], A[N][N], x;
3  int i, j, k;
4
5  #pragma scop
6  for (i = 0; i < N; ++i) {
7     x = A[i][i];
8     for (j = 0; j <= i - 1; ++j)
9        x = x - A[i][j] * A[i][j];
10    p[i] = 1.0 / sqrt(x);
11    for (j = i + 1; j < N; ++j) {
12       x = A[i][j];
13       for (k = 0; k <= i - 1; ++k)
14          x = x - A[j][k] * A[i][k] ;
15       A[j][i] = x * p[i];
16    }
17 }
18 #pragma endscop
```

$$A[i][k] \longrightarrow A[256 * i + 8 * k]$$

$$\overrightarrow{c} = \begin{bmatrix} 256 & 0 & 8 \end{bmatrix}$$

$$\mathbf{U} = \begin{bmatrix} -1 & 0 & 0 \\ -1 & -1 & 0 \\ 1 & 0 & -1 \end{bmatrix}$$

$$\overrightarrow{w} = \begin{bmatrix} 29 & 29 & 0 \end{bmatrix}$$

Outline

4. Trace-Based Affine Reconstruction of Code

   - Problem Formulation
   - Problem Resolution with CHOLESKY
   - **Extensions for Supporting Nonlinear Traces**
   - Experimental Evaluation

Supporting Nonlinearity: Input Noise

- Some trace files mainly contain references issued by a single access, but mixed with unrelated ones (e.g., nearly affine or unlabeled traces)

- The exploration of the solution space can be modified to discard until *max* observed accesses

$$\left\{ \hat{\sigma}^k = \sum_{r=0}^{e} \sigma^{k+r}, 0 < e \leq max \right\}$$

- Tolerance parameter for discarding a branch

Supporting Nonlinearity: Missing Data

- A trace file may be missing some data to make it completely representable by an affine loop

- The exploration of the solution space can be modified to insert until *max* predicted strides

$$\left\{ \sigma^k = \sum_{r=0}^{e} \hat{\sigma}^{k+r}, 0 < e \leq max \right\}$$

- Tolerance parameter to avoid the exploration of improbable branches

- Particular case: access guarded by a boolean function

Supporting Nonlinearity: Automatically Parallelized Codes with PLUTO

- Codes parallelized by simply adding a OpenMP `parallel` pragma & the iterations are scheduled statically
  - The same algorithm can be applied.
  - Otherwise
    - Piecewise reconstruction as sequence of perfectly nested loops

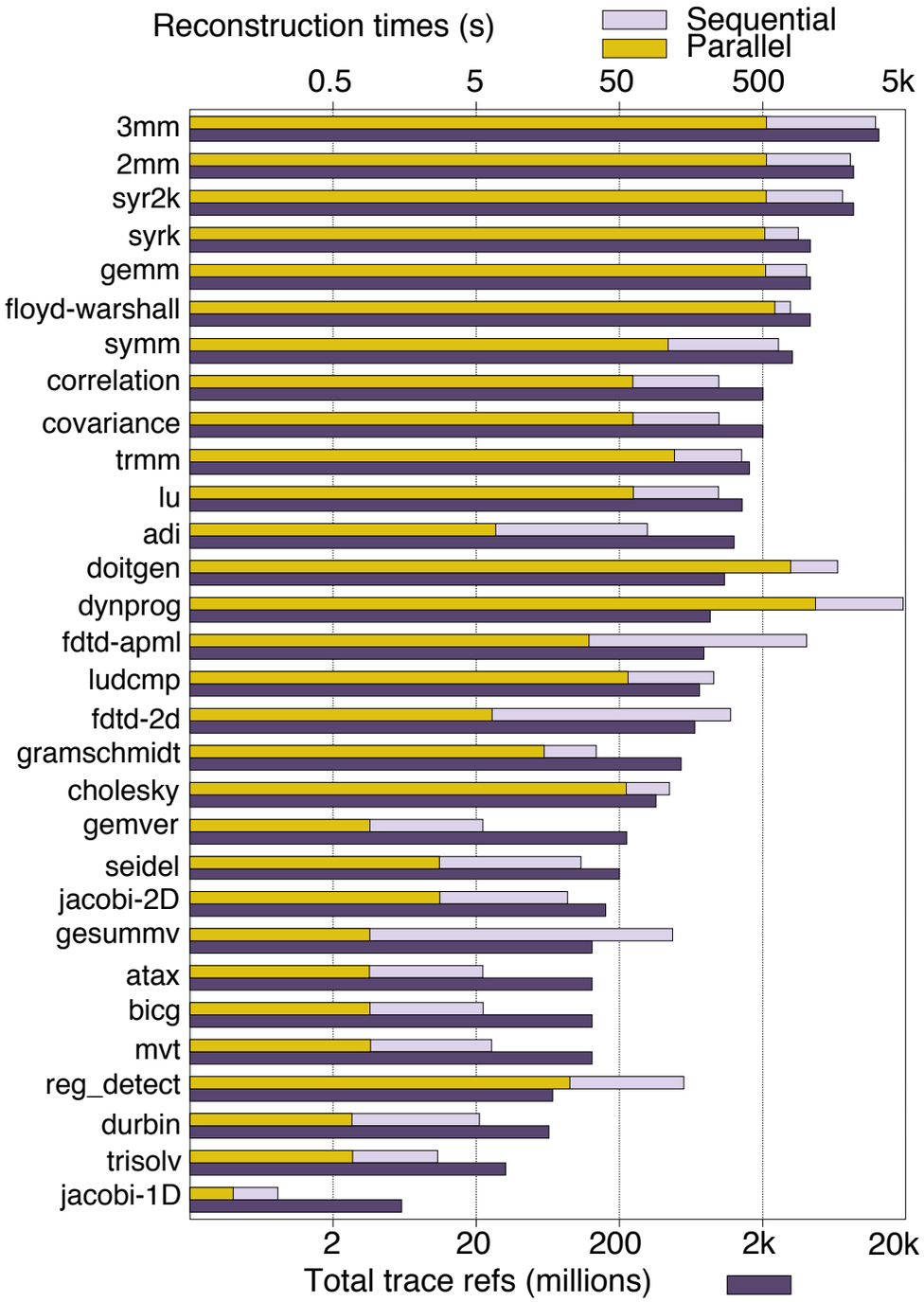
1: **FUNCTION** PIECEWISEEXTRACT
Input: $\overrightarrow{a}$: the execution trace
Input: max_depth: maximum reconstruction depth
Output: $\Omega = \{\mathcal{S}_0, \ldots, \mathcal{S}_{L-1}\}$: set of perfectly nested affine loops that form a piecewise reconstruction of $\overrightarrow{a}$
2: $\Omega \leftarrow$ PIECEWISEEXTRACT($\overrightarrow{a}, depth = 1$)
3: $curr\_depth \leftarrow 2$
4: **while** $(curr\_depth \leq max\_depth) \wedge (|\Omega| > 1)$ **do**
5: **for** $\mathcal{S}_l \in \Omega$ **do**
6: $\mathcal{S}'_l \leftarrow$ EXTRACT($\mathcal{S}_l, \overrightarrow{a}, depth = curr\_depth$)
7: **if** $\mathcal{S}'_l$ overlaps perfectly with $\{\mathcal{S}_l, \ldots, \mathcal{S}_{l'}\} \in \Omega$ **then**
8: $\Omega \leftarrow (\Omega - \{\mathcal{S}_l, \ldots, \mathcal{S}_{l'}\}) \cup \mathcal{S}'_l$
9: **end if**
10: $curr\_depth++$
11: **end for**
12: **end while**
13: **return** Ω
14: **end FUNCTION**=0

Outline

4. Trace-Based Affine Reconstruction of Code

   - Problem Formulation
   - Problem Resolution with CHOLESKY
   - Extensions for Supporting Nonlinear Traces
   - **Experimental Evaluation**

Experimental Evaluation: Affine Codes



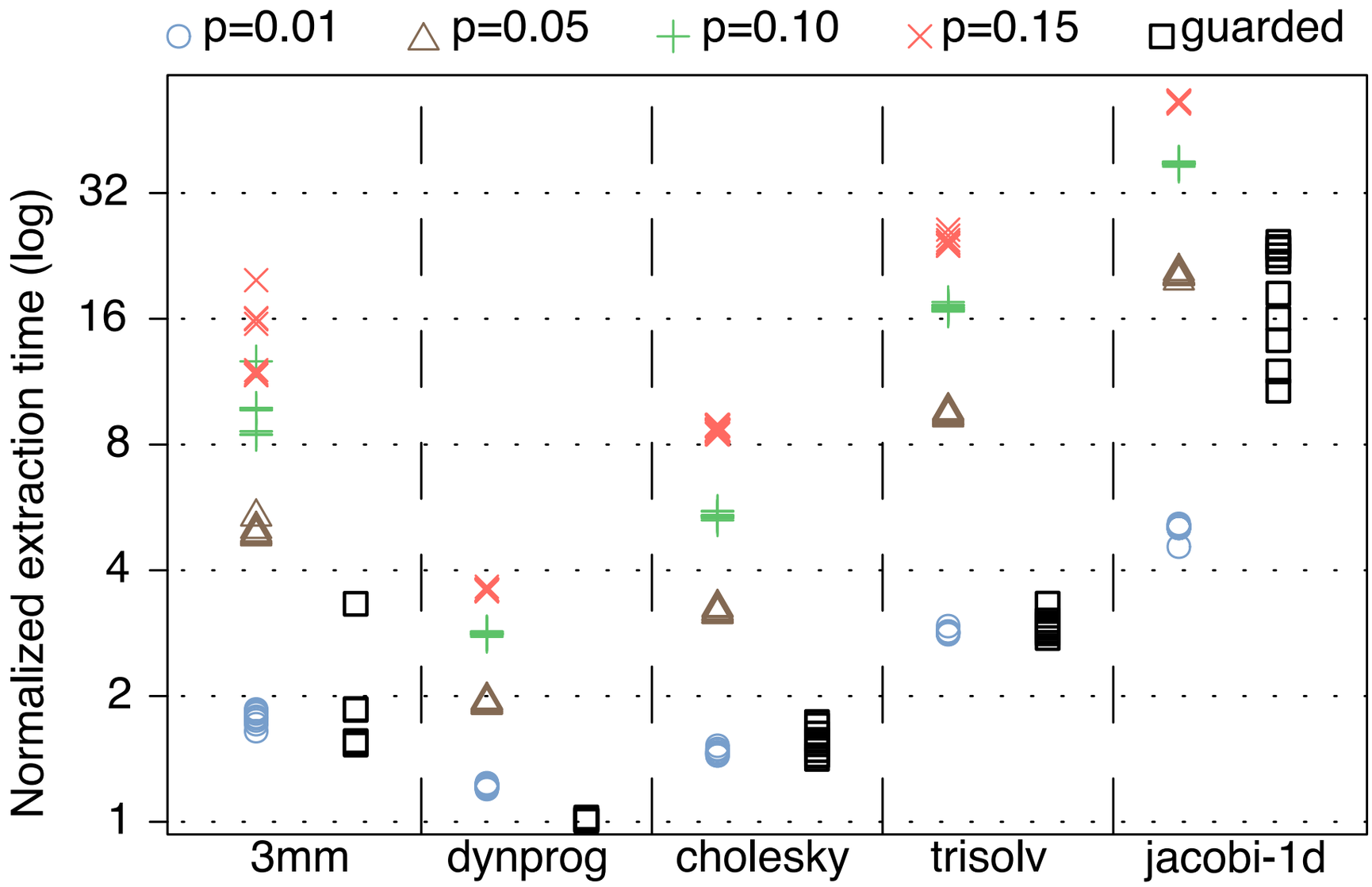
Trace	%	Trace	%	Trace	%
3mm	0.02	lu	0.11	seidel	0.00
2mm	0.04	adi	0.01	jac-2D	0.00
syr2k	0.02	doit.	0.58	gesum.	25.01
syrk	0.05	dynp.	0.00	atax	25.00
gemm	0.05	fdtd-a.	24.21	bicg	25.00
floyd	0.00	lud.	0.66	mvt	12.50
symm	0.13	fdtd-2d	0.01	reg_d.	2.07
corr.	0.67	grams.	0.58	durbin	100
covar.	0.37	chol.	0.58	trisolv	100
trmm	0.00	gemv.	21.43	jac-1D	100

% of trace reconstructed without gamma in 48h

Trace	%	Trace	%	Trace	%
3mm	99.85	lu	99.71	seidel	95.00
2mm	99.84	adi	98.00	jac-2D	95.00
syr2k	99.85	doit.	98.83	gesum.	74.95
syrk	99.83	dynp.	99.98	atax	74.96
gemm	99.83	fdtd-a.	75.62	bicg	74.96
floyd	99.88	lud.	99.99	mvt	87.46
symm	99.80	fdtd-2d	98.00	reg_d.	99.78
corr.	99.60	grams.	99.61	durbin	99.88
covar.	99.70	chol.	99.99	trisolv	99.89
trmm	99.97	gemv.	78.53	jac-1D	99.00

% of accesses predicted by gamma

Experimental Evaluation: Input Noise & Missing Data



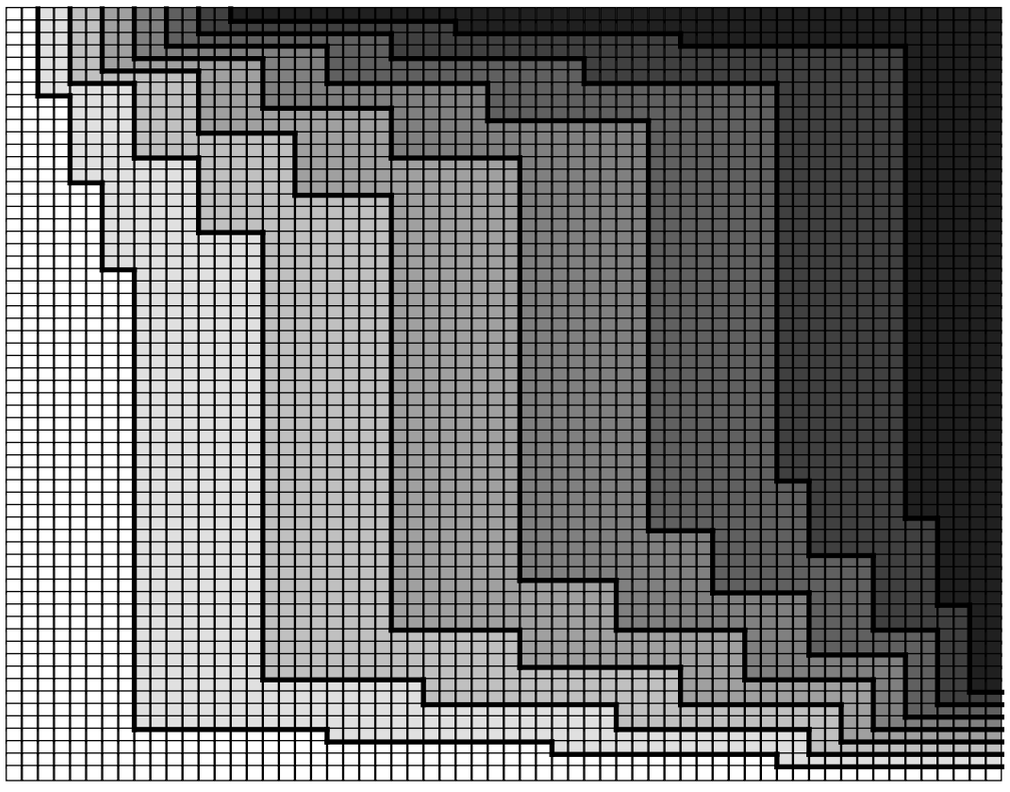
Experimental Evaluation: Piecewise Reconstruction (I)

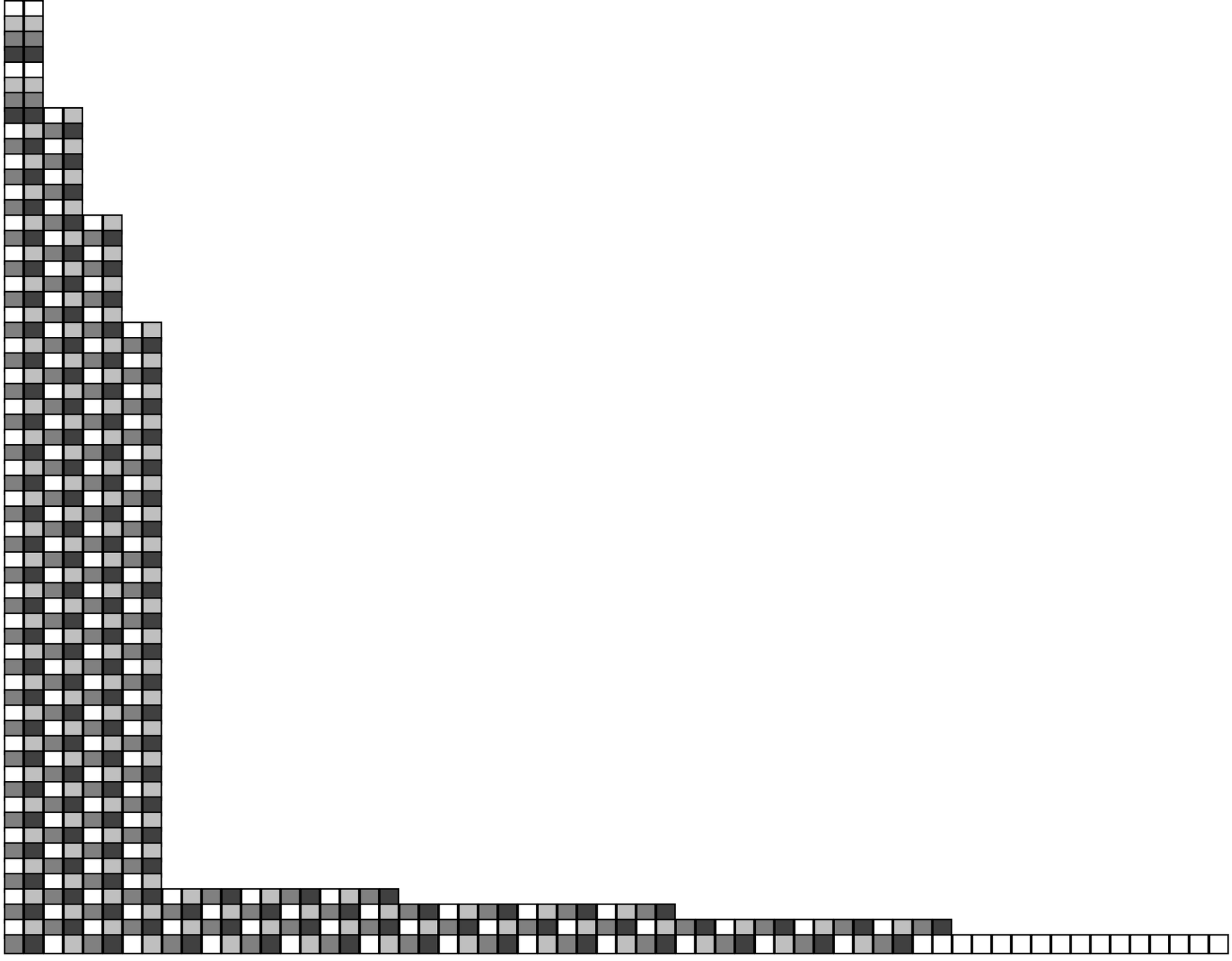
```
1 for (i=1;i<=N-2;i++)
2   for (j=1;j<=N-2;j++)
3     a[i][j] = ...
```

seidel

(from the PLUTO examples)

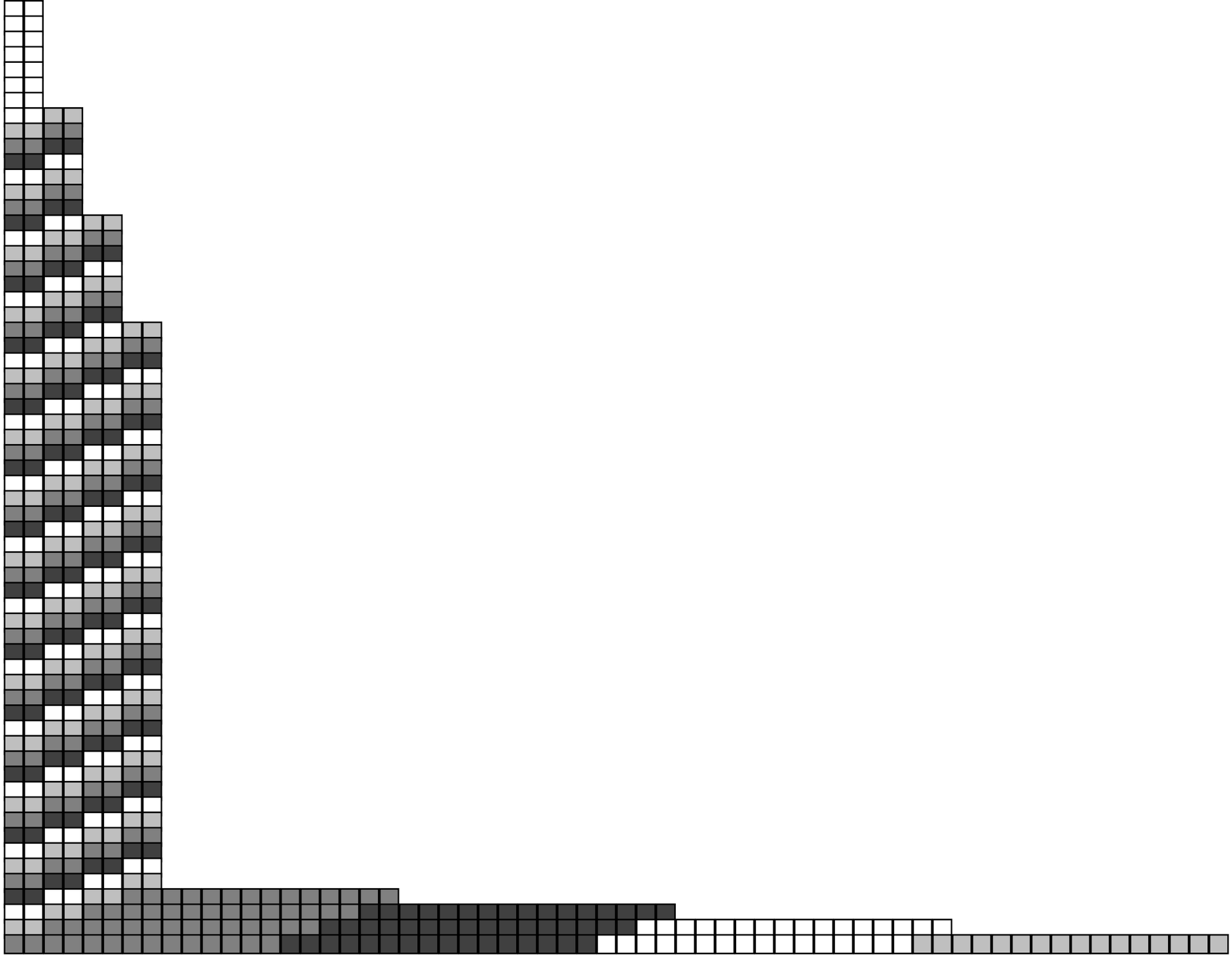
```
1 for (t1=3;t1<=3*N-6;t1++) {
2   lbp=max(ceild(t1+1,2),t1-N+2);
3   ubp=min(floord(t1+N-2,2),t1-1);
4 #pragma omp parallel for
5   for (t2=lbp;t2<=ubp;t2++)
6     a[(t1-t2)][(-t1+2*t2)] = ...
```





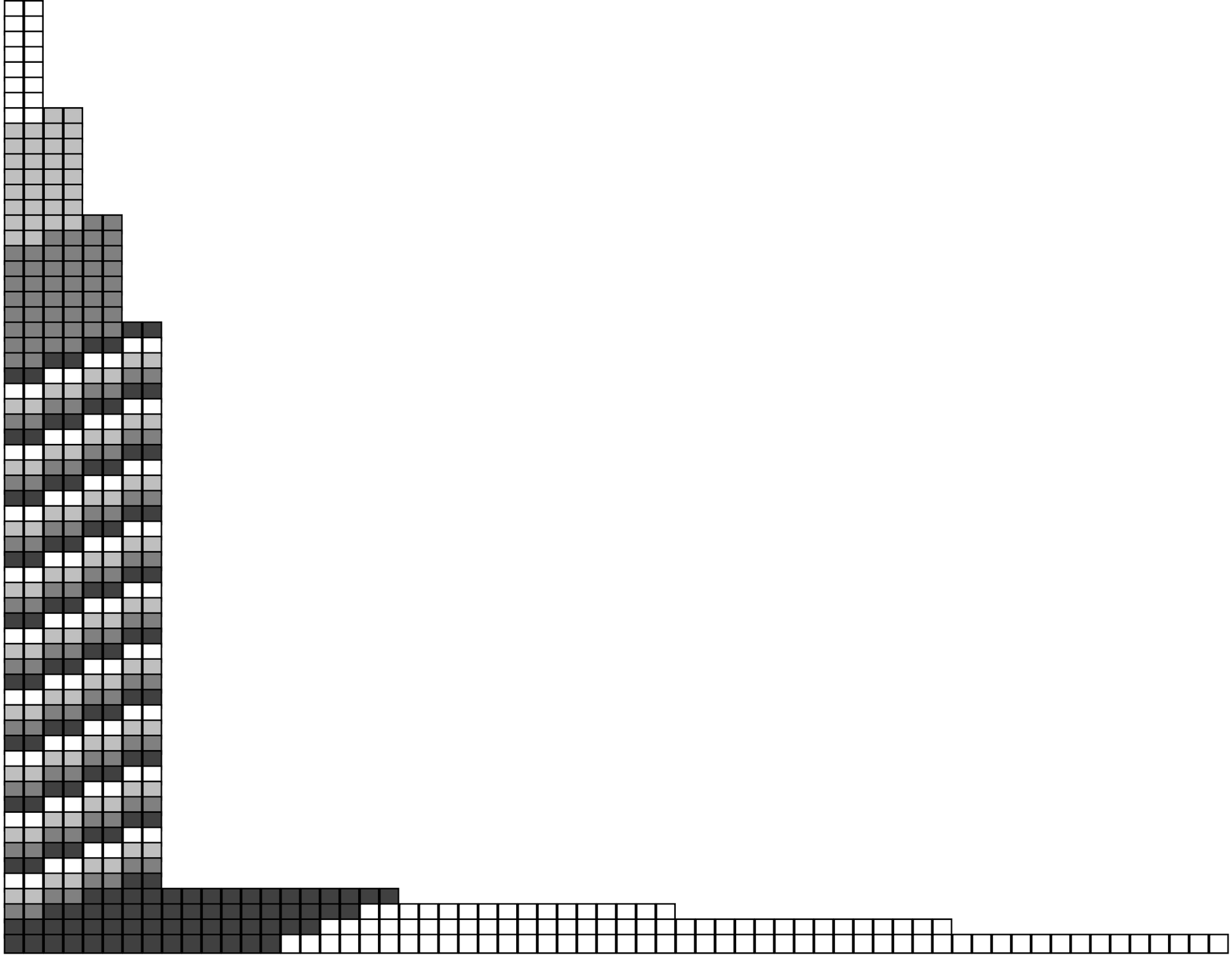
Reconstructed Trace Pieces of *seidel* for the Thread #0

max_depth = 1 (161 pieces)



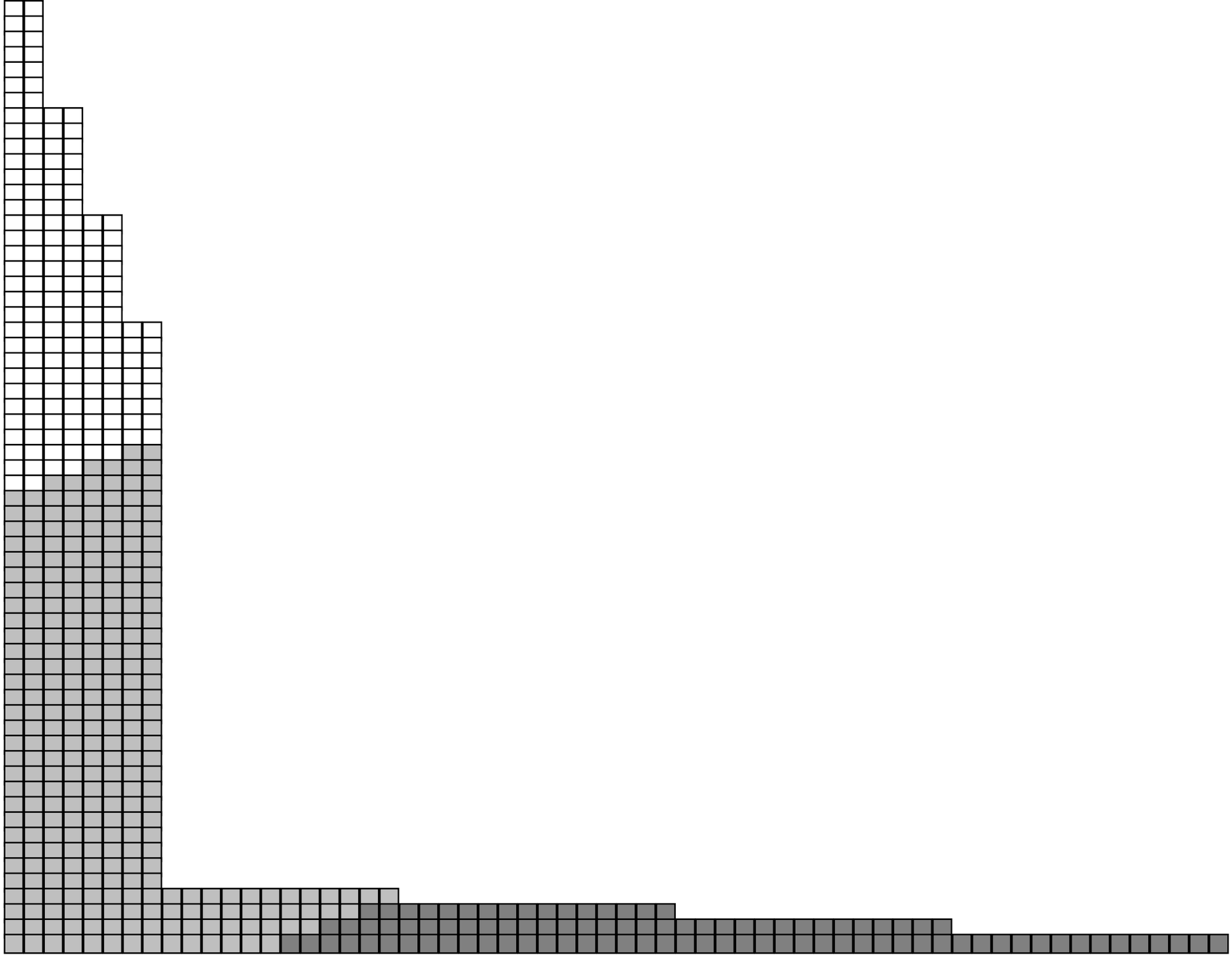
Reconstructed Trace Pieces of *seidel* for the Thread #0

max_depth = 2 (58 pieces)



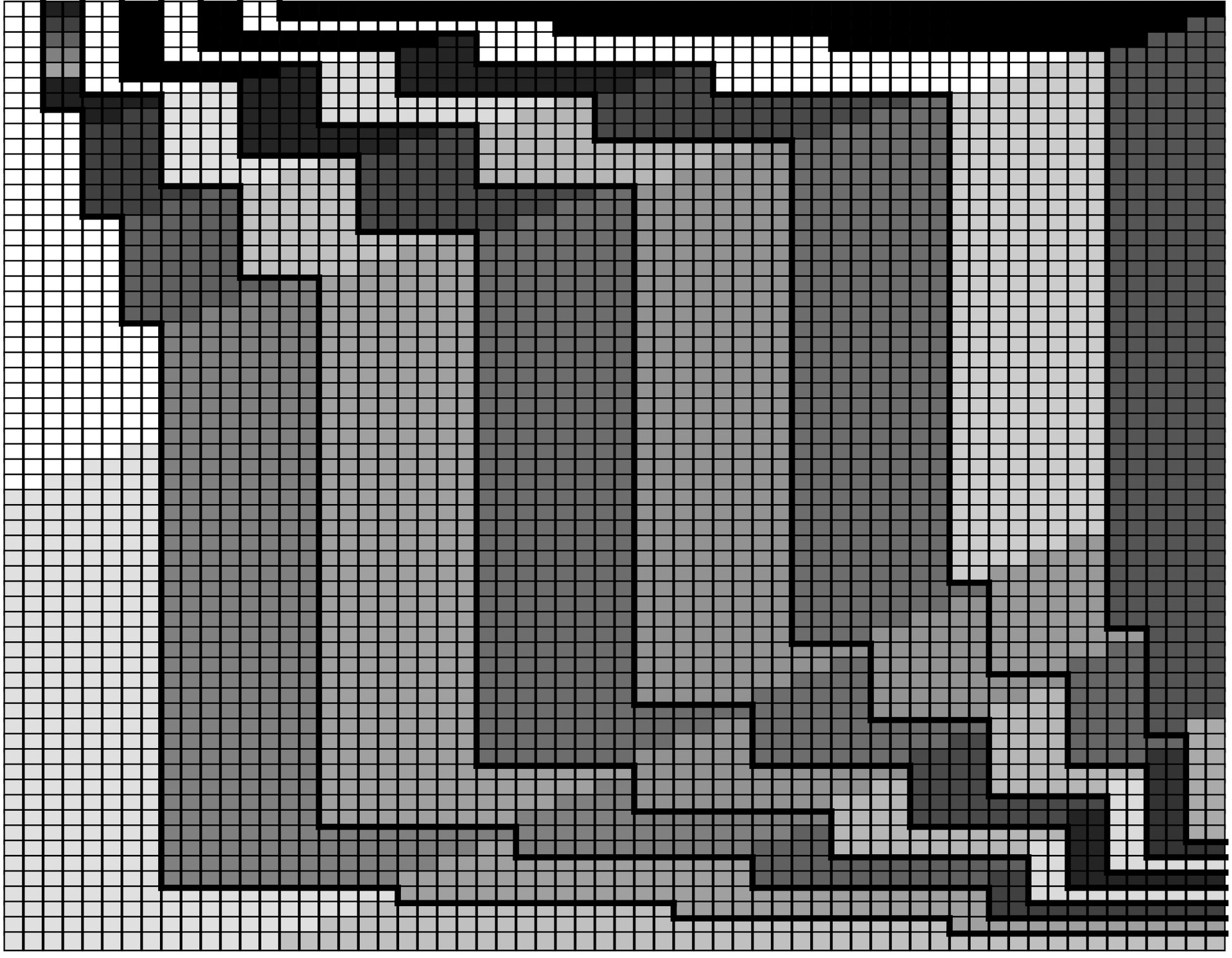
Reconstructed Trace Pieces of *seidel* for the Thread #0

max_depth = 3 (41 pieces)



Reconstructed Trace Pieces of *seidel* for the Thread #0

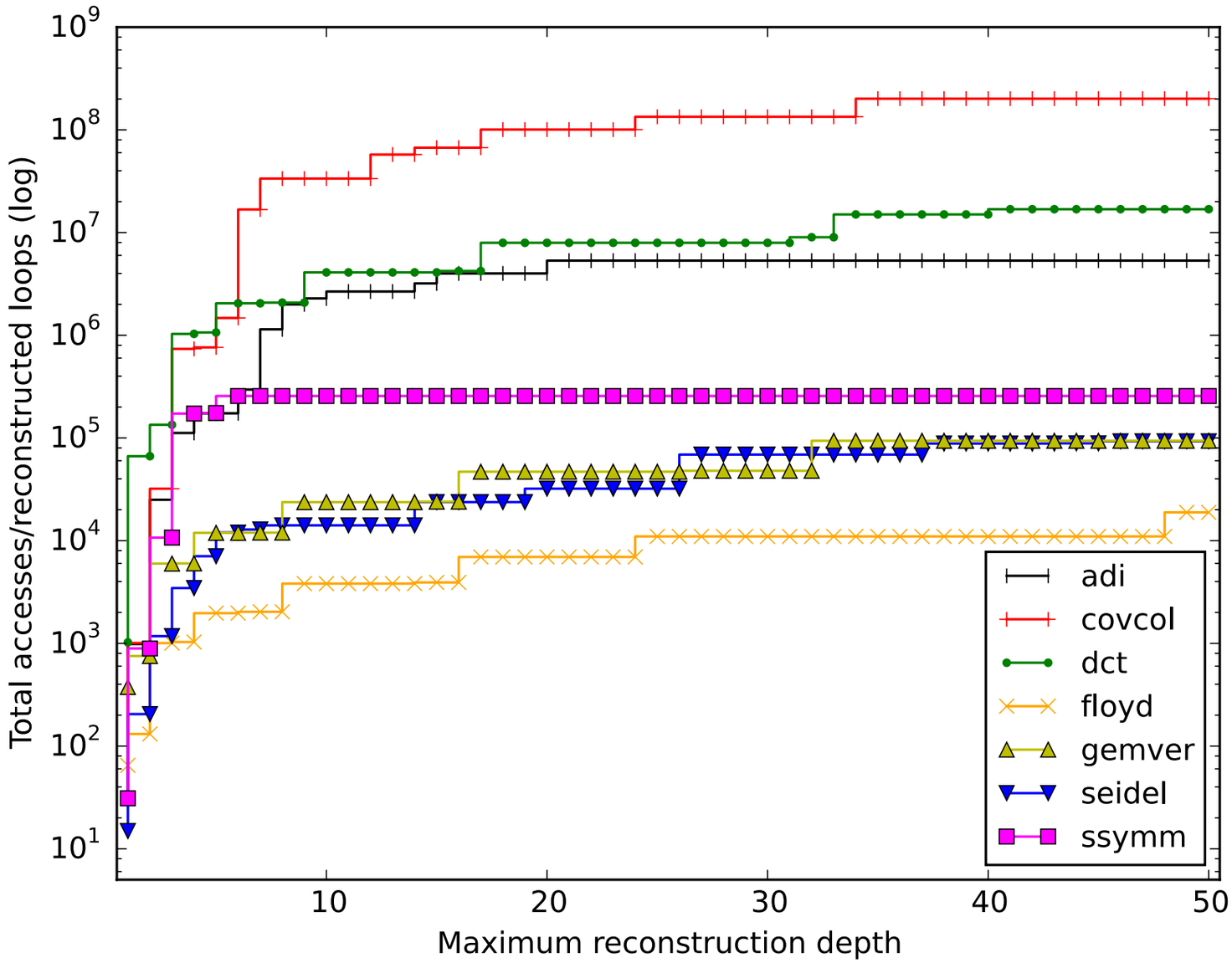
max_depth = 4 (3 pieces)



Reconstructed Trace Pieces of *seidel* for All the Threads

max_depth = 4

Experimental Evaluation: Piecewise Reconstruction (and VII)



increasing the maximum depth has diminishing returns

a small number of loops represent most of the issued accesses

Outline

1. Introduction
2. A Novel Compiler Support for Multicore Systems
3. Locality-Aware Automatic Parallelization for GPGPU
4. Trace-Based Affine Reconstruction of Code
5. **Conclusions**

Main Contributions (I)

Definition of a new compiler intermediate representation called KIR

- It provides the program characteristics needed for the automatic parallelization of the input sequential code
- It is built on top of diKernels to handle syntactical variations of the source code
- diKernels are connected with diKernel-level dependences and are grouped into execution scopes in order to recognize the computational stages of the input application

Main Contributions (II)

Generation of parallel code for multicore processors with the insertion of OpenMP directives

- Automatic partitioning algorithm of the KIR focused on the minimization of the overhead of thread synchronization

- Comprehensive benchmark suite that includes synthetic codes representative of frequently used diKernels, routines from dense/sparse linear algebra and image processing, and simulation applications

- Comparative evaluation in terms of effectiveness with GCC, ICC and PLUTO. The contenders fail to parallelize codes that contain both regular computations with complex control flows and irregular computations, and they do not optimize the joint parallelization of multiple loops

Main Contributions (III)

KIR-based locality-aware automatic parallelization technique that targets GPU-based heterogeneous systems

- It exploits data locality in the complex GPU memory hierarchy
- Tested with two representative case studies: CONV3D & SGEMM
- Chains of recurrences model accesses to n-dimensional arrays
- OpenHMPP directives enabled a great understandability and portability of the generated GPU code
- Performance evaluation on NVIDIA GPUs (with two different core architectures) has corroborated its effectiveness

Main Contributions (and IV)

Reconstruction of affine loop codes from their memory traces, considering one instruction at a time

- Formulated as the exploration of a tree-like solution space
- Large traces are processed in a matter of minutes, without user intervention or access to source/binary codes
- Extensions to deal with moderate nonlinearity in the trace and with automatic parallelized codes
- Applications such as trace compression/storage/communication, dynamic parallelization, memory placement and memory hierarchy design

Future Research Lines

- Using the trace-based reconstruction to increase the information available for the construction of the KIR

- New automatic partitioning algorithm of the KIR that handles the interactions between computations for heterogeneous clusters, considering both CPU-GPU interaction and inter-node communication

- Auto-tuning to select the best performant variant between several candidates of a parallelized diKernel

- Reconstructing the memory trace of a broader range of irregular computations

Publications (I)

- J. M. Andión, M. Arenaz, G. Rodríguez, and J. Touriño. A novel compiler support for automatic parallelization on multicore systems. *Parallel Computing*, 39(9):442–460, 2013. **[Q1 (15/102) in Computer Science, Theory & Methods in JCR 2013]**

- J. M. Andión, M. Arenaz, G. Rodríguez, and J. Touriño. A parallelizing compiler for multicore systems. In *Proceedings of the 17th International Workshop on Software and Compilers for Embedded Systems (SCOPES)*, pages 138–141, Sankt Goar, Germany, 2014. **[Type A in CORE2014]**

- J. M. Andión, M. Arenaz, and J. Touriño. Domain-independent kernel-based intermediate representation for automatic parallelization of sequential programs. In *Poster Abstracts of the 6th International Summer School on Advanced Computer Architecture and Compilation for High-Performance and Embedded Systems (ACACES)*, pages 71–74, Terrasa, Spain, 2010.

- J. M. Andión, M. Arenaz, and J. Touriño. Automatic partitioning of sequential applications driven by domain-independent kernels. In *Proceedings of the 15th Workshop on Compilers for Parallel Computing (CPC)*, CDROM, Vienna, Austria, 2010.

- J. M. Andión, M. Arenaz, and J. Touriño. A new intermediate representation for GCC based on the XARK compiler framework. In *Proceedings of the 2nd International Workshop on GCC Research Opportunities (GROW) (in conjunction with HiPEAC)*, pages 89–100, Pisa, Italy, 2010.

Publications (II)

- J. M. Andión, M. Arenaz, F. Bodin, G. Rodríguez, and J. Touriño. Locality-aware automatic parallelization for GPGPU with OpenHMPP directives. *International Journal of Parallel Programming* (in press), 2015. **[Q4 (79/102) in Computer Science, Theory & Methods en JCR 2013]**

- J. M. Andión, M. Arenaz, F. Bodin, G. Rodríguez, and J. Touriño. Locality-aware automatic parallelization for GPGPU with OpenHMPP directives. In *Proceedings of the 7th International Symposium on High-level Parallel Programming and Applications (HLPP)*, pages 217–238, Amsterdam, Netherlands, 2014. **[Type C in CORE2014]**

Publications (and III)

- G. Rodríguez, J. M. Andión, M. T. Kandemir, and J. Touriño. Trace-based Affine Reconstruction of Codes. In *Proceedings of the 14th International Symposium on Code Generation and Optimization (CGO)*, (accepted), Barcelona, Spain, 2016. **[Type A in CORE2014]**

- G. Rodríguez, J. M. Andión, J. Touriño, and M. T. Kandemir. Reconstructing affine codes from their memory traces. *Pennsylvania State University Technical Report CSE 15-001*, University Park, PA, USA, 2015.

Compilation Techniques for Automatic Extraction of Parallelism and Locality in Heterogeneous Architectures

José M. Andión

PhD Advisors: Gabriel Rodríguez and Manuel Arenaz

COMPUTER ARCHITECTURE GROUP
UNIVERSITY OF A CORUÑA
GAC.UDC.ES

UNIVERSIDADE DA CORUÑA