Software Performance Optimisation Group

Imperial College London

Generative and adaptive methods in performance programming

$f \circ g$

$p ; q$

$p \mid q$ $p \parallel q$

$p(q)$ $p<q>$

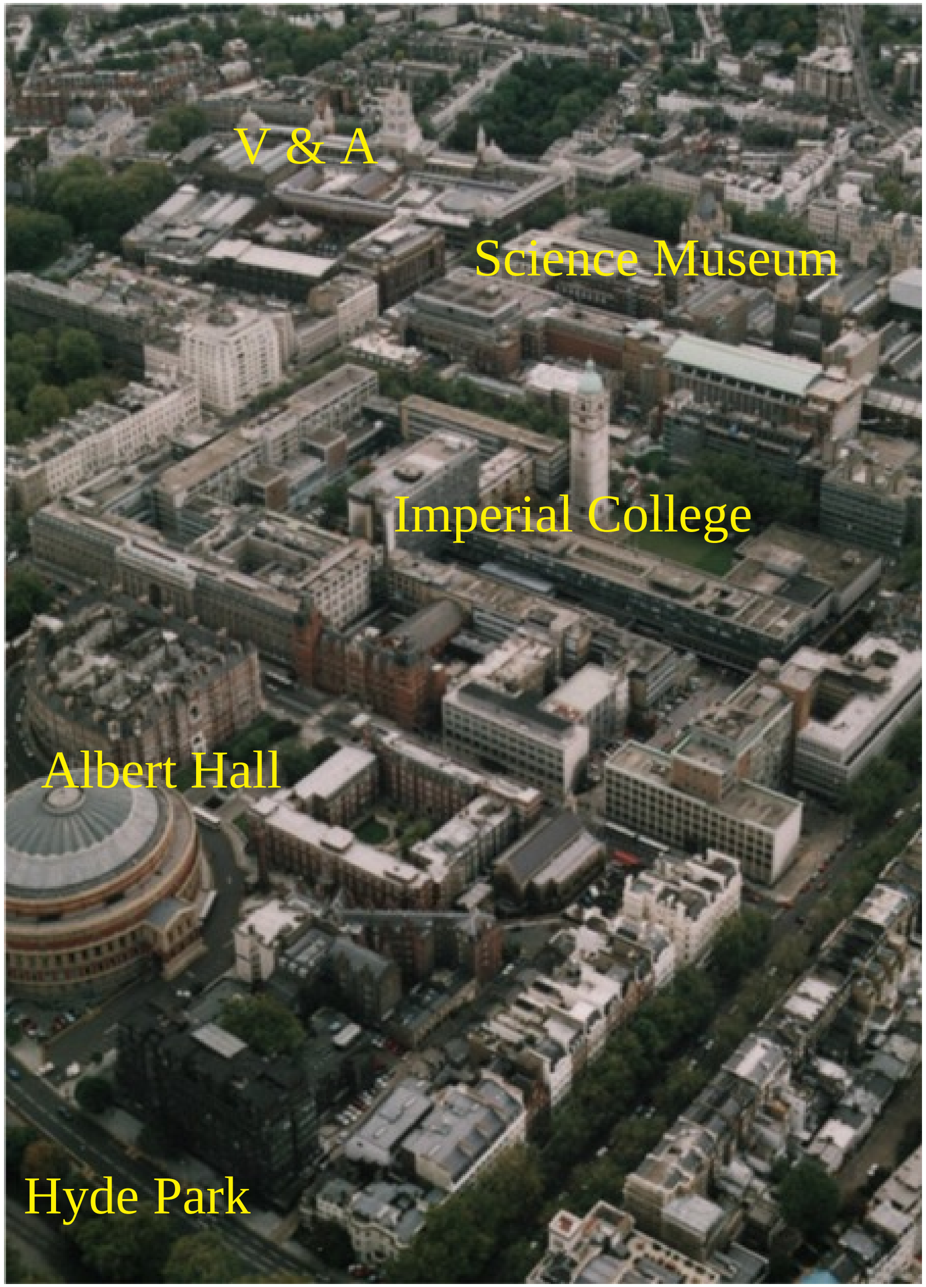
Paul H J Kelly

Imperial College London

Joint work with Olav Beckmann

Including contributions from Tony Field and numerous students

Where we're coming from…



- I lead the Software Performance Optimisation group at Imperial College, London
- Stuff I'd love to talk about another time:
  - Run-time code-motion optimisations across network boundaries in Java RMI
  - Bounds-checking for C, links with unchecked code
  - Is Morton-order layout for 2D arrays competitive?
  - Domain-specific optimisation frameworks
  - Domain-specific profiling
  - Proxying in CC-NUMA cache-coherence protocols – adaptive randomisation and combining

Performance programming

- Performance programming is the discipline of software engineering in its application to achieving performance goals
- This talk aims to review a selection of performance programming techniques we have been exploring

Construction

- What is the role of constructive methods in performance programming?
- **“by construction”**
- **“by design”**
- How can we build performance into a software project?
- How can we build-in the means to detect and correct performance problems?
- As early as possible
- With minimal disruption to the software’s long-term value?

- “In constructive logic, we can synthesize correct programs by expressing the specification as a formula, and proving it. We call this style of programming constructive”

  (Sato Masahiko / Kameya Yukiyoshi, *Constructive Programming based on SST/Λ*, IPSJ SIGNotes Software Foundation Abstract No.031 - 006)

Abstraction

- Most performance improvement opportunities come from adapting components to their context
- So the art of performance programming is to figure out how to design and compose components so this doesn't happen
- Most performance improvement measures break abstraction boundaries
- This talk is about two ideas which can help:
  - Run-time program generation (and manipulation)
  - Metadata, characterising data structures, components, and their dependence relationships

Abstraction

- Most performance improvement opportunities come from adapting components to their context
- So the art of performance programming is to figure out how to design and compose components so this doesn't happen
- Most performance improvement measures break abstraction boundaries
- This talk is about two ideas which can help:
  - Run-time program generation (and manipulation)
  - Metadata, characterising data structures, components, and their dependence relationships

Abstraction

- Most performance improvement opportunities come from adapting components to their context
- So the art of performance programming is to figure out how to design and compose components so this doesn't happen
- Most performance improvement measures break abstraction boundaries
- This talk is about two ideas which can help:
  - Run-time program generation (and manipulation)
  - Metadata, characterising data structures, components, and their dependence relationships

Abstraction

- Most performance improvement opportunities come from adapting components to their context
- So the art of performance programming is to figure out how to design and compose components so this doesn't happen
- Most performance improvement measures break abstraction boundaries
- This talk is about two ideas which can help:
  - Run-time program generation (and manipulation)
  - Metadata, characterising data structures, components, and their dependence relationships

Abstraction

- Most performance improvement opportunities come from adapting components to their context
- So the art of performance programming is to figure out how to design and compose components so this doesn't happen
- Most performance improvement measures break abstraction boundaries
- This talk is about two ideas which can help:
  - Run-time program generation (and manipulation)
  - Metadata, characterising data structures, components, and their dependence relationships

- This talk:
  - Communication fusion
  - Alignment in parallel BLAS
  - Partial evaluation/specialisation
  - Adapting to platform/resources
  - Cross-component loop fusion

Adapting to context

Dependence metadata

Performance metadata

Component model to support composition-time adaptation

Adaptation #1: Communication fusion

```
double s1, s2;
```

Component #1

```
void sum( double& data ) {
  double r = 0.0 ; …
  for (j=jmin;j<=jmax;j++) {
    r += data[j] ;
  }
  MPI_Allreduce(&r,&s1,1,MPI_SUM,...);
}
```

Component #2

```
void sumsq( double& data ) {
  double r = 0.0 ; …
  for (j=jmin;j<=jmax;j++) {
    r += data[j]*data[j] ;
  }
  MPI_Allreduce(&r,&s2,1,...);
}
```

- Example: calculating variance of distributed vector “data”

Component composition

```
double a[…][…],var[…] ;
for( i=0; i<N; i++ ) {
  sum(a[i]) ;
  sumSq(a[i]) ;
  var[i] = (s2-s1*s1/N)/(N-1);
}
```

Adaptation #1: Communication fusion

```
double rVec[2];
```

Component #1

```
void sum( double& data ) {
  double r = 0.0 ; …
  for (j=jmin;j<=jmax;j++) {
    r += data[j] ;
  }
 rVec[0] = r;
}
```

Component #2

```
void sumsq( double& data ) {
  double r = 0.0 ; …
  for (j=jmin;j<=jmax;j++) {
    r += data[j]*data[j] ;
  }
 rVec[1]= r;
}
```

- Example: calculating variance of distributed vector “data”

Component composition

```
double a[…][…],var[…] ;
for( i=0; i<N; i++ ) {
  sum(a[i]) ;
  sumSq(a[i]) ;
  MPI_Allreduce(&rVec,&s,2,
                MPI_SUM,..) ;
  var[i] = (s2-s1*s1/N)/(N-1);
}
```

- For N=3000 fusing MPI Allreduces improved performance on linux cluster by 48.7%

Adaptation #1: Communication fusion

```
CFL_Double s1(0), s2(0) ;
```

Shared variable declaration

Component #1

```
void sum( double& data ) {
  s1 = 0.0 ; …
  for (j=jmin;j<=jmax;j++) {
    s1 += data[j] ;
  }
}
```

Global reduction

Component #2

```
void sumsq( double& data ) {
  s2 = 0.0 ; …
  for (j=jmin;j<=jmax;j++) {
    s2 += data[j]*data[j] ;
  }
}
```

Global reduction

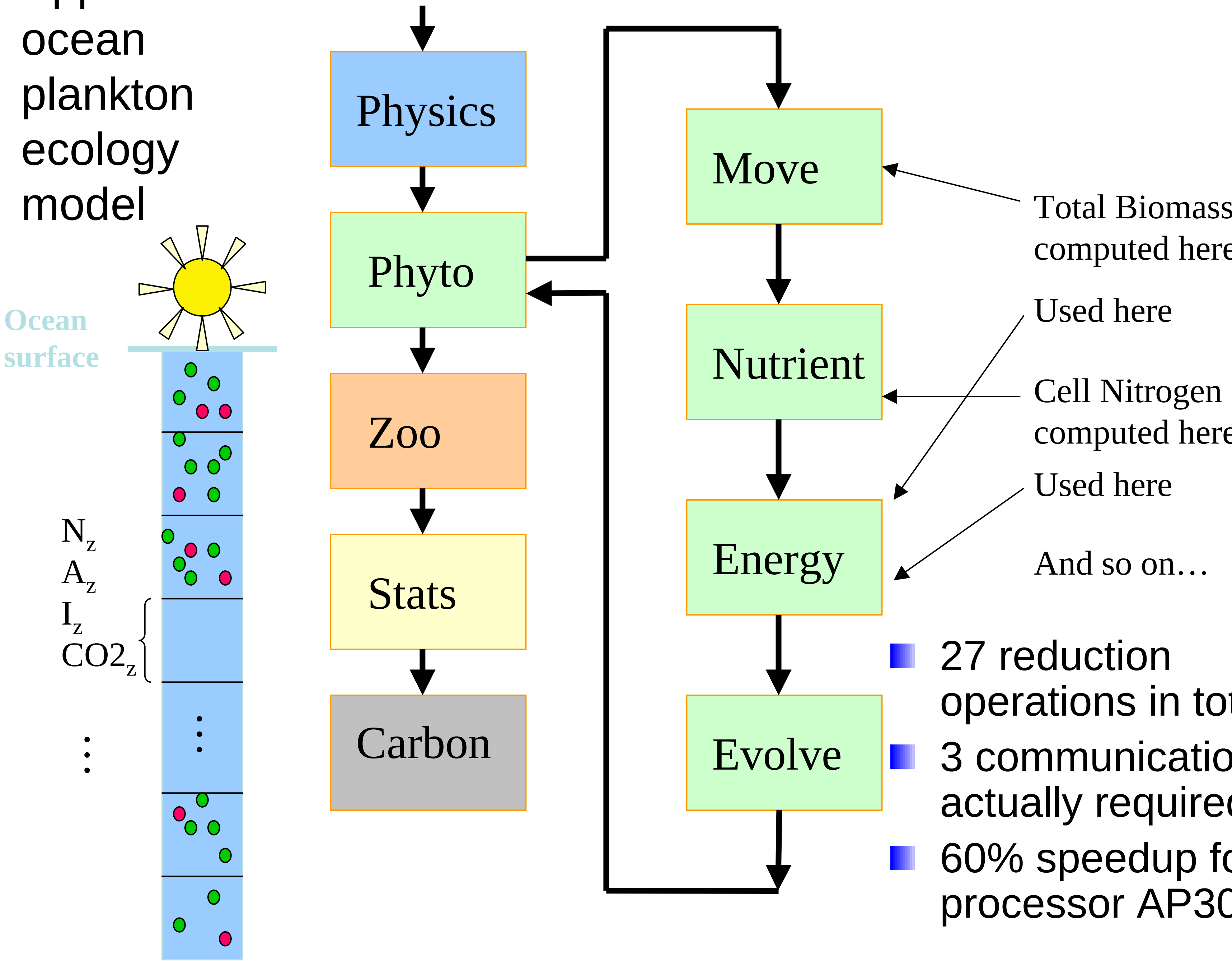
Component composition

```
double a[…][…],var[…] ;
for( i=0; i<N; i++ ) {
  sum(a[i]) ;
  sumSq(a[i]) ;
  var[i] = (s2-s1*s1/N)/(N-1);
}
```

Assignment to local
➡ force point

- For N=3000 our CFL library improved performance on linux cluster by 44.5%

- Application: ocean plankton ecology model



- 27 reduction operations in total
- 3 communications actually required!
- 60% speedup for 32-processor AP3000

A.J. Field, P.H.J. Kelly and T.L. Hansen, **"Optimizing Shared Reduction Variables in MPI Programs"**. In Euro-Par 2002

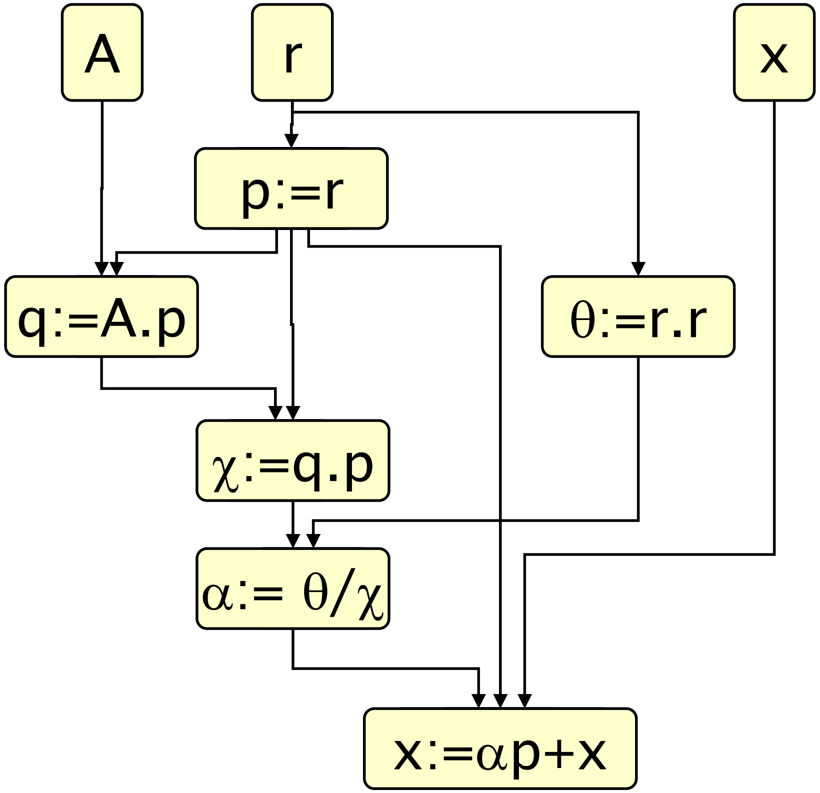
Adaptation #2: alignment in parallel BLAS

```
template<class Matrix, class Vector, class Precond, class Real>
int CG( const Matrix &A, Vector &x,
        const Vector &b, const Precond &M,
        int &max_iter, Real &tol )
{
  // local vector and scalar declarations & initial convergence test omitted

  for( int i = 1; i <= max_iter; i++ ) {
    z = M.solve( r );
    rho(0) = dot(r, z);

    if (i == 1)
      p = z;
    else {
      beta(0) = rho(0) / rho_1(0);
      p = z + beta(0) * p;
    }
    q = A*p;
    alpha(0) = rho(0) / dot(p, q);

    x += alpha(0) * p;
    r -= alpha(0) * q;

    if( (resid = norm(r) / normb) <= tol )
      tol = resid;
      max_iter = i;
      return 0;
    }
    rho_1(0) = rho(0);
  }
  tol = resid;
  return 1;
}
```

- This is a generic conjugate-gradient solver algorithm, part of Dongarra et al's IML++ library
- It is parameterised by the Matrix and Vector types
- Our DESOBLAS library implements this API for dense matrices
- In parallel using MPI

Adaptation #2: alignment in parallel BLAS

- Execution is delayed until output or conditional forces computation
- BLAS functions return opaque handles



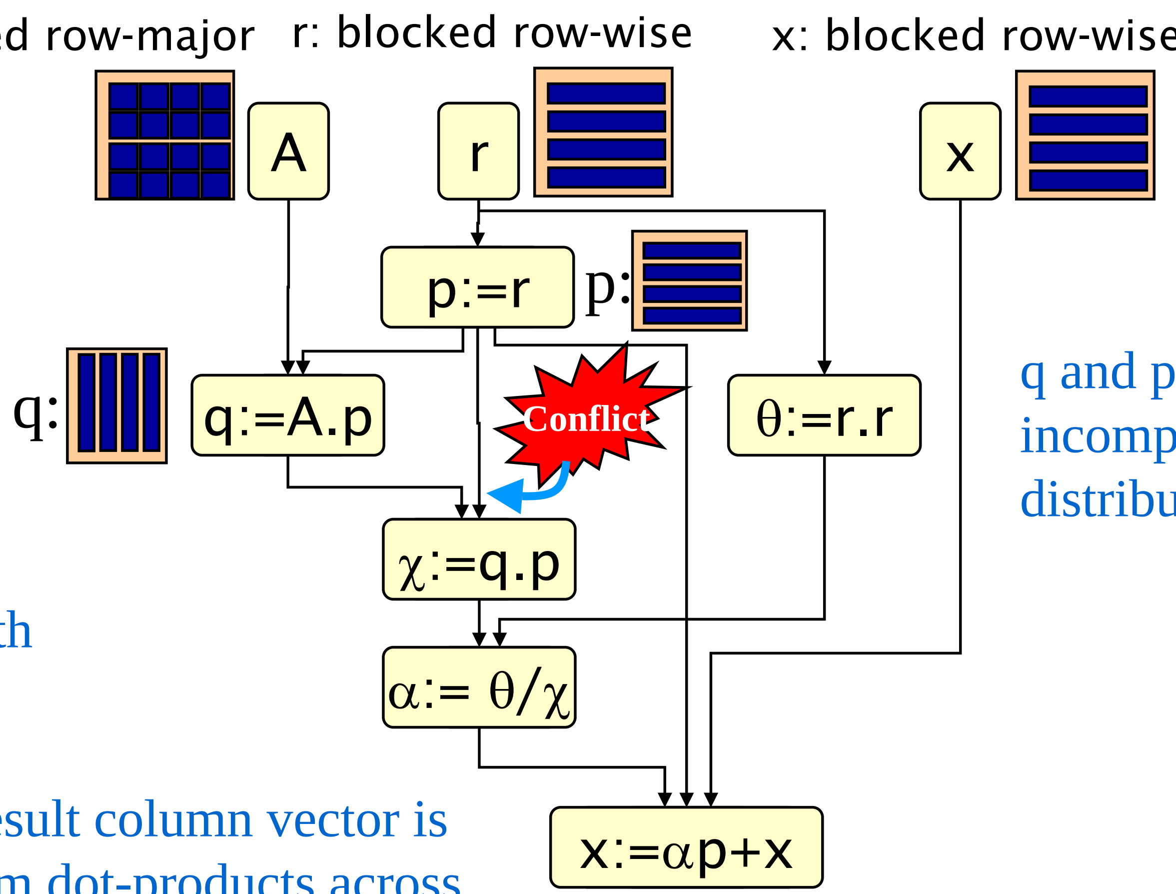
- Example: conjugate gradient

- Library builds up data flow graph "recipe" representing delayed computation
- This allows optimization to exploit foreknowledge of how results will be used

Adaptation #2: alignment in parallel BLAS

- For parallel dense BLAS, main issue is avoiding unnecessary data redistributions
- Consider just the first iteration:

Choose default distributions when variables initialised. Vectors are usually replicated



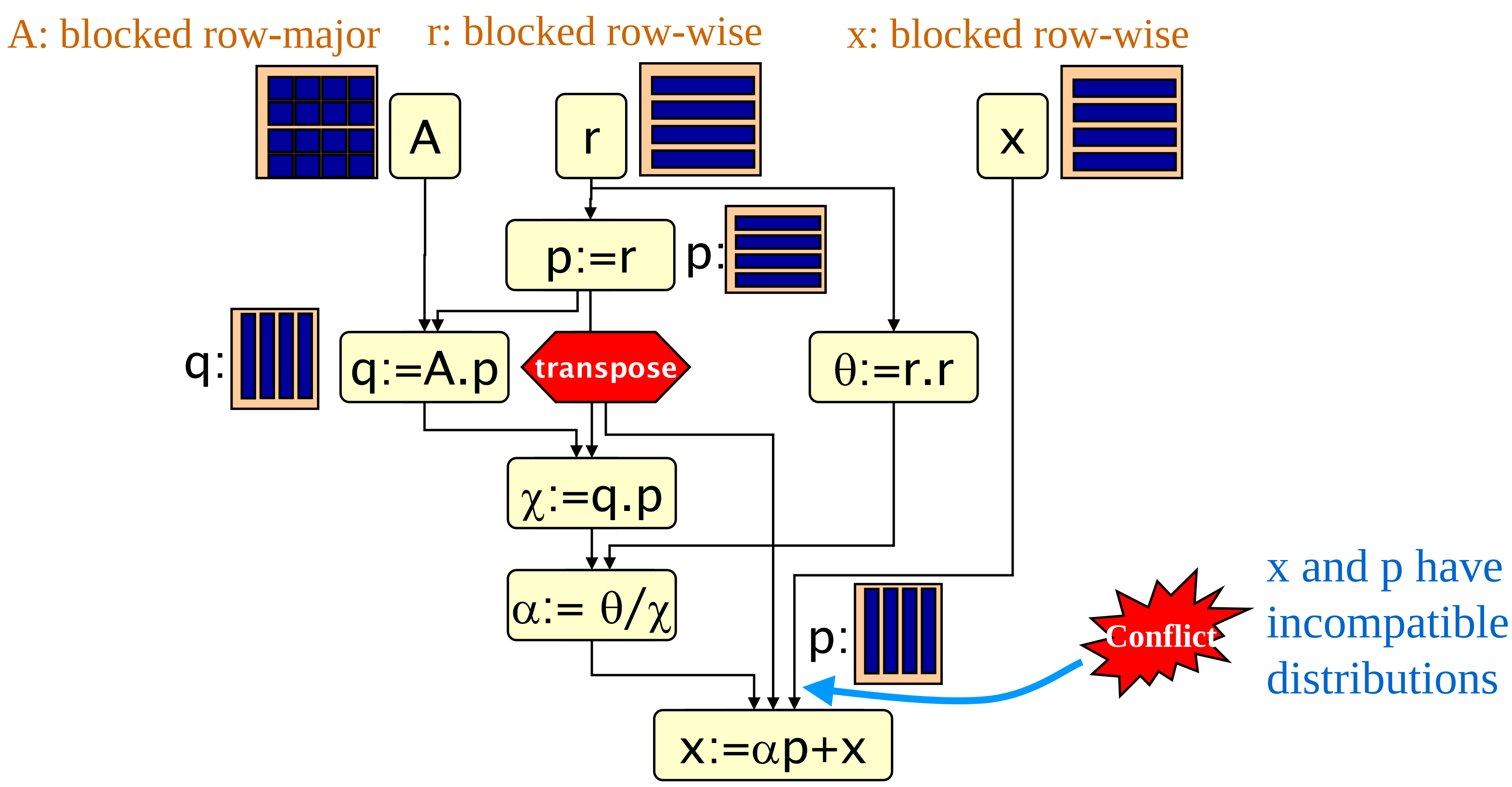
Result of matrix-vector multiply is aligned with the matrix columns

(because result column vector is formed from dot-products across each row)

q and p have incompatible distributions

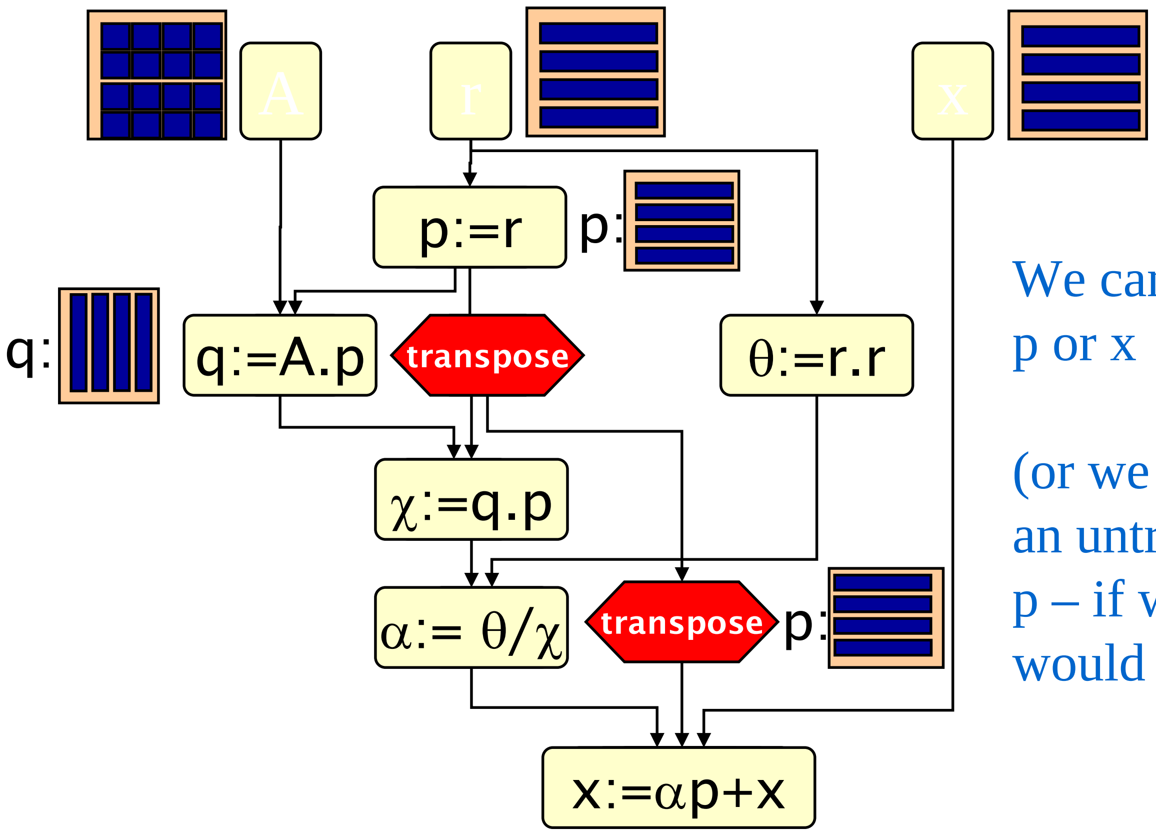
Adaptation #2: alignment in parallel BLAS

- We are forced to insert a transpose:



Adaptation #2: alignment in parallel BLAS

- We are forced to insert *another* transpose:

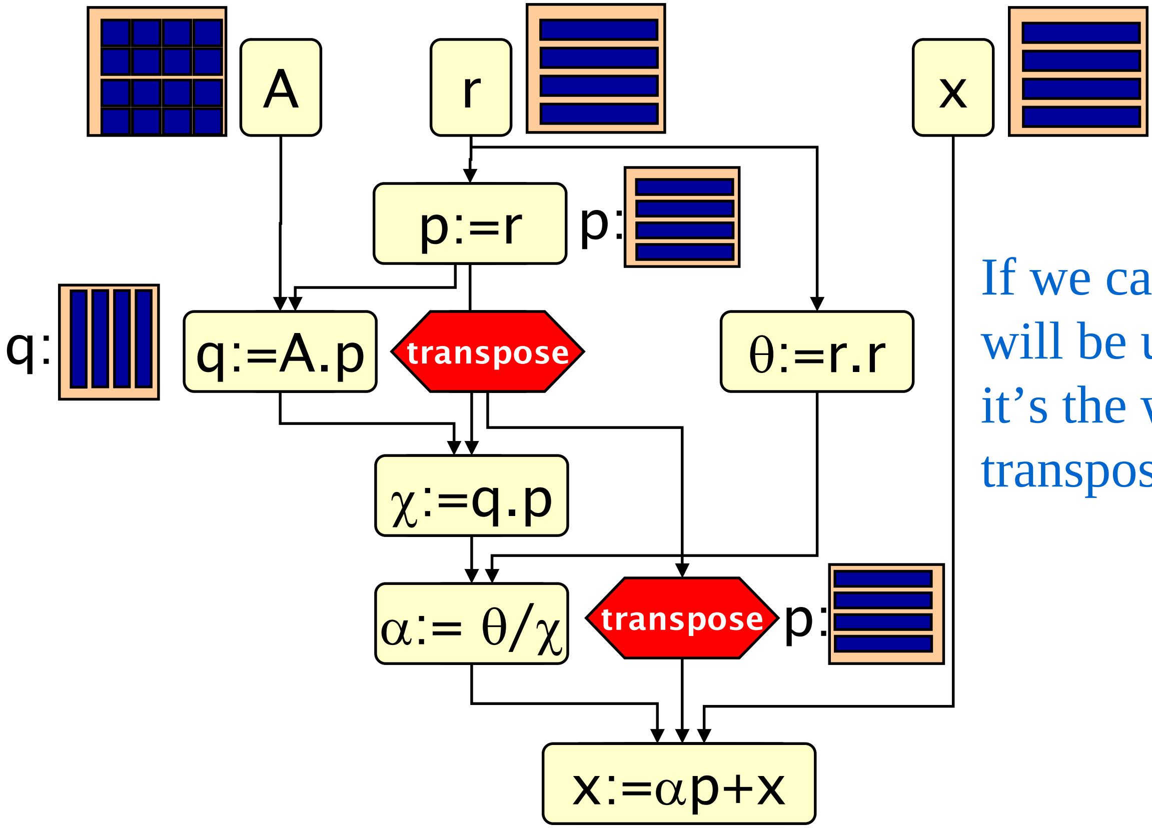


We can transpose either p or x

(or we could have kept an untransposed copy of p – if we'd known it would be needed)

Adaptation #2: alignment in parallel BLAS

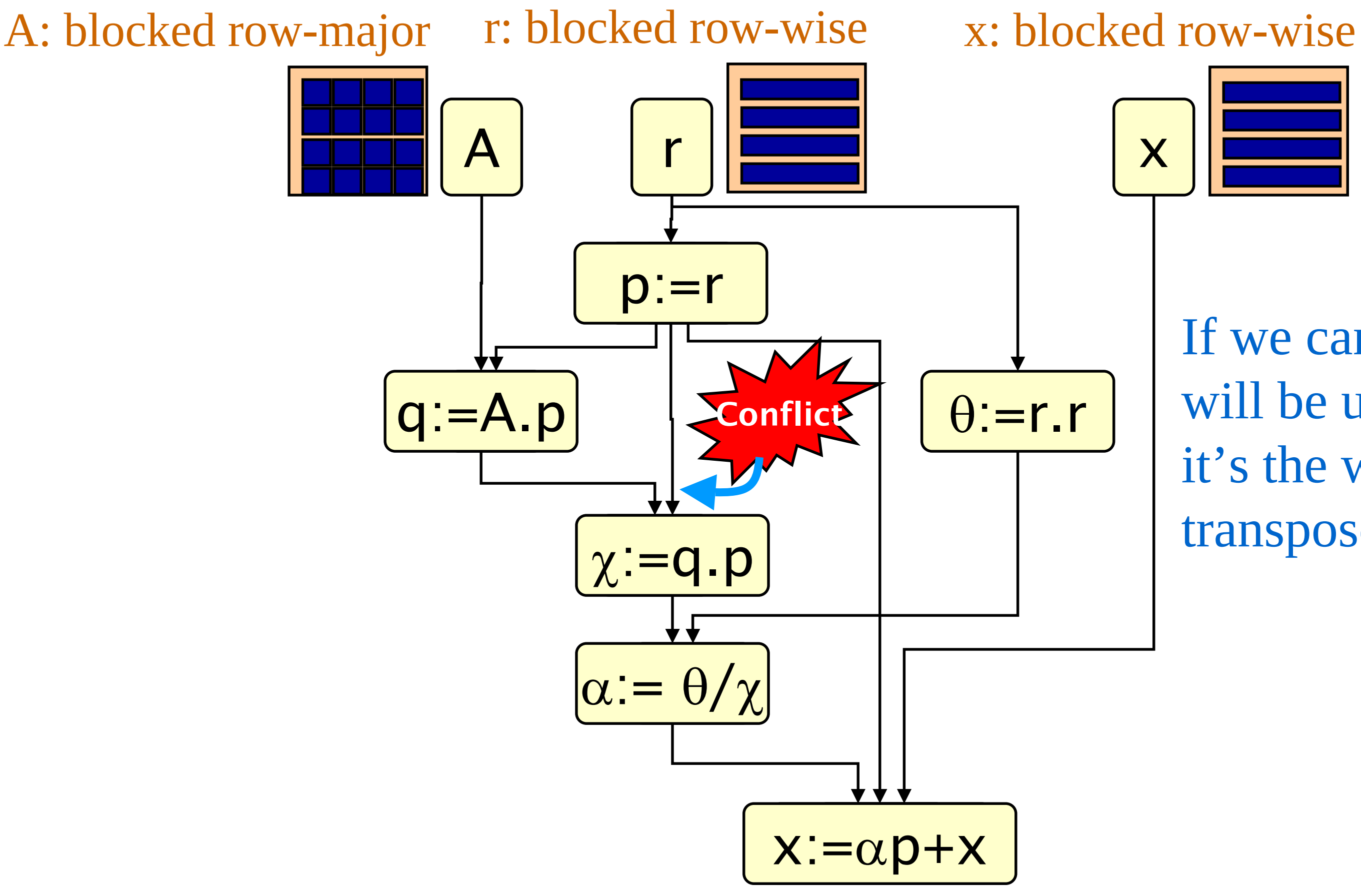
- Delayed execution allows us to see how values will be used and choose better:



If we can foresee how p will be used, we can see it's the wrong thing to transpose…

Adaptation #2: alignment in parallel BLAS

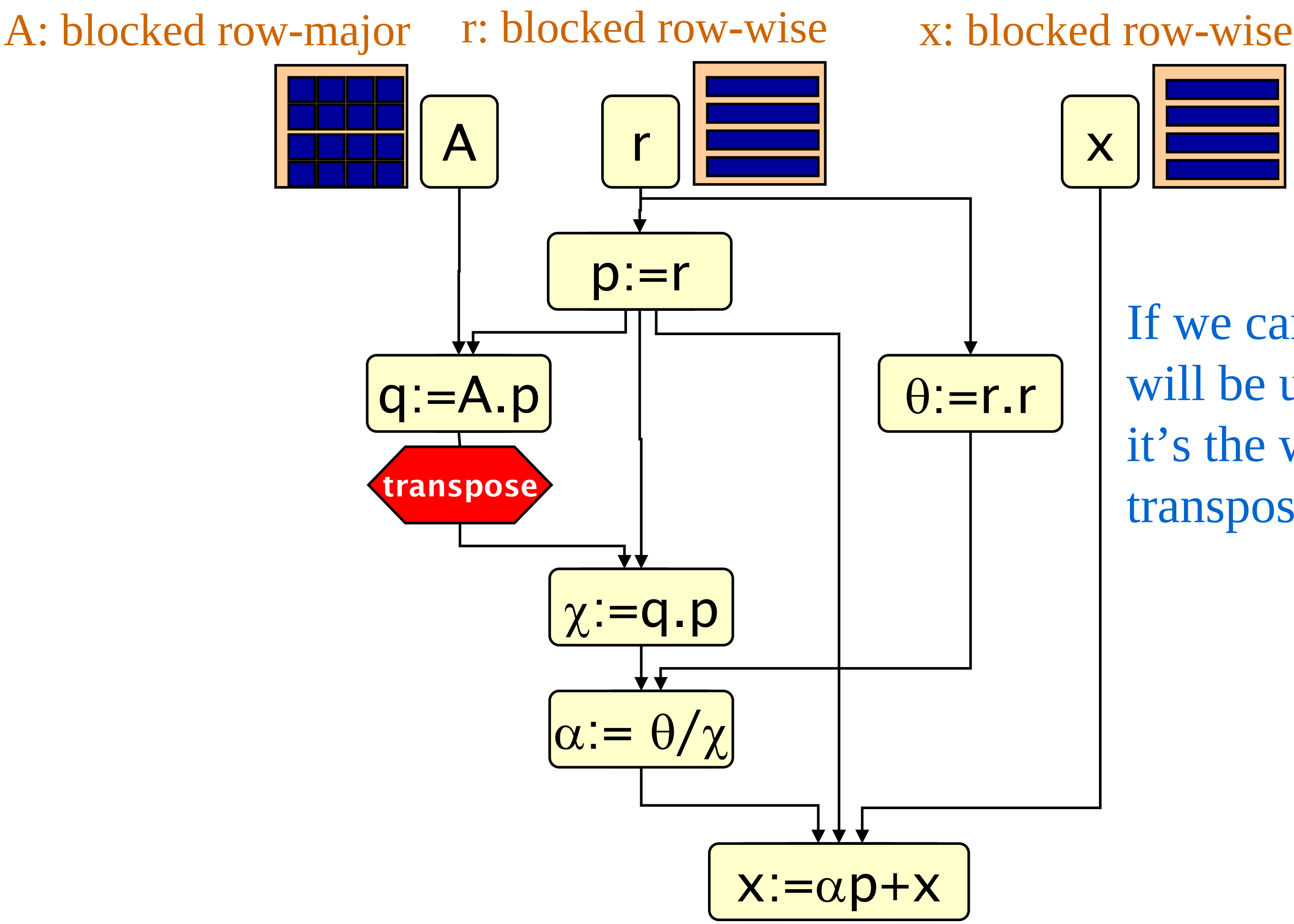
- Delayed execution allows us to see how values will be used and choose better:



If we can foresee how p will be used, we can see it's the wrong thing to transpose…

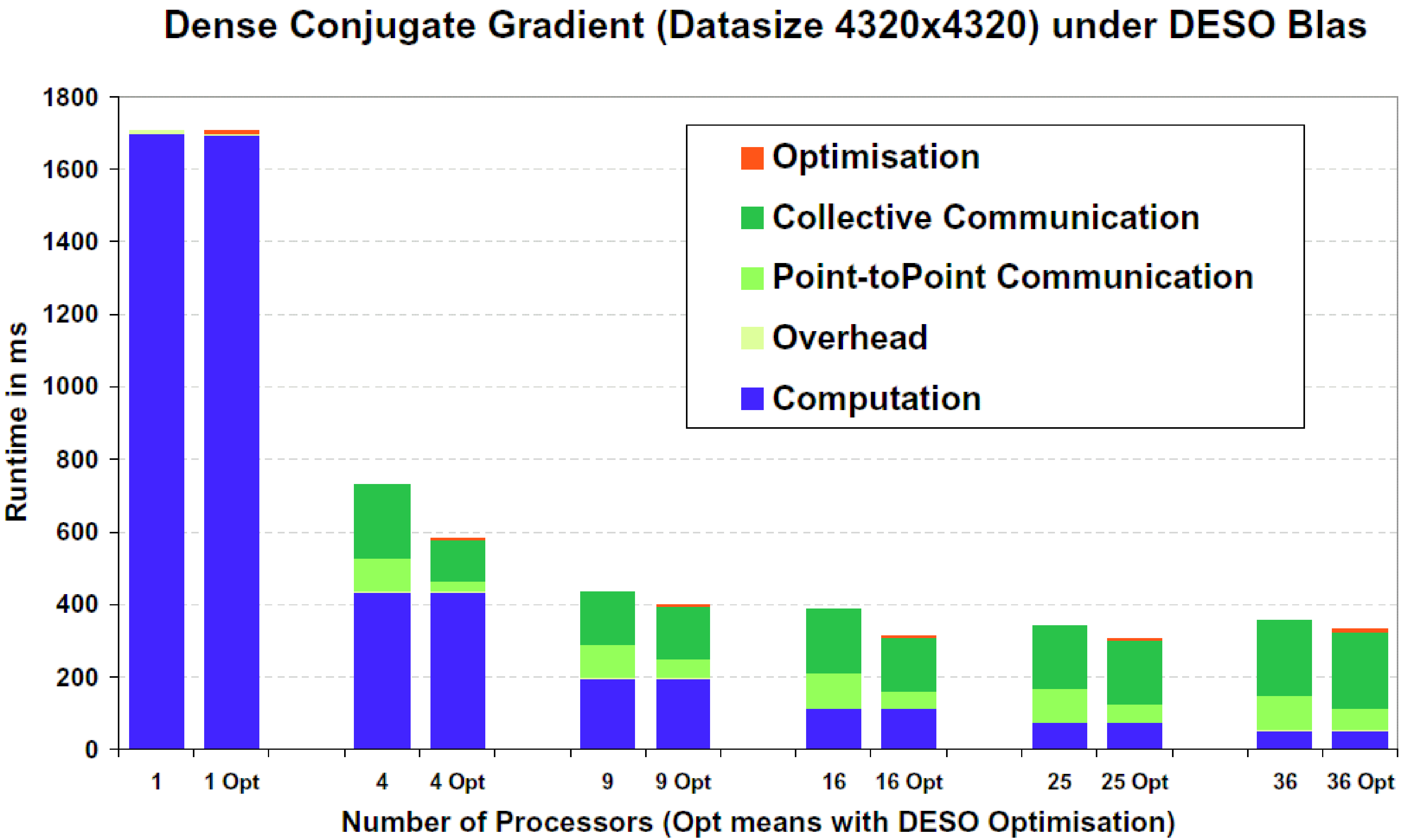
Adaptation #2: alignment in parallel BLAS

- Delayed execution allows us to see how values will be used and choose better:



If we can foresee how p will be used, we can see it's the wrong thing to transpose…

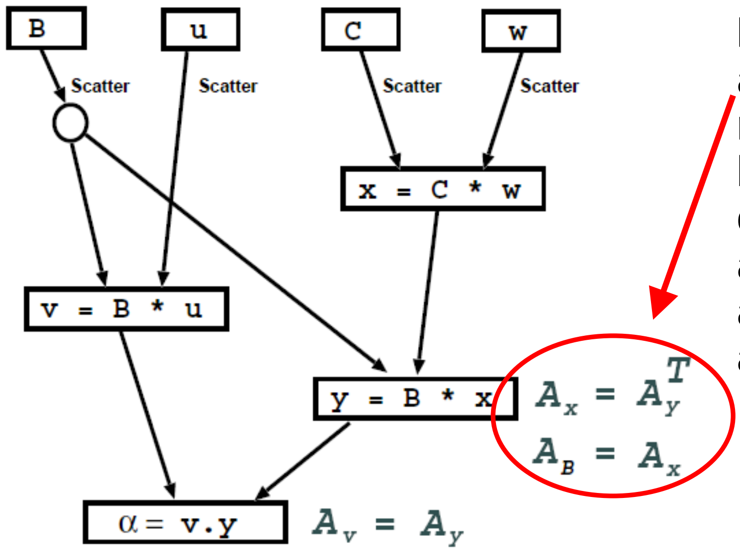
Avoiding redistributions: performance



- Cluster of 2GHz P4, 500 MB RAM, running Linux 2.4.20 and gcc 2.95.3, using C/Fortran bindings (not C++ overloading)

Metadata in DESOBLAS

- Each DESOBLAS library operator carries metadata, which is used at run-time to find an optimized execution plan
- For optimizing data placement, metadata is set of affine functions relating operator's output data placement to the placement of each input
- Network of invertible linear relationships allows optimizer to shift redistributions around dataflow graph to minimise communication cost
  - ((broadcasts and reductions involve singular placement relationships - see Beckmann and Kelly, LCPC'99 for how to make this idea still work))



Metadata: affine relationship between operand alignments and result alignment

Composition: metadata is assembled according to arcs of data flow graph to define system of alignment constraints:

$$A_u = A_v^T \quad A_x = A_y^T$$

$$A_A = A_u \quad A_w = A_x^T$$

$$A_C = A_w$$

$$A_B = A_x$$

Peter Liniker, Olav Beckmann and Paul H J Kelly, Delayed Evaluation, Self-Optimizing Software Components as a Programming Model. In Euro-Par 2002

Adaptation #3: specialisation

- The TaskGraph library is a portable C++ package for building and optimising code on-the-fly
- Compare:
  - `C (tcc) (Dawson Engler)
  - MetaOCaml (Walid Taha et al)
  - Jak (Batory, Lofaso, Smaragdakis)
- Multi-stage programming: “runtime code generation as a first-class language feature”

```
#include <TaskGraph>
#include <stdio.h>
#include <stdlib.h>
#include <sys/time.h>

using namespace tg;

int main( int argc, char argv[] ) {
  TaskGraph T;
  int b = 1, c = 1;

  taskgraph ( T ) {
    tParameter ( tVar ( int, a ) );

    a = a + c;
  }

  T.compile ( TaskGraph::GCC );
  T.execute ("a", &b, NULL);

  printf("b = %d\n", b);
}
```

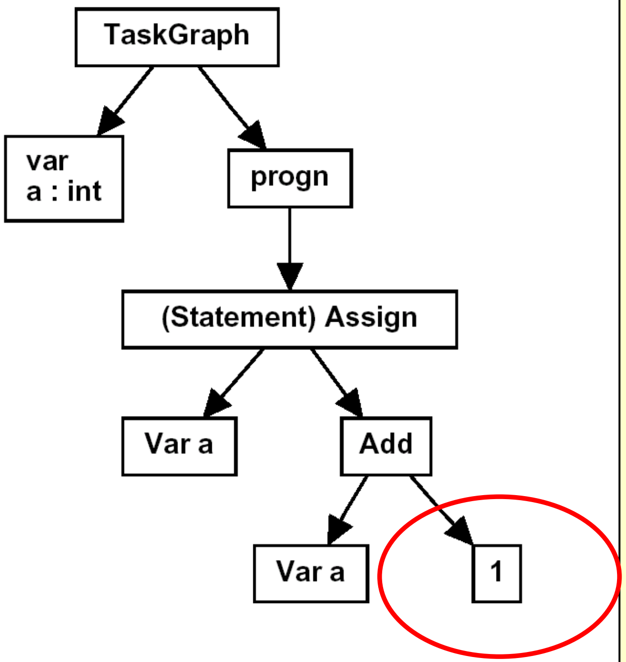
Adaptation #3: specialisation

- A taskgraph is an abstract syntax tree for a piece of executable code
- Syntactic sugar makes it easy to construct
- Defines a simplified sub-language
  - With first-class multidimensional arrays, no alliasing

```
#include <TaskGraph>
#include <stdio.h>
#include <stdlib.h>
#include <sys/time.h>

using namespace tg;

int main( int argc, char argv[] ) {
  TaskGraph T;
  int b = 1, c = 1;

  taskgraph ( T ) {
    tParameter ( tVar ( int, a ) );

    a = a + c;
  }

  T.compile ( TaskGraph::GCC );
  T.execute ( “a", &b, NULL);

  printf("b = %d\n", b);
}
```

Adaptation #3: specialisation

- Binding time is determined by types
- In this example
  - c is static
  - a is dynamic



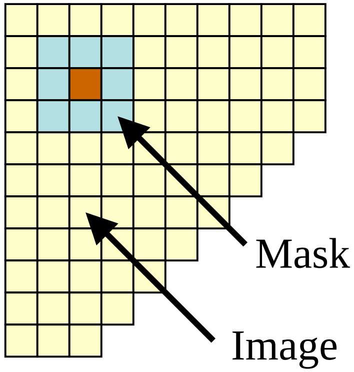
- built using value of c at construction time

```
#include <TaskGraph>
#include <stdio.h>
#include <stdlib.h>
#include <sys/time.h>

using namespace tg;

int main( int argc, char argv[] ) {
  TaskGraph T;
  int b = 1, c = 1;

  taskgraph ( T ) {
    tParameter ( tVar ( int, a ) );

    a = a + c;
  }

  T.compile ( TaskGraph::GCC );
  T.execute (“a", &b, NULL);

  printf("b = %d\n", b);
}
```

Adaptation #3: specialisation

Better example:

- Applying a convolution filter to a 2D image
- Each pixel is averaged with neighbouring pixels weighted by a stencil matrix




```
void filter (float *mask, unsigned n, unsigned m,
                 const float *input, float *output,
                 unsigned p, unsigned q)
{
  unsigned i, j;
  int       k, l;
  float     sum;
  int half_n = (n/2);
  int half_m = (m/2);

  for (i = half_n; i < p - half_n; i++) {
    for (j = half_m; j < q - half_m; j++) {
      sum = 0;
            // Loop bounds unknown at compile-time
            // Trip count 3, does not fill vector registers
      for (k = -half_n; k <= half_n; k++)
        for (l = -half_m; l <= half_m; l++)
          sum += input[(i + k) * q + (j + l)]
                      * mask[k * n + l];

      output[i * q + j] = sum;
    }
  }
}
```

- First without TaskGraph

Adaptation #3: specialisation

- TaskGraph representation of this loop nest
- Inner loops are static – executed at construction time
- Outer loops are dynamic
- Uses of mask array are entirely static

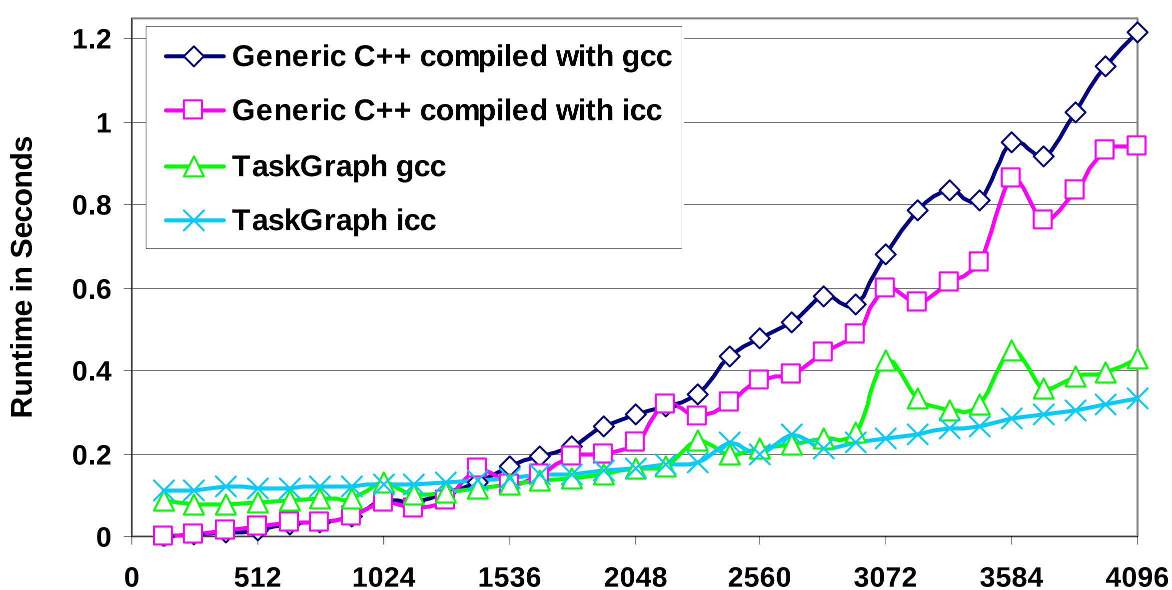
- This is deduced from the types of mask, k, m and l.

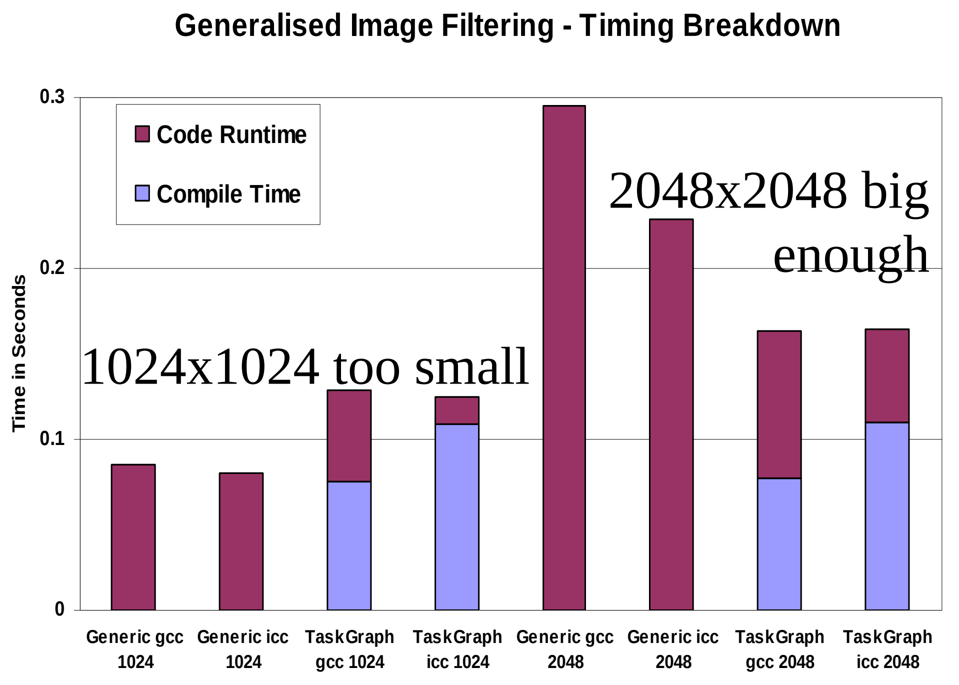
```
void taskFilter (TaskGraph &t,
                 float *mask, unsigned n, unsigned m,
                 unsigned p, unsigned q)
{
  taskgraph (t) {
    unsigned img_size[] = { IMG_SIZE, IMG_SIZE };
    tParameter(tArray(float, input, 2, img_size ));
    tParameter(tArray(float, output, 2, img_size ));
    unsigned k, l;
    unsigned half_n = (n/2);
    unsigned half_m = (m/2);

    tVar (float, sum);
    tVar (int, i);
    tVar (int, j);

    tFor (i, half_n, p - half_n - 1) {
      tFor (j, half_m, q - half_m - 1) {
        sum = 0;

        for ( k = 0; k < n; ++k )
          for ( l = 0; l < m; ++l )
            sum += input[(i + k - half_n)][(j + l - half_m)]
                   * mask[k * m + l];
        output[i][j] = sum;
      }                // Inner loops fully unrolled
    }                  // j loop is now vectorisable
  }
}
```

- Now with TaskGraph

Adaptation #3: specialisation

Image convolution using TaskGraphs: performance

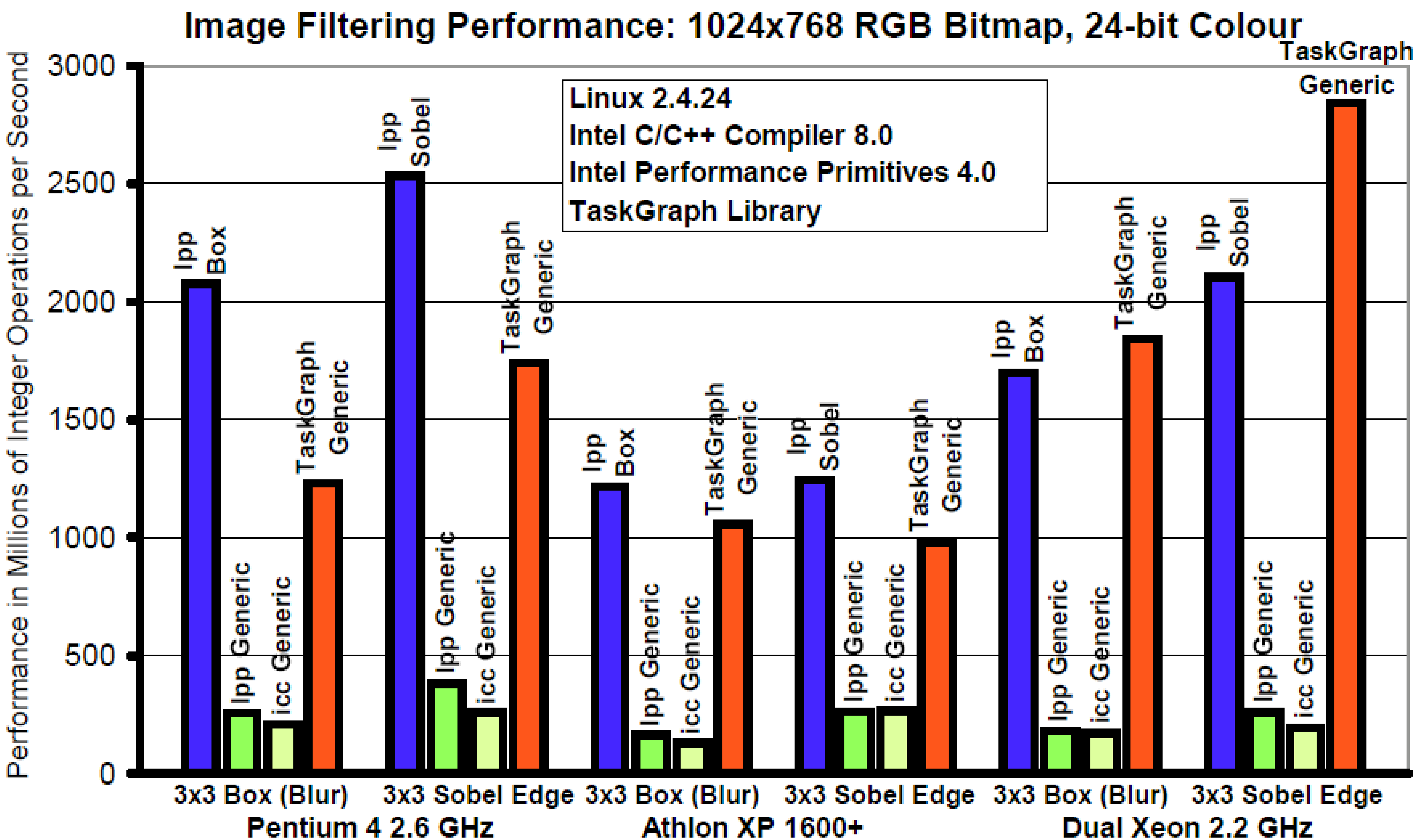




- We use a 3x3 averaging filter as convolution matrix
- Images are square arrays of single-precision floats ranging in size up to 4096x4096
- Measurements taken on a 1.8GHz Pentium 4-M running Linux 2.4.17, using gcc 2.95.3 and icc 7.0
- Measurements were taken for one pass over the image

(Used an earlier release of the TaskGraph library)

Adaptation #3: specialisation



- Application: Sobel filters in image processing (8-bit RGB data) – compared with Intel's Performance Programming Library

Adaptation #4: Adapting to platform/resources

- The TaskGraph library is a tool for dynamic code generation and optimisation
- Large performance benefits can be gained from specialisation alone

But there's more:

- TaskGraph library builds SUIF intermediate representation
- Provides access to SUIF analysis and transformation passes
  - SUIF (Stanford University Intermediate Form)
  - Detect and characterise dependences between statements in loop nests
  - Restructure – tiling, loop fusion, skewing, parallelisation etc

Tiling

- Example: matrix multiply

```
void taskMatrixMult (TaskGraph &t ,
                     TaskLoopIdentifier *loop) {
  taskgraph ( t ) {
    tParameter ( tArray ( float, a, 2, sizes ) );
    tParameter ( tArray ( float, b, 2, sizes ) );
    tParameter ( tArray ( float, c, 2, sizes ) );
    tVar ( int, i );
    tVar ( int, j );
    tVar ( int, k );

    tGetId ( loop[0] ); // label
    tFor ( i, 0, MATRIXSIZE - 1 ) {
      tGetId ( loop[1] ); // label
      tFor ( j, 0, MATRIXSIZE - 1 ) {
        tGetId ( loop[2] ); // label
        tFor ( k, 0, MATRIXSIZE - 1 ) {
          c[i][j] += a[i][k] * b[k][j];
        }
      }
    }
  }
}
```

Original TaskGraph for matrix multiply

```
int main ( int argc, char **argv ) {
  TaskGraph mm;
  TaskLoopIdentifier loop[3];

  // Build TaskGraph for ijk multiply
  taskMatrixMult ( loop, mm );

  // Interchange the j and k loops
  interchangeLoops ( loop[1], loop[2] );

  int trip[] = { 64, 64 };

  // Tile the j and k loops into 64x64 tiles
  tileLoop ( 2, &loop[1], trip );

  mm.compile ( TaskGraph::GCC );
  mm.execute ( "a", a, "b", b, "c", c, NULL );
}
```

Code to interchange and tile

Loop interchange and tiling

```
void taskMatrixMult (TaskGraph &t ,
                     TaskLoopIdentifier *loop) {
  taskgraph ( t ) {
    tParameter ( tArray ( float, a, 2, sizes ) );
    tParameter ( tArray ( float, b, 2, sizes ) );
    tParameter ( tArray ( float, c, 2, sizes ) );
    tVar ( int, i );
    tVar ( int, j );
    tVar ( int, k );

    tGetId ( loop[0] ); // label
    tFor ( i, 0, MATRIXSIZE - 1 ) {
      tGetId ( loop[1] ); // label
      tFor ( j, 0, MATRIXSIZE - 1 ) {
        tGetId ( loop[2] ); // label
        tFor ( k, 0, MATRIXSIZE - 1 ) {
          c[i][j] += a[i][k] * b[k][j];
        }
      }
    }
  }
}
```

Original TaskGraph for matrix multiply

```
int main ( int argc, char **argv ) {
  TaskGraph mm;
  TaskLoopIdentifier loop[3];

  // Build TaskGraph for ijk multiply
  taskMatrixMult ( loop, mm );

  // Interchange the j and k loops
  interchangeLoops ( loop[1], loop[2] );

  int trip[] = { 64, 64 };

  // Tile the j and k loops into 64x64 tiles
  tileLoop ( 2, &loop[1], trip );

  mm.compile ( TaskGraph::GCC );
  mm.execute ( "a", a, "b", b, "c", c, NULL );
}
```

Code to interchange and tile

```
extern void taskGraph_1(void **params)
{
  float (*a)[512];
  float (*b)[512];
  float (*c)[512];
  int i;
  int j;
  int k;
  int j_tile;
  int k_tile;

  a = *params;
  b = params[1];
  c = params[2];
  for (i = 0; i <= 511; i++)
    for (j_tile = 0; j_tile <= 511; j_tile += 64)
      for (k_tile = 0; k_tile <= 511; k_tile += 64)
        for (j = j_tile;
               j <= min(511, 63 + j_tile); j++)
          for (k = max(0, k_tile);
                 k <= min(511, 63 + k_tile); k++)
            c[i][k] = c[i][k] + a[i][j] * b[j][k];
}
```

- Generated code

  (Slightly tidied)

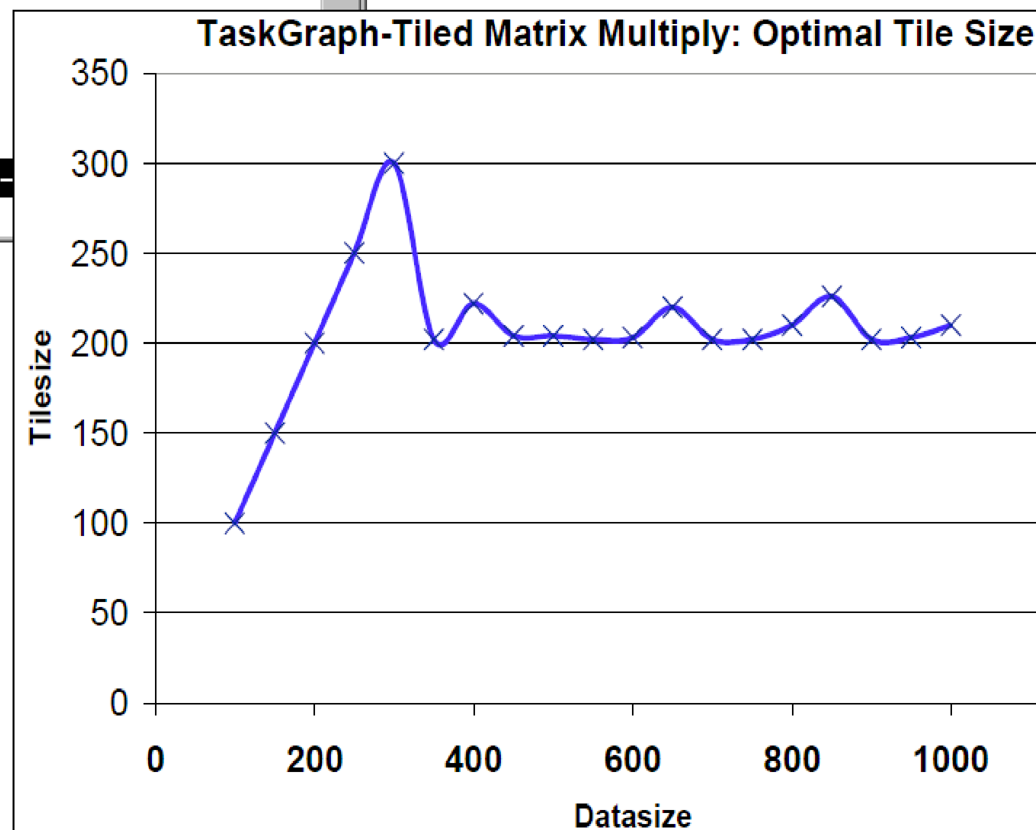
Software Perf

Imperial College London

Adaptation #4: Adapting to platform/resources

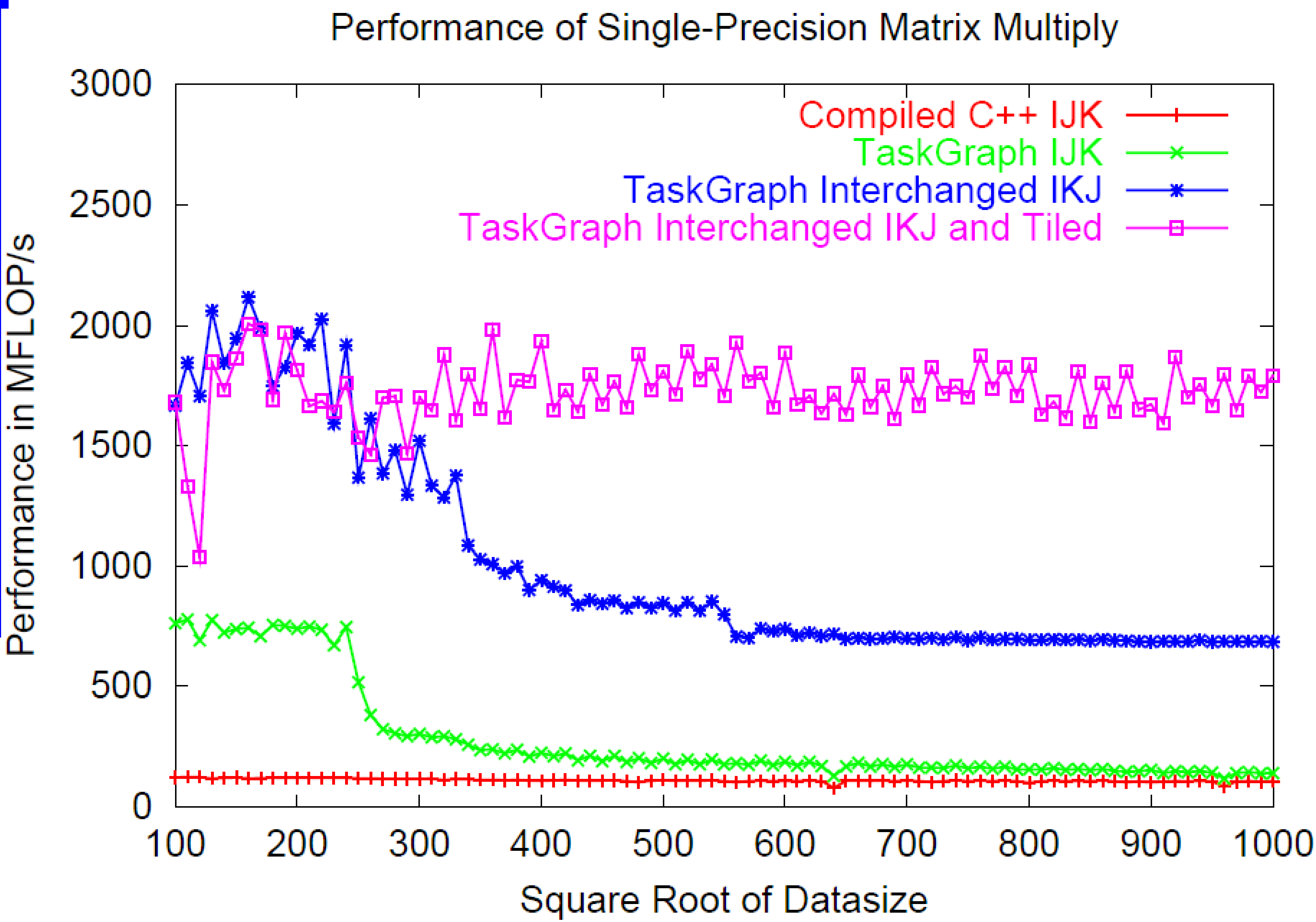
```
int bestTime;
int bestSize = 0;
for (int tsz = 4; tsz <= MATRIXSIZE; ++tsz) {
  int trip3 = { tsz, tsz, tsz };
  TaskLoopIdentifier loop[3];
  TaskGraph MM;
  taskMatrixMult(loop, MM);
  interchangeLoops(loop[1], loop[2]);
  tileLoop(3, &loop[0], trip3);
  MM.compile(TaskGraph::ICC, false);
  tt3 = time_function();
  MM.execute("A",A, "B",B, "C",C, NULL);
  time = time_function()-tt3;
  if (time < bestTime || bestSize == 0) {
    bestTime = time; bestSize = tsz;
  }
}
--\--  IterativeMM.cc        (C++)--L2-- 2%---
```

- We can program a search for the best implementation for our particular problem size, on our particular hardware

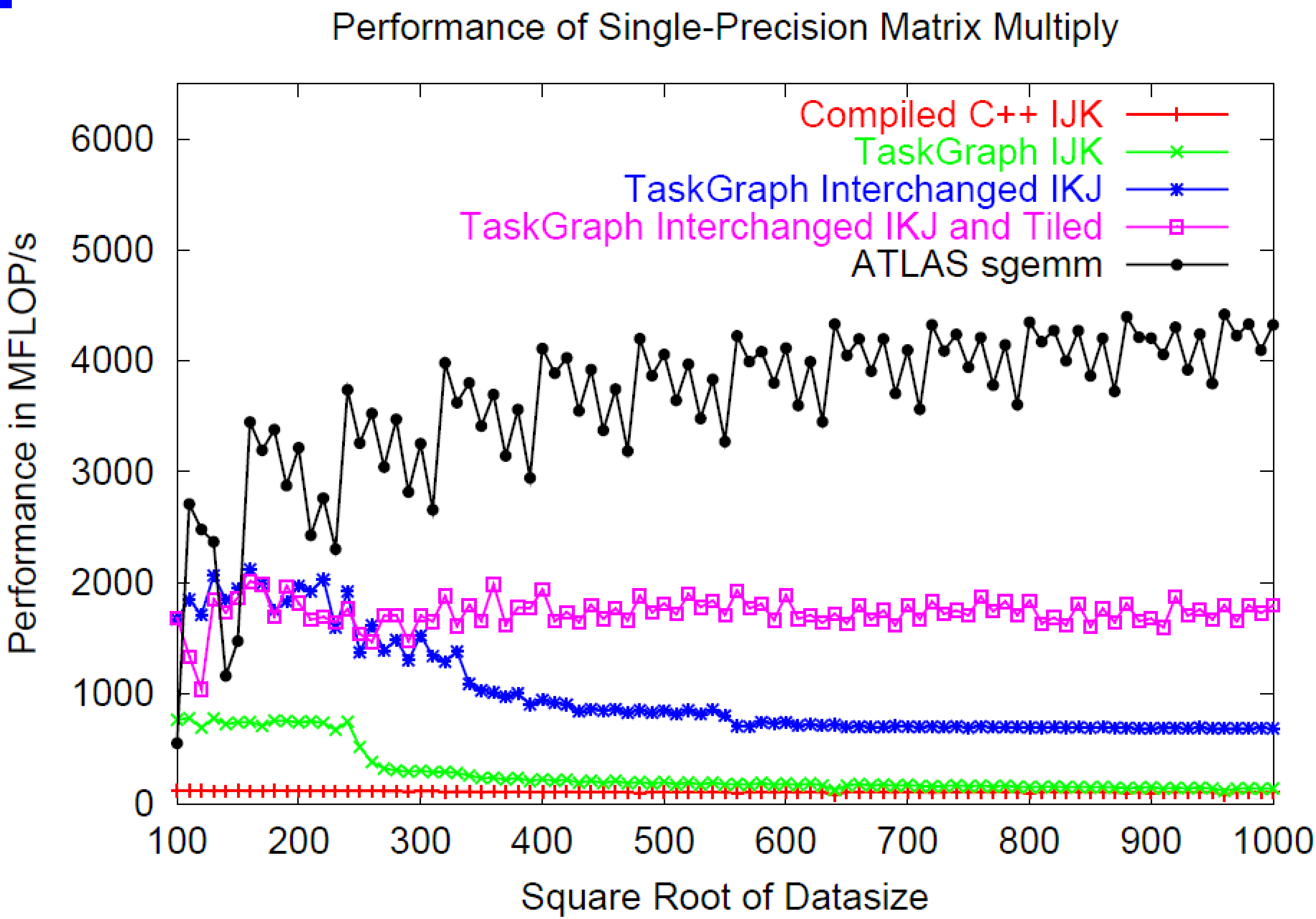


On Pentium 4-M, 1.8 GHz, 512KB L2 cache, 256 MB, running Linux 2.4 and icc 7.1.

Adaptation #4:
Adapting to platform/resources



Adaptation #4:
Adapting to platform/resources



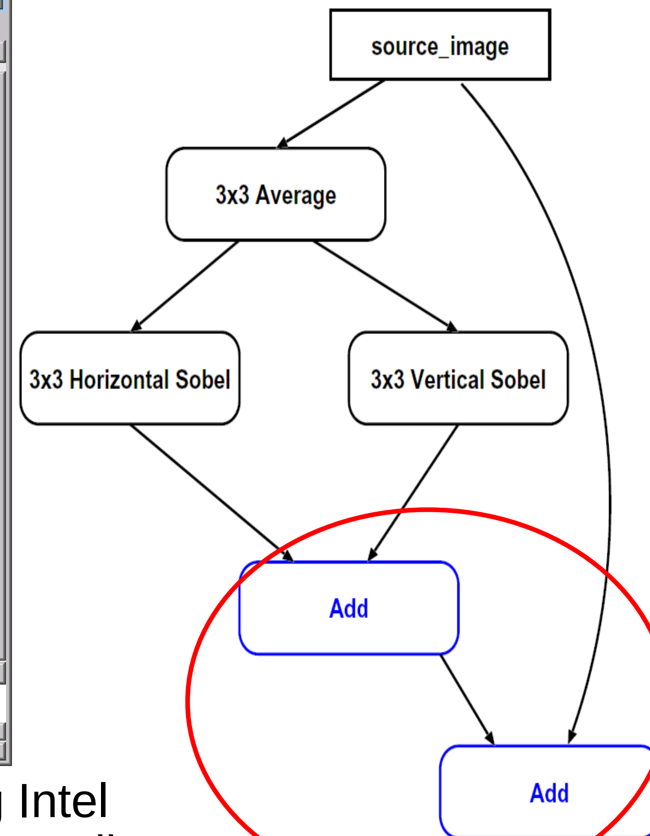
Potential for user-directed restructuring

- Programmer controls application of sophisticated transformations
- Performance benefits can be large – in this example >8x
- Different target architectures and problem sizes need different combinations of optimisations
  - ijk or ikj?
  - Hierarchical tiling
  - 2d or 3d?
  - Copy reused submatrix into contiguous memory?
- Matrix multiply is a *simple* example

Olav Beckmann, Alastair Houghton, Paul H J Kelly and Michael Mellor, Run-time code generation in C++ as a foundation for domain-specific optimisation. Domain-Specific Program Generation, Springer (2004).

Cross-component loop fusion

```
emacs@SECONDSELF
Buffers Files Tools Edit Search Mule C++ Help
  TaskGraph T;
  taskgraph( T ) {
    unsigned int ds[] = fsz, szg;
    tParameter(tArrayFromList(float, dstimg, 2, ds));
    tParameter(tArrayFromList(float, srcimg, 2, ds));
    tArrayFromList( float, blur , 2, ds );
    // ...
    instantiateBlur (blur , srcimg, i , j , sz, sz , 3);
    instantiateSobelHoriz(horiz , blur , i , j , sz, sz);
    instantiateSobelVert (vert , blur , i , j , sz, sz);
    instantiateAdd(both, vert , horiz , i , j , sz, sz);
    instantiateAdd(dstimg, blur , both, i , j , sz, sz);
  }
  T.applyOptimisation ("fusion");
  T.compile(TaskGraph::ICC, true);
  T.execute("dstimg", result , "srcimg" , image, NULL);
}
--\--  Filter.cc          (C++)--L10--Bot-----------------
```



- Image processing example
- Blur, edge-detection filters then sum with original image
- Final two additions using Intel Performance Programming Library:

```
emacs@SECONDSELF
Buffers Files Tools Edit Search Mule C++ Help
  // Ipp Domain Specific Library
  ippiAdd_32f_C1R( horiz, length , vert , length,
                   both, length , whole );
  ippiAdd_32f_C1R( image, length, both, length,
                   result , length , whole );
--\--  FilterIPP.cc       (C++)--L5--All----------
```

Software Group

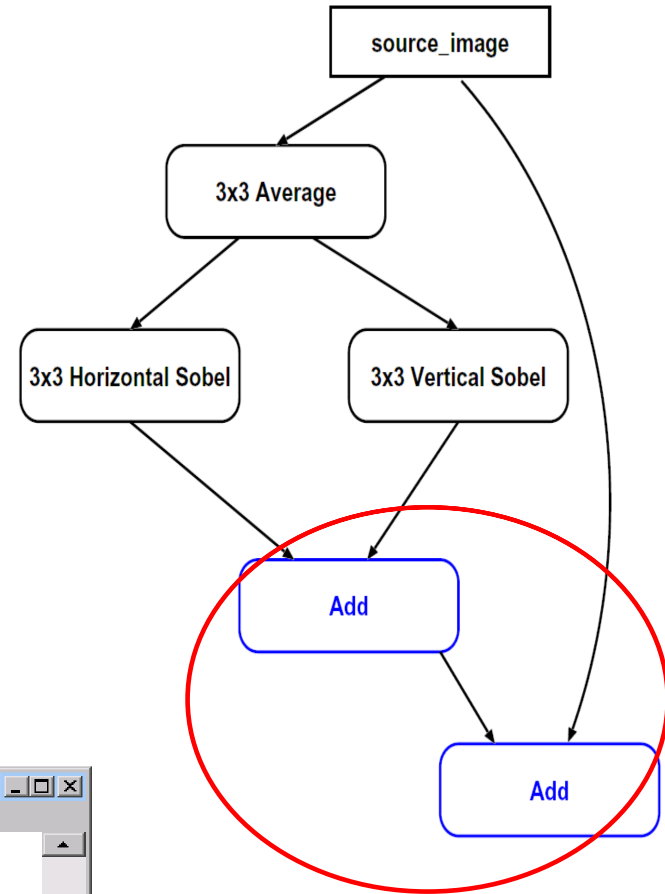
Imperial College London

Cross-component loop fusion

emacs@SECONDSELF

Buffers Files Tools Edit Search Mule C++ Help

```
// TaskGraph Generated Code
  for ( i = 0; i <= 1199; i++) {
    for ( j = 0; j <= 1599; j++) {
      both[ i ][ j ] = vert [ i ][ j ] + horiz [ i ][ j ];
    }
  }
  for ( i = 0; i <= 1199; i++) {
    for ( j = 0; j <= 1599; j++) {
      tgimage[i ][ j ] = blur [ i ][ j ] + both[ i ][ j ];
    }
  }
}
--\--  FilterGenerated.cc      (C++)--L2--Bot-------------
```



- After loop fusion:

emacs@SECONDSELF

Buffers Files Tools Edit Search Mule C++ Help

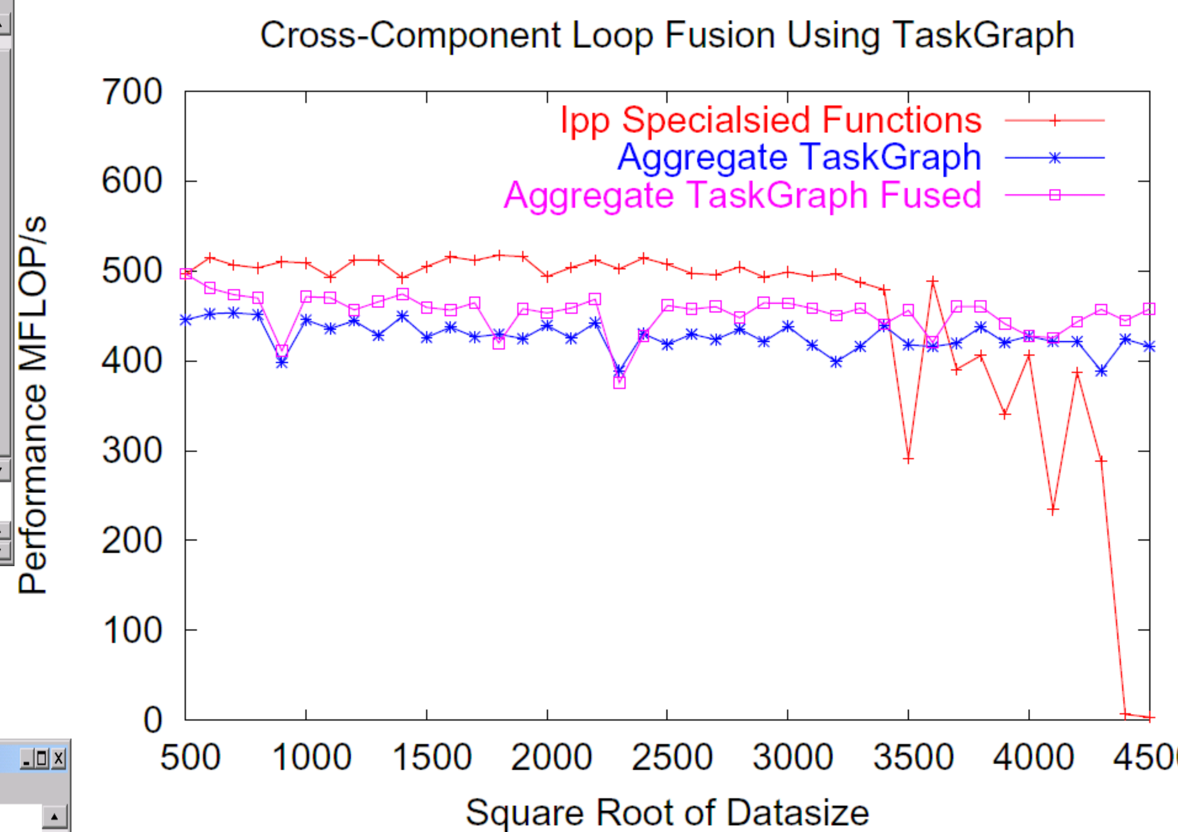
```
// TaskGraph Optimised Generated Code
  for ( i = 0; i <= 1199; i++) {
    for ( j = 0; j <= 1599; j++) {
      both[ i ][ j ] = vert [ i ][ j ] + horiz [ i ][ j ];
      tgimage[i ][ j ] = blur [ i ][ j ] + both[ i ][ j ];
    }
  }
}
--\--  FilterGenerated.cc      (C++)--L16--Bot------------
```

Cross-component loop fusion


```
// TaskGraph Generated Code
for ( i = 0; i <= 1199; i++) {
  for ( j = 0; j <= 1599; j++) {
    both[ i ][ j ] = vert [ i ][ j ] + horiz [ i ][ j ];
  }
}
for ( i = 0; i <= 1199; i++) {
  for ( j = 0; j <= 1599; j++) {
    tgimage[i ][ j ] = blur [ i ][ j ] + both[ i ][ j ];
  }
}
}
--\--  FilterGenerated.cc      (C++)--L2--Bot------------
```

- After loop fusion:

```
// TaskGraph Optimised Generated Code
for ( i = 0; i <= 1199; i++) {
  for ( j = 0; j <= 1599; j++) {
    both[ i ][ j ] = vert [ i ][ j ] + horiz [ i ][ j ];
    tgimage[i ][ j ] = blur [ i ][ j ] + both[ i ][ j ];
  }
}
}
--\--  FilterGenerated.cc      (C++)--L16--Bot------------
```

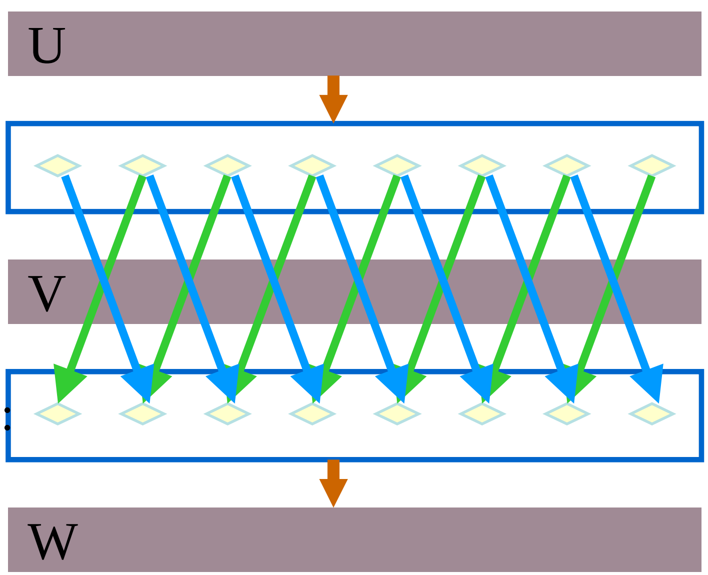


- Simple fusion leads to small improvement
- Beats Intel library only on large images
- Further fusion opportunities require skewing/retiming

Performance-programming Component model

- Dependence metadata
  - Components should carry a description of their dependence structure
  - That is based on an abstraction of the component's Iteration Space Graph (ISG)
- Eg to allow simple check for validity of loop and communication fusion
- Eg to determine dependence constraints on distribution
- Eg so we can align data distributions to minimise communication
- To predict communication volumes

Jacobi1D(U,V):



Jacobi1D(V,W):

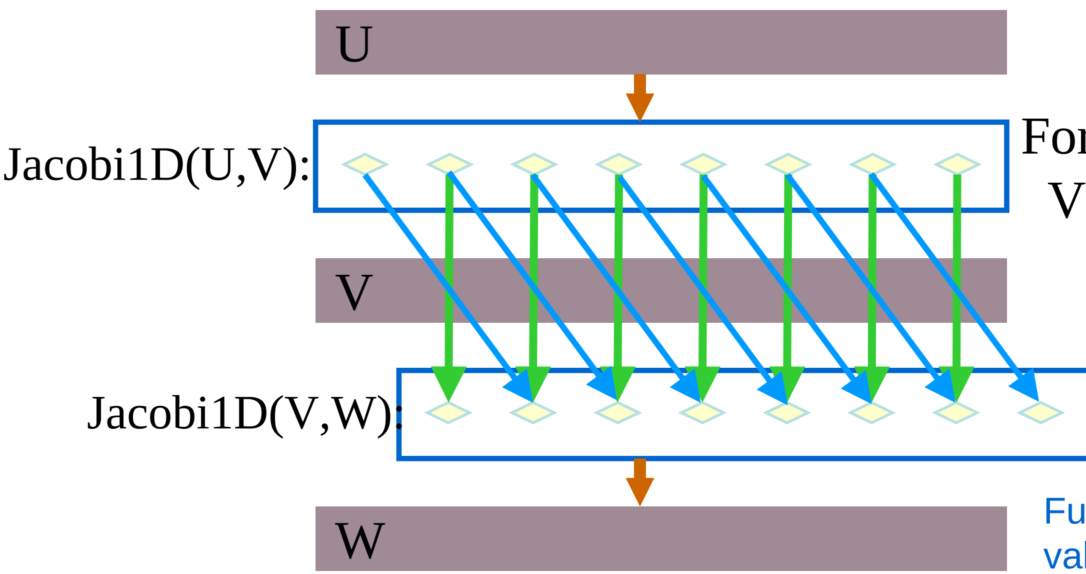
For (i=1; i<N; i++)
  V[i] = (U[i-1] + U[i+1])/2

For (i=1; i<N; i++)
  W[i] = (V[i-1] + V[i+1])/2

Fusion ***invalid***: iteration i of second loop reads value generated at iteration i of first loop

Performance-programming Component model

- Dependence metadata
  - Components should carry a description of their dependence structure
  - That is based on an abstraction of the component's Iteration Space Graph (ISG)
- Eg to allow simple check for validity of loop and communication fusion
- Eg to determine dependence constraints on distribution
- Eg so we can align data distributions to minimise communication
- To predict communication volumes



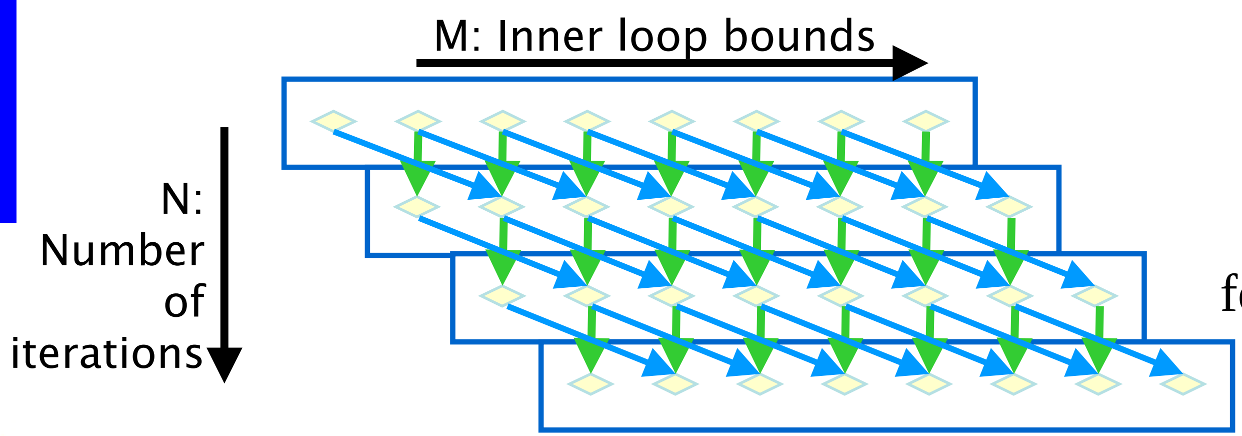

```
For (i=1; i<N; i++)
  V[i] = (U[i-1] + U[i+1])/2
```

```
For (i=1; i<N; i++)
  W[i] = (V[i-1] + V[i+1])/2
```

Fusion ***valid***: iteration i of second loop reads value generated at iteration i of first loop

Performance-programming Component model

- Performance metadata
  - Components should carry a model of how execution time depends on parameters and configuration
  - That is based on an abstraction of the component's Iteration Space Graph (ISG)
  - Eg to allow scheduling and load balancing
  - Eg to determine communication-computation-recomputation tradeoffs

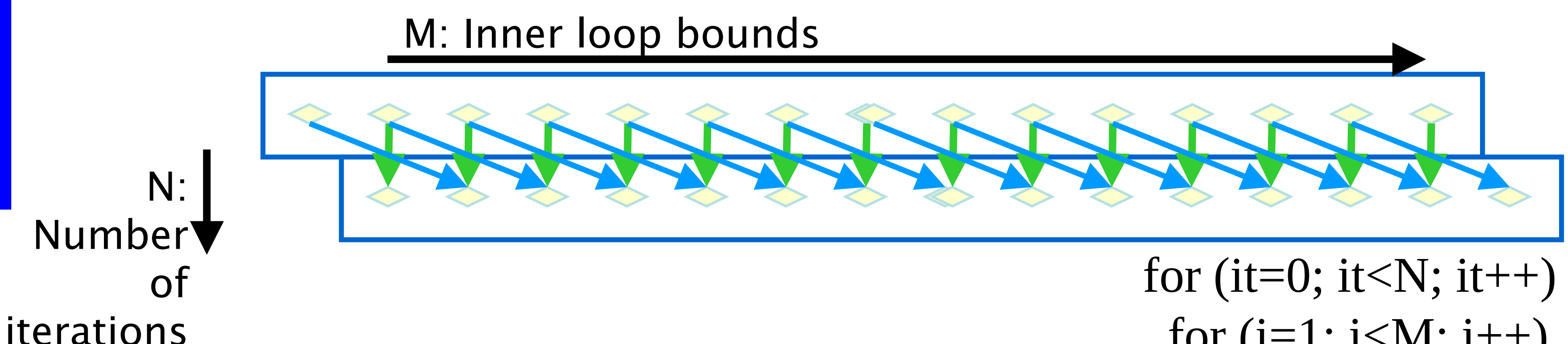



```
for (it=0; it<N; it++)
 for (i=1; i<M; i++)
   V[i] = (U[i-1] + U[i+1])/2
```

- Compute volume: N.(M-1)
- Input volume: M
- Output volume: M-1

Performance-programming Component model

- Performance metadata
  - Components should carry a model of how execution time depends on parameters and configuration
  - That is based on an abstraction of the component's Iteration Space Graph (ISG)
  - Eg to allow scheduling and load balancing
  - Eg to determine communication-computation-recomputation tradeoffs




```
for (it=0; it<N; it++)
  for (i=1; i<M; i++)
    V[i] = (U[i-1] + U[i+1])/2
```

- Compute volume: N.(M-1)
- Input volume: M
- Output volume: M-1

Component metadata research agenda

- We want to adapt to shape of data
- But in interesting applications, data shape is not regular
  - Shape description/metadata depends on data values
  - Metadata size is significant
  - Metadata generation/manipulation is significant part of computational effort
- The problem:
  - Cost of organising and analysing the data may be large compared to the computation itself
  - Size of metadata may be large compared with size of the data itself
- What does this mean?
  - Some kind of reflective programming
  - Arguably, metaprogramming
- Programs that make runtime decisions about how much work to do to optimise future execution

Paul H J Kelly, Olav Beckmann, Tony Field and Scott Baden, "Themis: Component dependence metadata in adaptive parallel applications". Parallel Processing Letters, Vol. 11, No. 4 (2001)

Conclusions

- Performance programming as a software engineering discipline
- The challenge of preserving abstractions
- The need to design-in the means to solve performance problems
- Adaptation to data-flow context
- Adaptation to platform/resources
- Adaptation to data values, sizes, shapes
- Making component composition explicit: build a plan, optimise it, execute it

Acknowledgements

- This work was funded by EPSRC
- Much of the work was done by colleagues and members of my research group, in particular
  - Olav Beckmann
  - Tony Field
- Students:
  - Alastair Houghton, Michael Mellor, Peter Fordham, Peter Liniker, Thomas Hansen