Cross-loop Optimization of Arithmetic Intensity and Data Locality for Unstructured Mesh Applications

Fabio Luporini
Imperial College London

Florian Rathgeber, Lawrence Mitchell, Andrew McRae, Doru Bercea, David Ham, Paul Kelly
Imperial College London
Istvan Reguly, Mike Giles
University of Oxford
Carlo Bertolli
IBM T. J. Watson (past Imperial College/Oxford)
Rod Tohid, J. “Ram” Ramanujam
Louisiana State University
Christopher Krieger, Catherine Olschanowsky, Michelle Strout
Colorado State University
Ana Lucia Varbanescu
University of Amsterdam

12 November 2014
We pursue solving PDEs, *fast*

Case we are particularly concerned with: on the fly weather forecast with *given time limit*
We pursue solving PDEs, fast

+ Raise the level of abstraction
  (through domain specific languages)

Stack of optimizing compilers

\[
\int \nabla \cdot \rho p \, dx
\]
This talk

- \( \int \nabla \cdot \rho p \, dx \) \rightarrow MAGIC from DSL for PDEs to loop chains
- MAGIC \rightarrow fast code tiling for unstructured meshes
- MAGIC \rightarrow fast code COFFEE: expression compiler
This talk

• $\int \nabla \cdot \rho \, p \, dx \rightarrow \text{MAGIC}$ from DSL for PDEs to loop chains

• $\text{MAGIC} \rightarrow \text{fast code}$ tiling for unstructured meshes

• $\text{MAGIC} \rightarrow \text{fast code}$ COFFEE: expression compiler

TALK’s MESSAGE (My group’s philosophy):
• getting the abstraction right enables implementing the MAGIC

• the MAGIC enables automatic powerful cross-loop optimization, which means faster code than you can get when writing it by hand and “having faith” in your favourite compiler
From the DSL to loop chains

Firedrake provides a DSL for finite element methods

phi, p = Function(mesh, ...)
...
while not convergence:
{
...
    phi -= dt / 2 * p
    if ...
        p += (assemble(dt*inner(nabla_grad(v),...)))*dx
    else:
        solve(...)  
    ...
    phi += dt / 2 * p
...
}
Firedrake provides a DSL for finite element methods

phi, p = Function(mesh, ...)
...
while not convergence:
{
...
 phi -= dt / 2 * p
if ...
  p += (assemble(dt*inner(nabla_grad(v),...)))*dx
else:
  solve(...)
...
 phi += dt / 2 * p
...
}
From the DSL to loop chains

Firedrake provides a DSL for finite element methods

\[
\text{phi, p} = \text{Function}(\text{mesh}, ...) \\
\]

... while not convergence:
{
  ...
  phi -= dt / 2 * p 
  if ...:
    p += (assemble(dt*inner(nabla_grad(v),...)))*dx
  else:
    \text{solve}(...)
  ...
  phi += dt / 2 * p
  ...
}

Loop over the mesh!
Loop over the mesh!
Loop over the mesh!
From the DSL to loop chains

Firedrake provides a DSL for finite element methods

\[
\phi, p = \text{Function}(\text{mesh}, ...) \\
\]

... while not convergence:
{
  ...
  \phi = \frac{dt}{2} \cdot p \\
  if ...
  
  p += \text{assemble}(dt \cdot \text{inner}(\text{nabla}\_\text{grad}(v), ...)) \cdot dx \\
  else:
  \text{solve}(...) \\
  ...
  \phi += \frac{dt}{2} \cdot p \\
  ...
}
From the DSL to loop chains

Firedrake provides a DSL for finite element methods

\[ \text{phi, p} = \text{Function}(\text{mesh}, \ldots) \]

... \[ \text{while not convergence:} \]

{ \[ \ldots \]

\[ \text{phi} = \frac{\text{dt}}{2} \ast \text{p} \]

\[ \text{if} \ldots: \]

\[ \text{p} \ast= \left( \text{assemble}(\text{dt} \ast \text{inner} (\text{nabla_grad} (\text{v}), \ldots)) \right) \ast \text{dx} \]

\[ \text{else:} \]

\[ \text{solve} (\ldots) \]

\[ \ldots \]

\[ \text{phi} \ast= \frac{\text{dt}}{2} \ast \text{p} \]

\[ \ldots \]

\[ } \]

...
The resulting non-affine parallel-loops chain

while not convergence:
{
  \textbf{forall cells}\n  ...
  for i
    for j
      ... expr(i, j)
  A[C[i]] = ... \\

  \textbf{forall edges}\n  A[E[i]] = ...

  ...

  function call !

  \textbf{forall cells}\n  ...
}
The resulting non-affine parallel-loops chain

while not convergence:
{
  forall cells
    ...
    for i
      for j
        ... expr(i, j)
        A[C[i]] = ...

  forall edges
    A[E[i]] = ...
    ...

  function call!

  forall cells
    ...
}
The resulting non-affine parallel-loops chain

```c
while not convergence:
{
  forall cells
    ...
    for i
      for j
        ... expr(i, j)
        A[C[i]] = ...
  forall edges
    A[E[i]] = ...
    ...
  function call!

  foreach cells
    ...
}
```

Dependencies through **indirect memory accesses** (C and E not known at compile time): break many compiler optimizations.

Computing **expr** can be so expensive, depending on the equation being solved, that the loop becomes compute-bound.
Towards tiling non-affine loops

while not convergence:
{
    forall cells
        ...
        for i
            for j
                ... expr(i, j)
                A[C[i]] = ...

    forall edges
        A[E[i]] = ...
        ...
        ...

    function call!

    forall cells
        ...
}
Towards tiling non-affine loops

while not convergence:
{
    forall cells
        ...
        for i
            for j
                ... expr(i, j)
                \( A[C[i]] = \ldots \)
    forall edges
        \( A[E[i]] = \ldots \)
        ...
    function call !
    forall cells
        ...
}
Towards tiling non-affine loops

while not convergence:
{
    forall cells
        ...
        for i
            for j
                ... expr(i, j)
                A[C[i]] = ...

    forall edges
        A[E[i]] = ...
        ...
        ...

    function call !

    forall cells
        ...
}
Generalized sparse tiling example

Par loop 1:  
\texttt{forall edges}  
read local data  
\texttt{increment adjacent vertices}

Par loop 2:  
\texttt{forall cells}  
read adjacent vertices  
write local data
Generalized sparse tiling example

Par loop 1:  
\begin{align*}
\text{forall } & \text{edges} \quad & \\
& \text{read local data} \quad & \\
& \text{increment adjacent vertices} \quad & 
\end{align*} 

Par loop 2:  
\begin{align*}
\text{forall } & \text{cells} \quad & \\
& \text{read adjacent vertices} \quad & \\
& \text{write local data} \quad & 
\end{align*}
Generalized sparse tiling example

Par loop 1:  

\texttt{forall edges}
\texttt{read local data}
\texttt{increment adjacent vertices}

Par loop 2:  

\texttt{forall cells}
\texttt{read adjacent vertices}
\texttt{write local data}
Generalized sparse tiling example

Par loop 1: \texttt{forall edges}
\hspace{1cm} read local data
\hspace{1cm} increment adjacent vertices

Par loop 2: \texttt{forall cells}
\hspace{1cm} read adjacent vertices
\hspace{1cm} write local data
Generalized sparse tiling example

Par loop 1:  
forall edges  
  read local data  
  increment adjacent vertices

Par loop 2:  
forall cells  
  read adjacent vertices  
  write local data
Generalized sparse tiling example

forall edges
read local data
**increment adjacent vertices**

forall cells
**read adjacent vertices**
write local data
Generalized sparse tiling example

I. Seed (shared) set partitioning

forall edges
read local data
**increment adjacent vertices**

forall cells
**read adjacent vertices**
write local data
Generalized sparse tiling example

1. **Seed (shared) set partitioning**

   - forall edges
     - read local data
       - *increment adjacent vertices*
   - forall cells
     - *read adjacent vertices*
     - write local data

Partitions (i.e. “base” tiles) fit the cache!
Generalized sparse tiling example

forall edges
read local data
\textbf{increment adjacent vertices}

Seed (shared) set partitioning

forall cells
\textbf{read adjacent vertices}
write local data

0. RED, 1 BLUE

1. Seed (shared) set partitioning and coloring
Lower color (number) => Higher scheduling priority
Generalized sparse tiling example

forall edges
    read local data
    \textbf{increment adjacent vertices}

Seed (shared) set partitioning

forall cells
    \textbf{read adjacent vertices}
    write local data

0. RED, 1 BLUE

1. Seed (shared) set partitioning and coloring
   Lower color (number) => Higher scheduling priority

Property after executing the red edges:
all red vertices are updated, while blue ones are not
Generalized sparse tiling example

1. Seed (shared) set partitioning and coloring
   Lower number => Higher scheduling priority

2. First loop over edges, data-flow analysis:
   assign \textbf{MIN} color over adjacent vertices => Property

forall edges
   read local data
   \textbf{increment adjacent vertices}

forall cells
   \textbf{read adjacent vertices}
   write local data

0. RED, 1 BLUE
Generalized sparse tiling example

1. Seed (shared) set partitioning and coloring
   Lower number => Higher scheduling priority

2. First loop over edges, data-flow analysis:
   assign MIN color over adjacent vertices => Property

3. Second loop over cells, data-flow analysis:
   Property => assign MAX color over adjacent vertices

forall edges
   read local data
   increment adjacent vertices

Seed (shared) set partitioning

forall cells
   read adjacent vertices
   write local data

0. RED, 1 BLUE
Parallel execution: the coloring problem

The longer the loop chain, the larger the tile expansion

forall edges

0. RED, 1 BLUE
Parallel execution: the coloring problem

The longer the loop chain, the larger the tile expansion

for all edges

0. RED, 1 BLUE
Parallel execution: the coloring problem

The longer the loop chain, the larger the tile expansion

forall edges

0. RED, 1 BLUE

Race conditions are now possible!
Parallel execution: the coloring problem

The longer the loop chain, the larger the tile expansion
Parallel execution: the coloring problem

The longer the loop chain, the larger the tile expansion

Solution: we color the \textit{k-distant mesh} instead (\(K = 2\) here)
Parallel execution: the coloring problem

The longer the loop chain, the larger the tile expansion

Solution: we color the $k$-distant mesh instead ($K = 2$ here)
Parallel execution: the coloring problem

The longer the loop chain, the larger the tile expansion

Solution: we color the $k$-distant mesh instead ($K = 2$ here)
Parallel execution: the coloring problem

The longer the loop chain, the larger the tile expansion

Solution: we color the k-distant mesh instead ($K = 2$ here)
Performance evaluation - Airfoil

- **Problem:**
  - Semi-structured mesh, ~700000 quadrilateral cells
  - ~1.11x over MPI (no NUMA issue!), including inspector cost
  - Time stepping loop unrolled, 6 loops tiled

- **Setup:**
  - Intel Sandy Bridge (dual-socket 8-core Xeon E5-2680)
  - Intel compiler 13, -xAVX, -O3, -xHost
Preliminary study of Volna
Collaboration with Rod Tohid, LSU

• Movie: tiny, structured mesh, just for illustration purpose (real mesh, which we will use, is refined, fully unstructured and made of 1.5M vertices)

• In the time stepping loop, two loop chains broken by a global reduction. Here, tiled the second one, made of 5 loops (edges, cells, edges, cells, edges)
Preliminary study of Volna
Collaboration with Rod Tohid, LSU

• Movie: tiny, structured mesh, just for illustration purpose (real mesh, which we will use, is refined, fully unstructured and made of 1.5M vertices)

• In the time stepping loop, two loop chains broken by a global reduction. Here, tiled the second one, made of 5 loops (edges, cells, edges, cells, edges)
Towards optimizing arithmetic intensity in FEM assembly

while not convergence:
{
    forall cells
        ... expr(i, j)
        A[C[i]] = ...

    forall edges
        A[E[i]] = ...
        ...

    function call!

    forall cells
        ...
}
Towards optimizing arithmetic intensity in FEM assembly

while not convergence:
{
    
    forall cells
    ... 
    
    for i
    for j
    ... expr(i, j)
    A[C[i]] = ...

    forall edges
    A[E[i]] = ...
    ...

    function call!

    forall cells
    ...
}

- Loop tiling is a general technique (i.e. not bound to any specific domain). Now focus on finite element.

- FEM execution time $\sim$ assembly + solver (fun call)

- Context: Firedrake, where we rely on automated code generation, i.e. we abstract from the specific equation!
while not convergence:
{
    forall cells
    ... for i for j ... expr(i, j)
    A[C[i]] = ...

    forall edges
    A[E[i]] = ...
    ...

    function call !

    forall cells ...
}
How did we get that “expr”?

\[ A^K_{ij} = \int_K w \nabla \phi^K_i \cdot \nabla \phi^K_j \, dx \]

Many per equation (from “weak form”), one parallel loop each

The numerical evaluation (quadrature) leads to \textit{expr}.

\textbf{forall} cells
\[ \ldots A[C[i]] \ldots \]
\textbf{for} \( i \)
\textbf{for} \( j \)
\[ \ldots \text{expr}(i, j) \]
Some examples to “feel” the problem complexity

$m, n, o$ rarely greater than 30
typically between 3 and 15

...  
...  
for (int ip = 0; ip < m; ++ip) {
    ...
    for (int j = 0; j < n; ++j) {
        for (int k = 0; k < o; ++k) {
        }
    }
}  
...
Some examples to “feel” the problem complexity

... m, n, o rarely greater than 30 typically between 3 and 15

for (int ip = 0; ip < m; ++ip) {
    ...
}
for (int j = 0; j < n; ++j) {
    for (int k = 0; k < o; ++k) {
    }
}
...
for (int ip = 0; ip < m; ++ip) {
    for (int j = 0; j < n; ++j) {
        for (int k = 0; k < o; ++k) {
            }  
        }  
    }  
}  

Hyperelasticity operator

Some examples to “feel” the problem complexity

for (int ip = 0; ip < m; ++ip) {
    for (int j = 0; j < n; ++j) {
        for (int k = 0; k < o; ++k) {
            }  
        }  
    }  
}  

Hyperelasticity operator
Some examples to “feel” the problem complexity

```
for (int ip = 0; ip < m; ++ip) {
    for (int j = 0; j < n; ++j) {
```

```
... 
```

```

```
What do we have to do with such monsters?

I’ll use an extremely simplified example. Key questions:
- Common sub-expressions
- Loop-invariants
- Re-association and factorization
- Vectorization
What do we have to do with such monsters?

I'll use an extremely simplified example. Key questions:

- Common sub-expressions
- Loop-invariants
- Re-association and factorization
- Vectorization

What a compiler can do for us?
What do we have to do with such monsters?

I'll use an extremely simplified example. Key questions:
- Common sub-expressions
- Loop-invariants
- Re-association and factorization
- Vectorization

What a compiler can do for us?
Need to be tackled “jointly”, not individually
What do we have to do with such monsters?

I'll use an extremely simplified example. Key questions:

- Common sub-expressions
- Loop-invariants
- Re-association and factorization
- Vectorization

What a compiler can do for us?

Need to be tackled “jointly”, not individually

Example:

\[ A[i] = B[i] + C[i] \]

Vectorization
What do we have to do with such monsters?

I'll use an extremely simplified example. Key questions:

- Common sub-expressions
- Loop-invariants
- Re-association and factorization
- Vectorization

What a compiler can do for us?

Need to be tackled “jointly”, not individually

Example:

\[ A[i] = B[i] + C[i] \]

Vectorization  Loop invariants
What do we have to do with such monsters?

I'll use an extremely simplified example. Key questions:

- Common sub-expressions
- Loop-invariants
- Re-association and factorization
- Vectorization

What a compiler can do for us?

Need to be tackled “jointly”, not individually

Example:

\[ A[i] = B[i] + C[i] \]

Vectorization  \[\rightarrow\]  Loop invariants  \[\leftarrow\]  Re-association/factorization
What do we have to do with such monsters?

I'll use an extremely simplified example. Key questions:

- Common sub-expressions
- Loop-invariants
- Re-association and factorization
- Vectorization

What a compiler can do for us?

Need to be tackled “jointly”, not individually

Example:

A[i] = B[i] + C[i]

Small loops require special attention!
Cross-loop optimization of arithmetic intensity

for i
   for j
      for k
         \[ A[j][k] += B[i][j] \times C[i][k] + (E[i][j] \times \beta + F[i][j] \times \gamma) + (B[i][j] \times D[i][k]) \times \alpha \]
Cross-loop optimization of arithmetic intensity

for i
  for j
    for k
      A[j][k] += B[i][j] * C[i][k] + (E[i][j]*β + F[i][j]*γ) + (B[i][j] * D[i][k])*α

for i
  for j
    for k
      A[j][k] += B[i][j] * C[i][k] + (E[i][j]*β + F[i][j]*γ) + (B[i][j] * D[i][k])*α

Innermost-loop invariant
Cross-loop optimization of arithmetic intensity

for i
  for j
    tmp = (E[i][j] * β + F[i][j] * γ)
  for k
               (B[i][j] * D[i][k]) * α

OK, compilers do this easily…
Cross-loop optimization of arithmetic intensity

for i
for j
    tmp = (E[i][j]*β + F[i][j]*γ)
for k
             (B[i][j] * D[i][k])α

OK, compilers do this easily…

… but need promotion for vectorization!
Important because small loops

for i
for j
    TMP[j] = (E[i][j]*β + F[i][j]*γ)
for j
for k
    A[j][k] += B[i][j] * C[i][k] + TMP[j] +
             (B[i][j] * D[i][k])α
Cross-loop optimization of arithmetic intensity

for i
  for j
    TMP[j] = (E[i][j] * β + F[i][j] * γ)
  for j
    for k
      A[j][k] += B[i][j] * C[i][k] + TMP[j] +
                  (B[i][j] * D[i][k]) * α
Cross-loop optimization of arithmetic intensity

for i
  for j
    TMP[j] = (E[i][j] * β + F[i][j] * γ)
  for j
    for k
      A[j][k] += B[i][j] * C[i][k] + TMP[j] + (B[i][j] * D[i][k]) * α

for i
  for j
    TMP[j] = (E[i][j] * β + F[i][j] * γ)
  for j
    for k
Cross-loop optimization of arithmetic intensity

for i
  for j
    TMP[j] = (E[i][j]*β + F[i][j]*γ)
  for j
    for k
      A[j][k] += B[i][j] * (C[i][k] + D[i][k]*α) + TMP[j]
Cross-loop optimization of arithmetic intensity

for i
  for j
    TMP[j] = (E[i][j]*β + F[i][j]*γ)
  for j
    for k
      A[j][k] += B[i][j] * (C[i][k] + D[i][k]*α) + TMP[j]

  Outer-loop invariant: no way your compiler thinks “globally”
Cross-loop optimization of arithmetic intensity

for i
  for j
    TMP[j] = (E[i][j]*β + F[i][j]*γ)
  for j
    for k
      A[j][k] += B[i][j] * (C[i][k] + D[i][k]*α) + TMP[j]

Outer-loop invariant: no way your compiler thinks “globally”

for i
  for j
    TMP[j] = (E[i][j]*β + F[i][j]*γ)
  for k
    TMP2[k] = (C[i][k] + D[i][k]*α)
  for j
    for k
Cross-loop optimization of arithmetic intensity

for i
for j
for k
  \[ A[j][k] += B[i][j] \ast C[i][k] + (E[i][j]*\beta + F[i][j]*\gamma) + (B[i][j] \ast D[i][k])*\alpha \]
Cross-loop optimization of arithmetic intensity

for i
  for j
    for k
      A[j][k] += B[i][j] * C[i][k] + (E[i][j]*\beta + F[i][j]*\gamma) + (B[i][j] * D[i][k])*\alpha

for i
  for j
    TMP[j] = (E[i][j]*\beta + F[i][j]*\gamma)
  for k
    TMP2[k] = (C[i][k] + D[i][k]*\alpha)
  for j
    for k
Cross-loop optimization of arithmetic intensity

for i
  for j
    for k
      A[j][k] += B[i][j] * C[i][k] + (E[i][j]*β + F[i][j]*γ) + (B[i][j] * D[i][k])*α

for i
  for j
    TMP[j] = (E[i][j]*β + F[i][j]*γ)
  for k
    TMP2[k] = (C[i][k] + D[i][k]*α)
  for j
    for k
Cross-loop optimization of arithmetic intensity

for i
    for j
        for k
            A[j][k] += B[i][j] * C[i][k] + (E[i][j] * β + F[i][j] * γ) +
                        (B[i][j] * D[i][k]) * α

Padding and data alignment?

for i
    for j
        TMP[j] = (E[i][j] * β + F[i][j] * γ)
for k
    TMP2[k] = (C[i][k] + D[i][k] * α)
for j
    for k
Cross-loop optimization of arithmetic intensity

for i
  for j
    for k
      A[j][k] += B[i][j] * C[i][k] + (E[i][j]*β + F[i][j]*γ) + (B[i][j] * D[i][k])*α

for i
  for j
    TMP[j] = (E[i][j]*β + F[i][j]*γ)
  for k
    TMP2[k] = (C[i][k] + D[i][k]*α)
  for j
    for k
<table>
<thead>
<tr>
<th>PDE 1</th>
<th>PDE 2</th>
<th>PDE 3</th>
<th>PDE 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>Navier Stokes</td>
<td>Helmholtz</td>
<td>Elasticity</td>
<td>...</td>
</tr>
<tr>
<td>PDE 1</td>
<td>PDE 2</td>
<td>PDE 3</td>
<td>PDE 4</td>
</tr>
<tr>
<td>------------</td>
<td>------------</td>
<td>------------</td>
<td>------------</td>
</tr>
<tr>
<td>Navier Stokes</td>
<td>Helmholtz</td>
<td>Elasticity</td>
<td>...</td>
</tr>
</tbody>
</table>

ANY PARAMETRIZATION YOU LIKE
element, poly order, coefficients, etc.
The COFFEE Project

PDE 1
Navier Stokes

PDE 2
Helmholtz

PDE 3
Elasticity

PDE 4
...

ANY PARAMETRIZATION YOU LIKE
element, poly order, coefficients, etc.

COFFEE
COmpiler For Fast Expression Evaluation
used in Firedrake trunk!
FEM-independent, 5000 lines of code
The COFFEE Project

PDE 1
Navier Stokes

PDE 2
Helmholtz

PDE 3
Elasticity

PDE 4
...

ANY PARAMETRIZATION YOU LIKE
element, poly order, coefficients, etc.

COFFEE
COmpiler For Fast Expression Evaluation
used in Firedrake trunk!
FEM-independent, 5000 lines of code

HW 1:
Haswell CPU

HW 2:
Tesla GPU

HW 3:
XeonPhi

HW 4:
...
• Problem:
  • linear elasticity with $f=1$ and $f=2$ coefficient functions
  • polynomial order 1 (left fig) and 2 (right fig)
  • mesh: tetrahedral, 196608 elements (CG family)
  • max application speedup: 1.47x (but grows with complexity of equation!)

• Setup:
  • Single core of an Intel Sandy Bridge (I7-2600 CPU @ 3.40GHz)
  • Intel compiler (version 13.1, -O3, -xAVX, -ip, -xHost)
• Problem:
  • hyperelasticity, with $f=0$ and $f=1$ coefficient functions
  • polynomial order 3
  • mesh: small enough to fit the L2 cache of the architecture
  • Original, FEniCS-optimized, COFFEE-optimized, COFFEE-autotuned

• Setup:
  • Single core of an Intel Sandy Bridge (i7-2600 CPU @ 3.40GHz)
  • Intel compiler (version 13.1, -O3, -xAVX, -ip, -xHost)
Conclusions and Future Work

• What I’ve shown you is implemented in **real tools**. COFFEE, in particular, is used in Firedrake trunk and automatically does the expression manipulation discussed (plus lots of other stuff!).

• Generalized Sparse Tiling is an on-going project. Rod Tohid (Louisiana State University) is tackling new problems (MPI) and working on new, real-world applications (VOLNA).

• Combining **domain-specific** and **technology** knowledge allows you to deliver optimizations more powerful than you can write by hand

• Where are we going now?
  • Generalized Sparse Tiling => Overlapped tiling? Hard fusion?
  • COFFEE on manycores
Thanks!

- Questions -
Spare slides
Unstructured meshes used for discretization

- To discretize a PDE’s domain
- “Unstructured” implies the mesh connectivity can be practically expressed only through arrays of indices (e.g. $A[B[i]]$)
- Same program applied to different meshes, so the mesh (connectivity) is known only at run-time.
Goal: improving cache locality on CPUs

- In the original OP2 execution model, parallel loops are executed one after the other.
- We break this execution model by determining group of tiles spanning multiple loops, in a way that data dependencies are satisfied.
Goal: improving cache locality on CPUs

- In the original OP2 execution model, parallel loops are executed one after the other.
- We break this execution model by determining group of tiles spanning multiple loops, in a way that data dependencies are satisfied.

A possibility is overlapped tiling (also known as communication avoiding).
Goal: improving cache locality on CPUs

- In the original OP2 execution model, parallel loops are executed one after the other.
- We break this execution model by determining a group of tiles spanning multiple loops, in a way that data dependencies are satisfied.

We rather focus on so-called Sparse Tiling, in which we explicitly keep track of tile dependencies.
void incrVertices (double* e, double* v1, double* v2)
{
    *v1 += *e;
    *v2 += *e;
}

op_par_loop (incrVertices, edges,
    op_arg_dat (edgesDat, -1, OP_ID, OP_READ),
    op_arg_dat (vertexDat, 0, edges2vertices, OP_INC),
    op_arg_dat (vertexDat, 1, edges2vertices, OP_INC));

The right abstraction simplifies the analysis!
void incrVertices(
    double* e,
    double* v1,
    double* v2)
{
    *v1 += *e;
    *v2 += *e;
}

op_par_loop (incrVertices, edges,
    op_arg_dat (edgesDat, -1, OP_ID,          OP_READ),
    op_arg_dat (vertexDat, 0, edges2vertices, OP_INC),
    op_arg_dat (vertexDat, 1, edges2vertices, OP_INC));

The right abstraction simplifies the analysis!
void incrVertices(
    double* e,
    double* v1,
    double* v2)
{
    *v1 += *e;
    *v2 += *e;
}

void incrVertices
    (double* e,
     double* v1,
     double* v2)
{
    *v1 += *e;
    *v2 += *e;
}

op_par_loop (incrVertices, edges,
    op_arg_dat (edgesDat, -1, OP_ID, OP_READ),
    op_arg_dat (vertexDat, 0, edges2vertices, OP_INC),
    op_arg_dat (vertexDat, 1, edges2vertices, OP_INC));
void incrVertices(
    double* e,
    double* v1,
    double* v2)
{
    *v1 += *e;
    *v2 += *e;
}

op_par_loop (incrVertices, edges,
   op_arg_dat (edgesDat, -1, OP_ID,          OP_READ),
   op_arg_dat (vertexDat, 0, edges2vertices, OP_INC),
   op_arg_dat (vertexDat, 1, edges2vertices, OP_INC));

The right abstraction simplifies the analysis!
The COFFEE Project

• Embedded and actually used in Firedrake master!

• Could be integrated with FEniCS, because both framework use the same DSL compiler

• Therefore, potentially, a user space of ~1000 scientists!

• Of course, a lot still has to be done

• Source code is >5000 lines of Python code, and is becoming finite element independent