Effective Resource-Driven Loop Splitting for Large Unstructured Mesh Applications on GPUs

rand (Rod Tohid, *J. "Ram" Ramanujam*
Louisiana State University
Fabio Luporini, Adam Betts, Florian Rathgeber,
Graham Markall, David A. Ham, Paul Kelly
Imperial College London
Carlo Bertolli
IBM T.J. Watson
Istvan Reguly, Gihan Mudalige, Michael B. Giles)
University of Oxford

HiStencils'15 20 January 2015

Goal: fast, automated resolution of PDEs

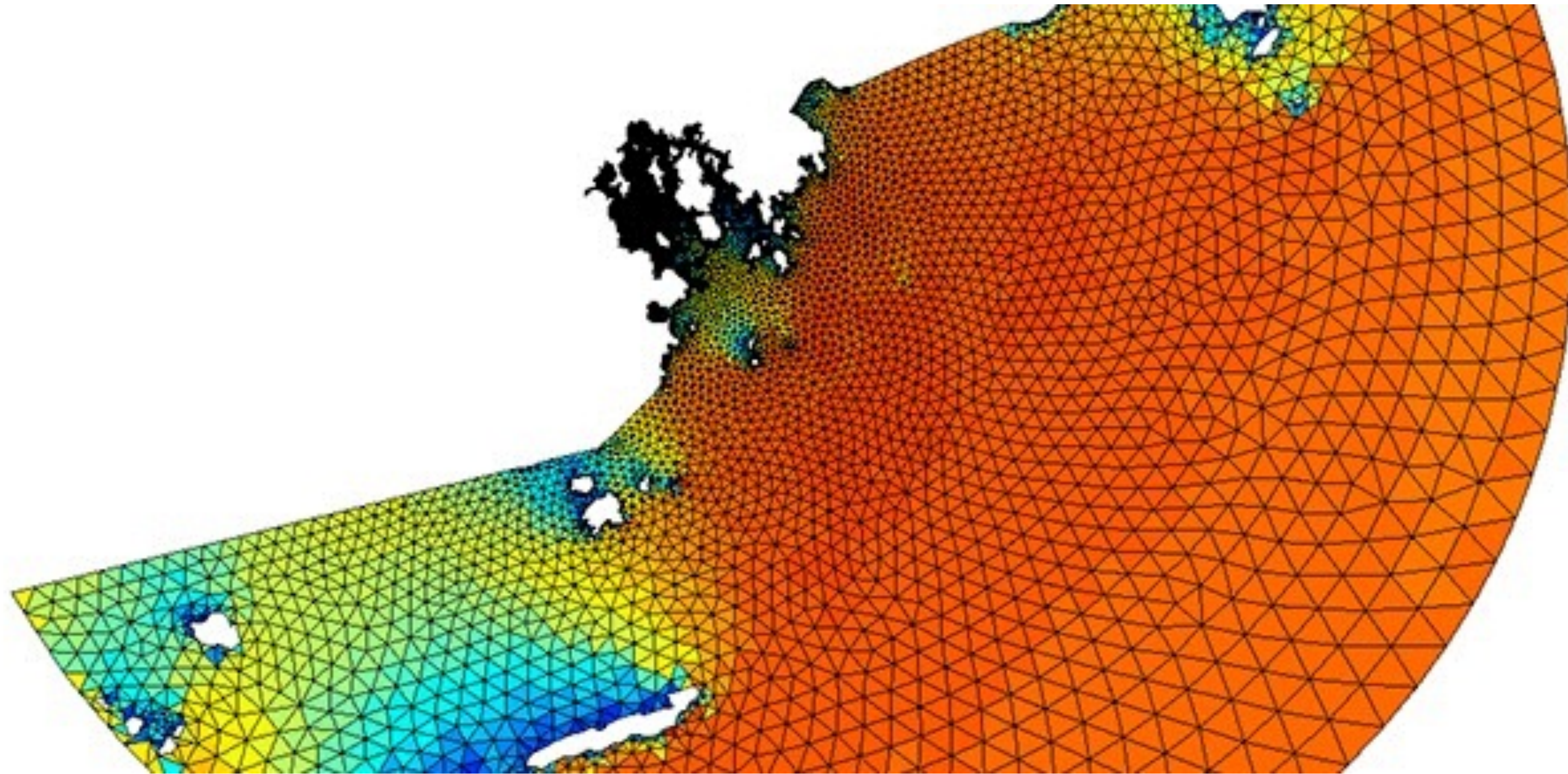


Image publicly available from http://www.bmtargoss.com/

Real-world computations we are interested in:

- weather forecast in a given time window (e.g., 1 hour)
- simulation turbo-machinery of aircraft engines (HYDRA)
- simulation of tsunami waves (VOLNA)
- ...

Goal: fast, automated resolution of PDEs

Goal: fast, automated resolution of PDEs

+

Raise the level of abstraction

(through domain specific languages)

$$\int \nabla \cdot \rho \ p \ dx$$

Goal: fast, automated resolution of PDEs

+

Raise the level of abstraction
(through domain specific languages)

$$\int \nabla \cdot \rho \; p \; dx$$

+

Stack of optimizing compilers

Goal: fast, automated resolution of PDEs

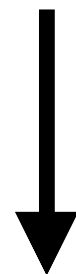
+

Raise the level of abstraction
(through domain specific languages)

$$\int \nabla \cdot \rho \; p \; dx$$

+

Stack of optimizing compilers



faster code than you can reasonably write “by hand”

From DSL to loop chains

Firedrake provides a DSL for finite element methods

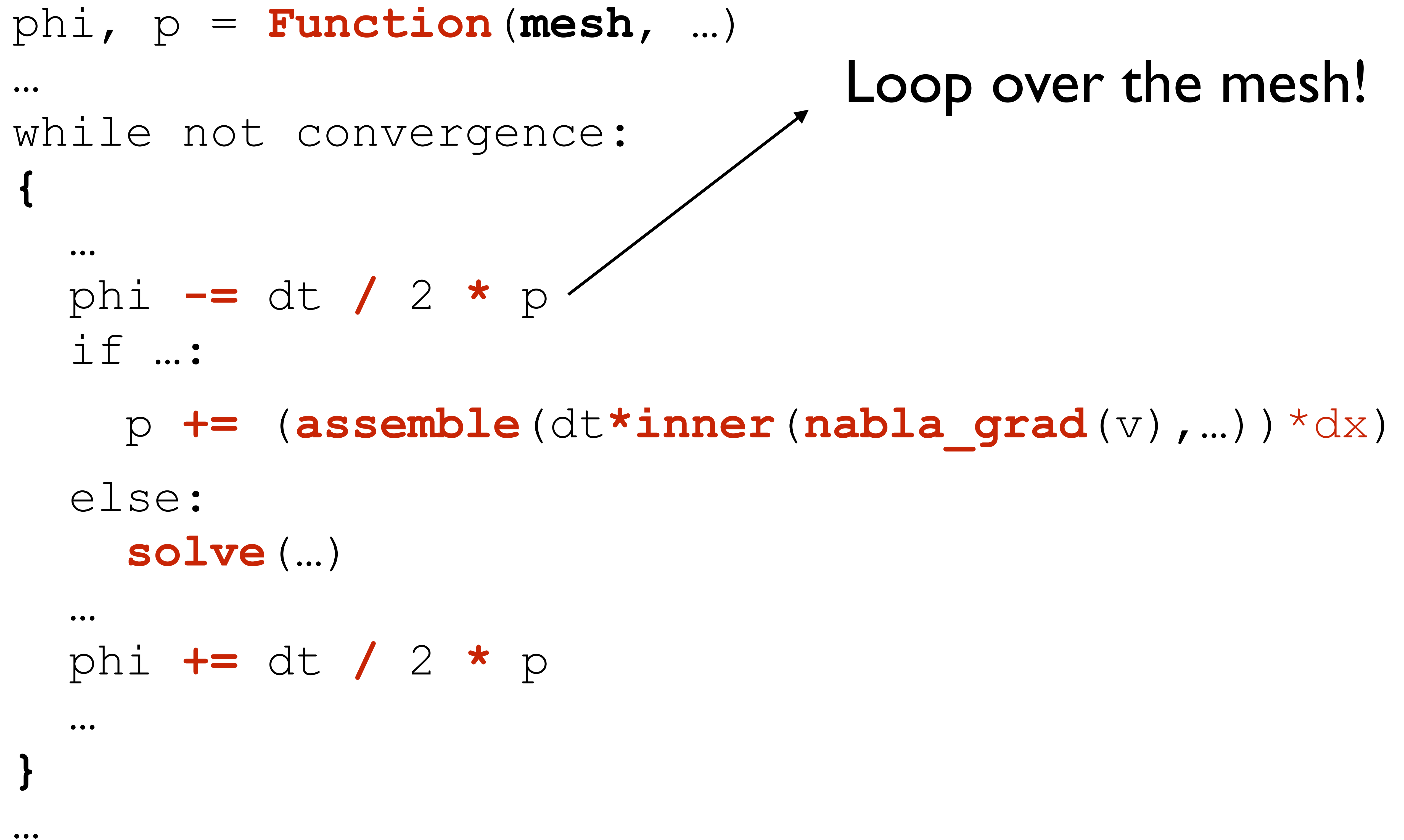
```
phi, p = Function(mesh, …)
…
while not convergence:
{
   …
   phi -= dt / 2 * p
   if …:

      p += (assemble(dt*inner(nabla_grad(v),…))*dx)

   else:
      solve(…)
   …
   phi += dt / 2 * p
   …
}
…
```

From DSL to loop chains

Firedrake provides a DSL for finite element methods



From 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
   …
}
…
```

Loop over the mesh!

Loop over the mesh!

From 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
   …
}
…
```

Loop over the mesh!

Loop over the mesh!

Loop over the mesh!

From 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
   …
}
…
```

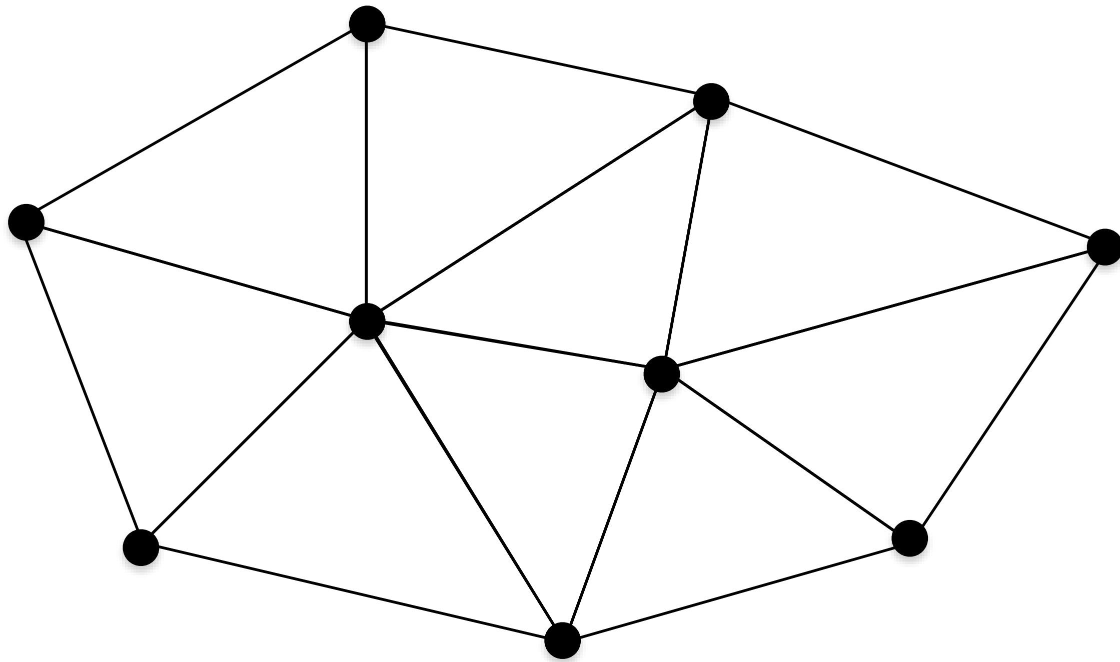
Loop over the mesh!

Loop over the mesh!

Call to third party library!

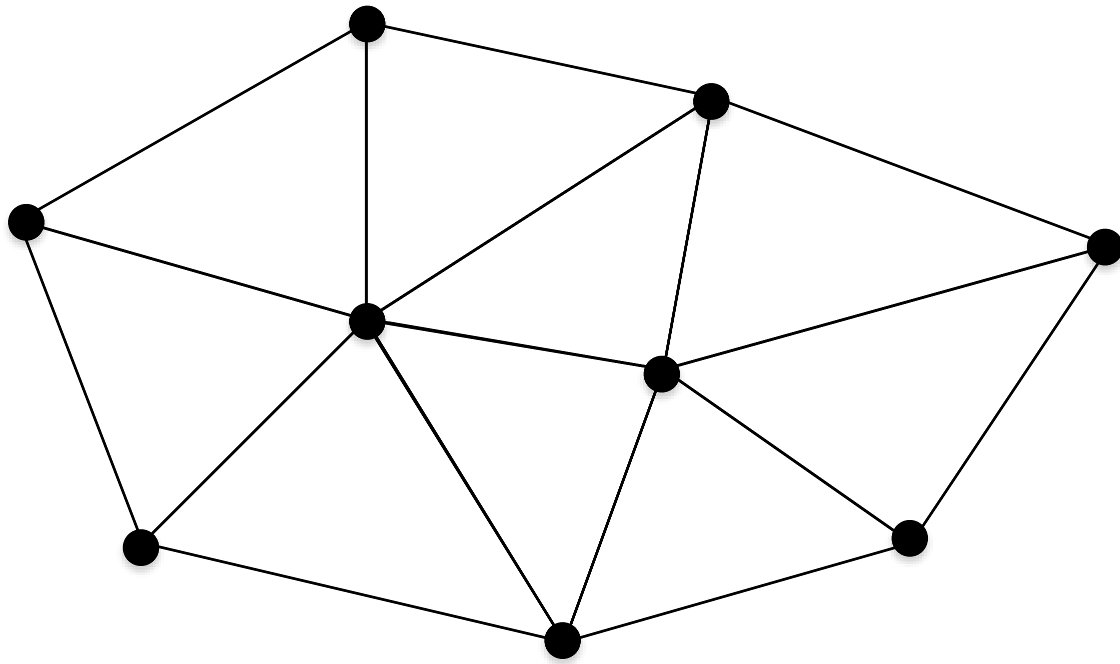
Loop over the mesh!

The OP2 programming model



```
void incrVertices (
  double* e_weight,
  double* v1,
  double* v2) {
    *v1 += f(e_weight)
    *v2 += f(e_weight)
}
```

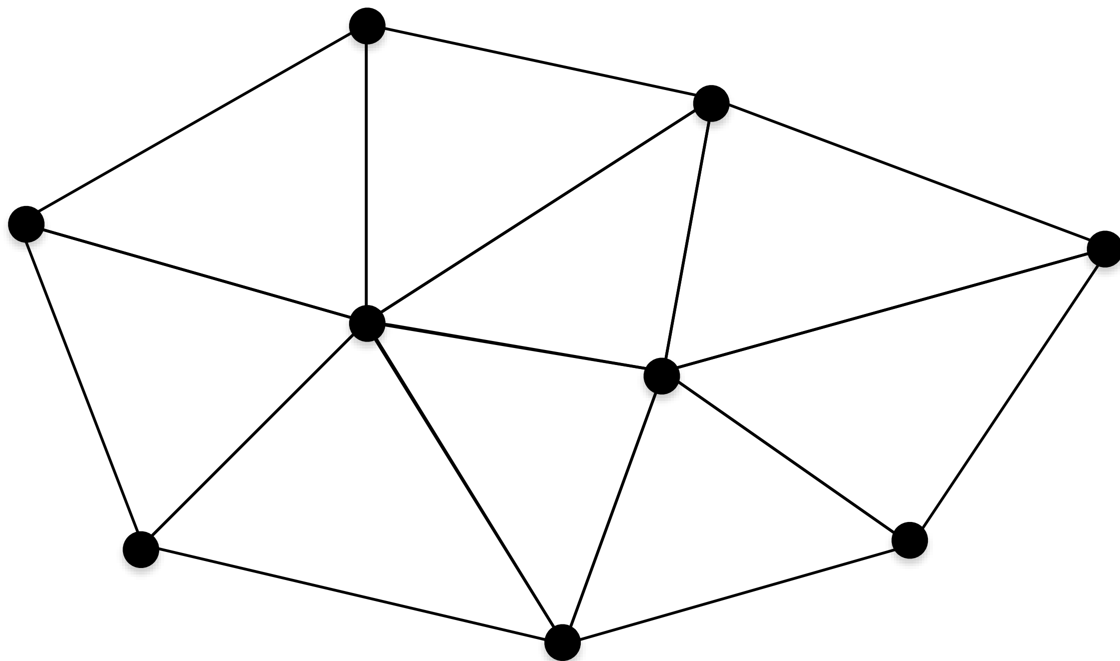
The OP2 programming model



```
void incrVertices (
   double* e_weight,
   double* v1,
   double* v2) {
      *v1 += f(e_weight)
      *v2 += f(e_weight)
}
```

```
op_par_loop (incrVertices, edges,
```

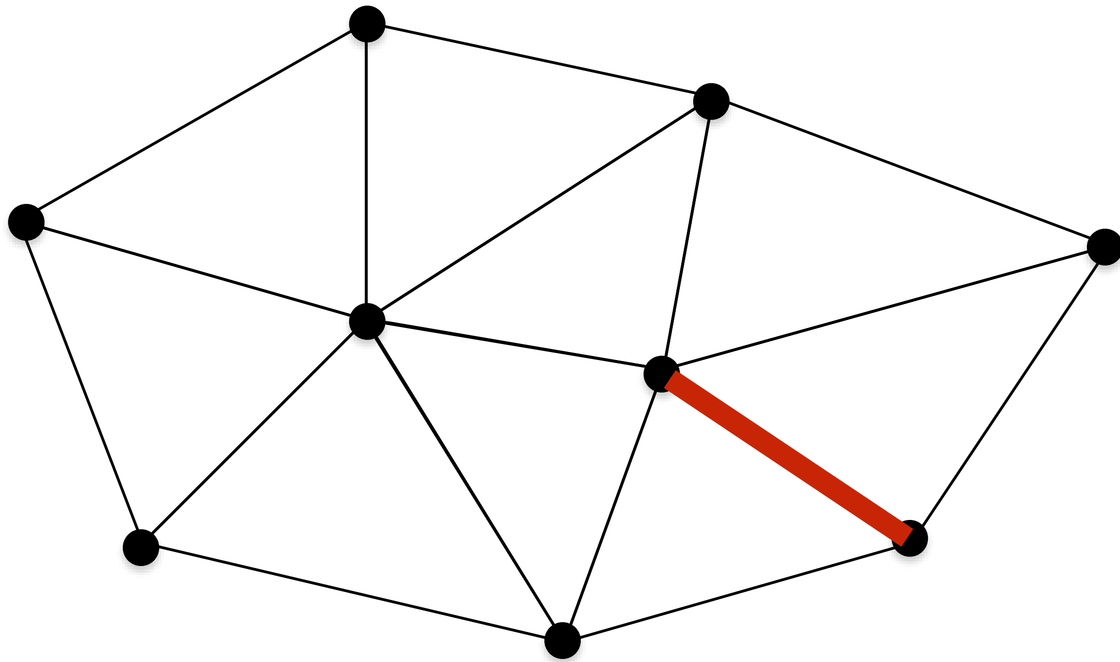
The OP2 programming model



```
void incrVertices (
   double* e_weight,
   double* v1,
   double* v2) {
     *v1 += f(e_weight)
     *v2 += f(e_weight)
}
```

```
op_par_loop (incrVertices, edges,
```

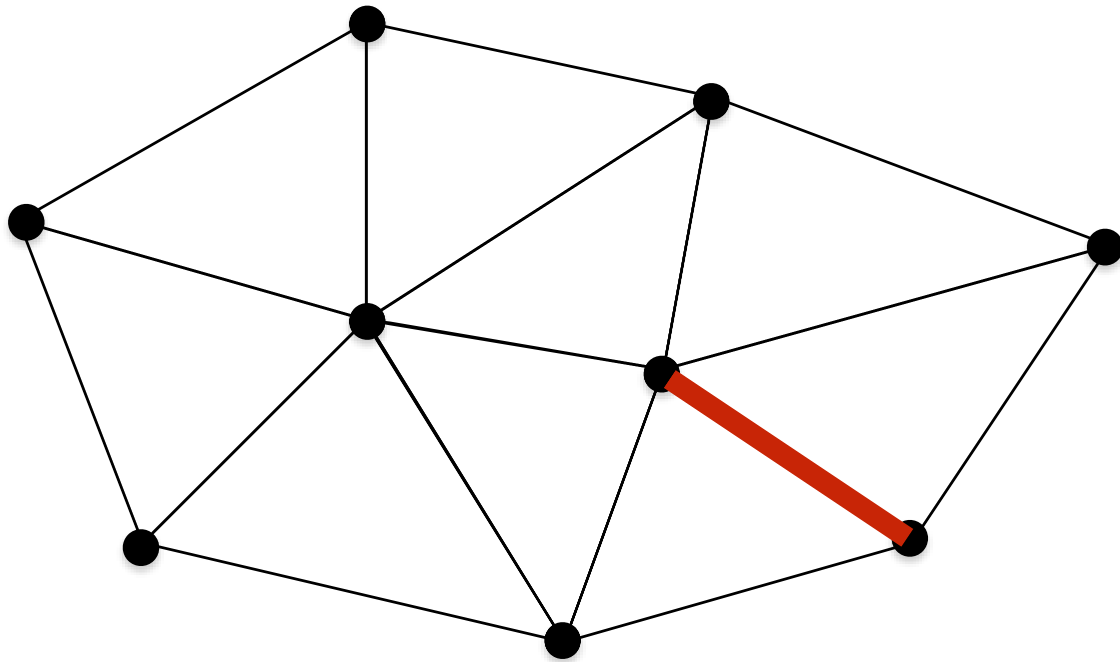
The OP2 programming model



```
void incrVertices (
    double* e_weight,
    double* v1,
    double* v2) {
      *v1 += f(e_weight)
      *v2 += f(e_weight)
}
```

```
op_par_loop (incrVertices, edges,
```

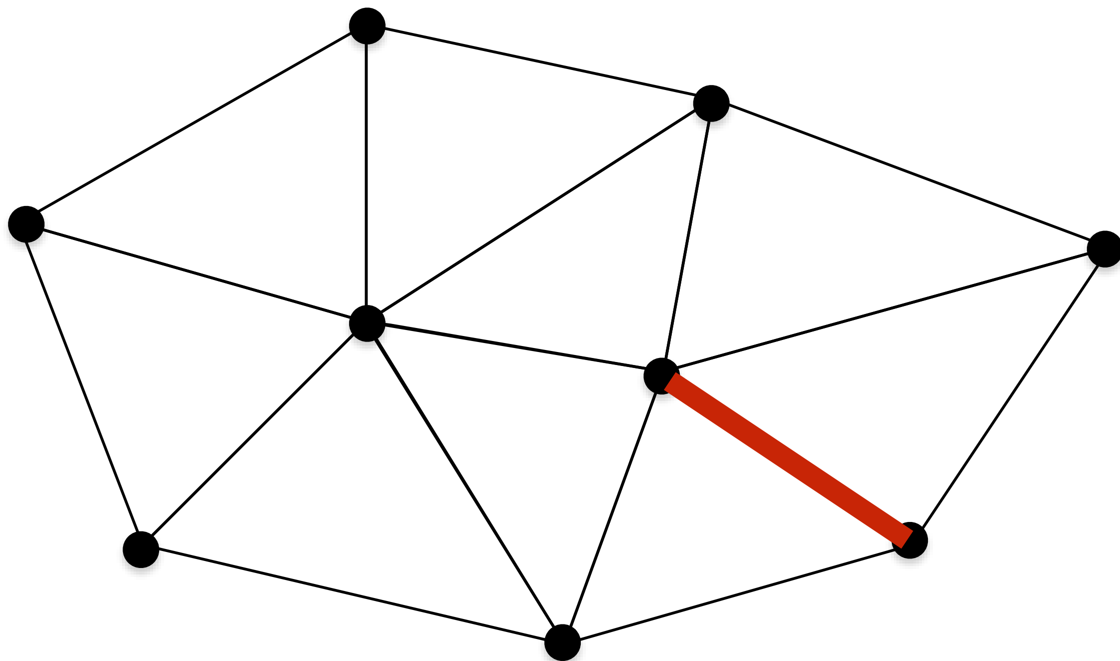
The OP2 programming model



```
void incrVertices (
   double* e_weight,
   double* v1,
   double* v2) {
      *v1 += f(e_weight)
      *v2 += f(e_weight)
}
```

```
op_par_loop (incrVertices, edges,
  op_arg_dat (edgeWeight, -1, OP_ID,          OP_READ),
```

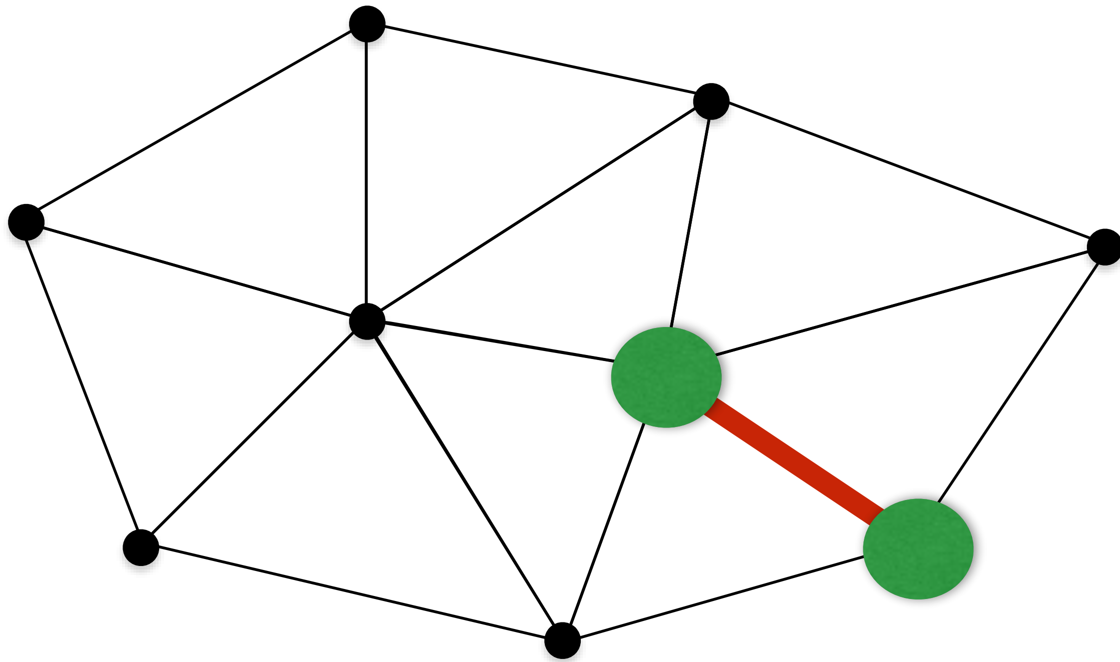
The OP2 programming model



```
void incrVertices (
    double* e_weight,
    double* v1,
    double* v2) {
      *v1 += f(e_weight)
      *v2 += f(e_weight)
}
```

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

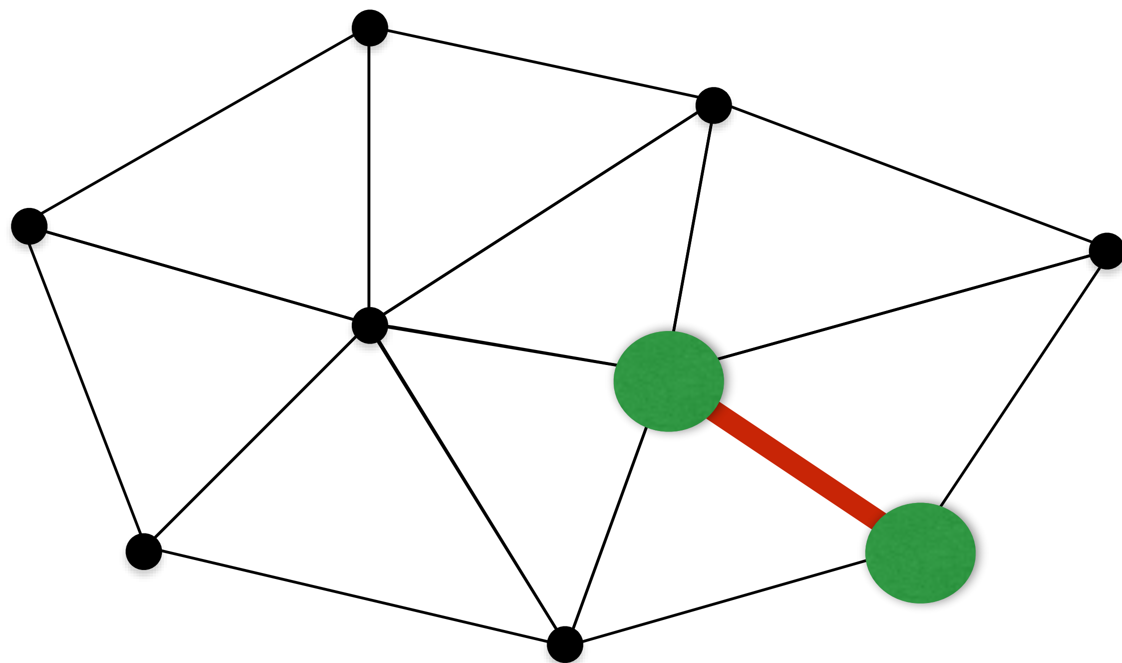
The OP2 programming model



```
void incrVertices (
   double* e_weight,
   double* v1,
   double* v2) {
     *v1 += f(e_weight)
     *v2 += f(e_weight)
}
```

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

The OP2 programming model

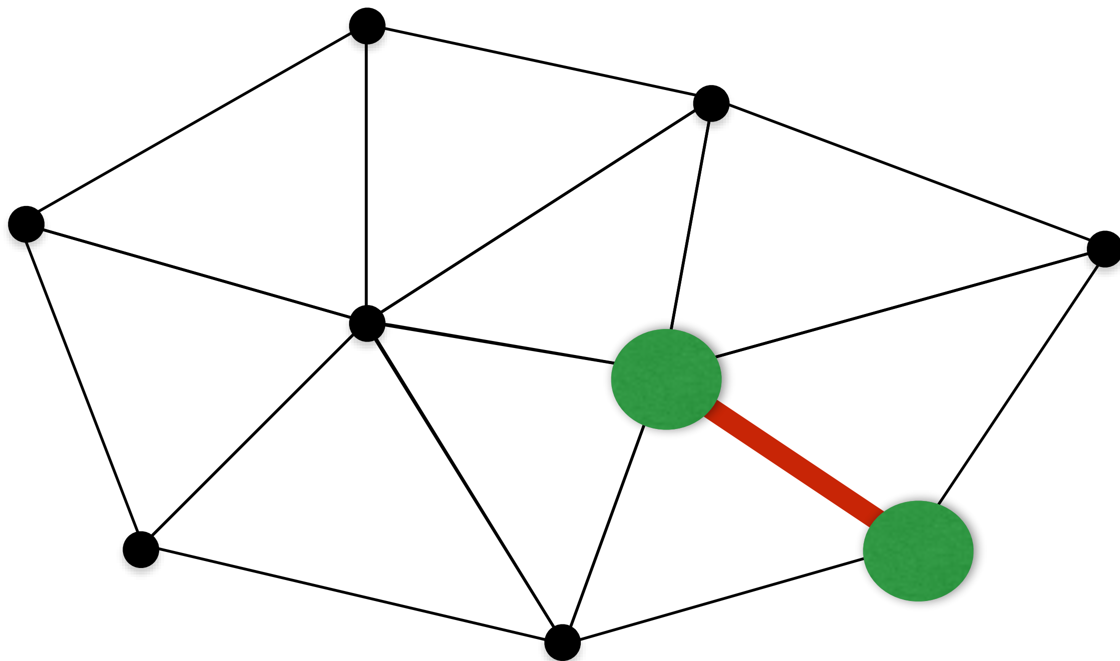


```
void incrVertices (
   double* e_weight,
   double* v1,
   double* v2) {
     *v1 += f(e_weight)
     *v2 += f(e_weight)
}
```

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

INDIRECT MEMORY ACCESSES (A[B[i]])!

The OP2 programming model



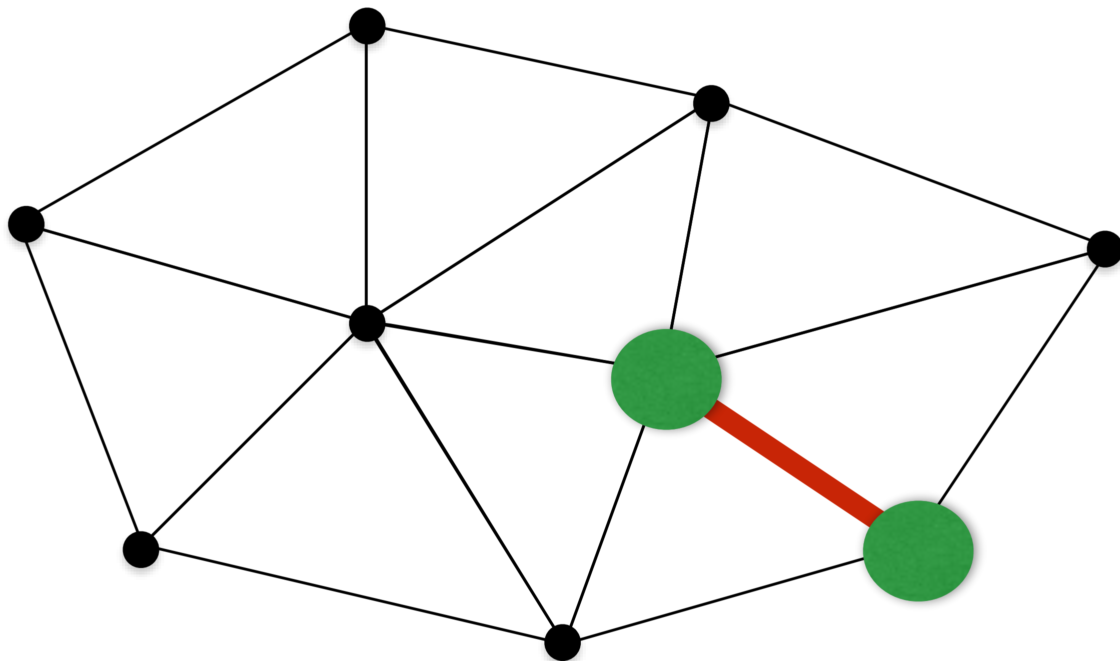
```
void incrVertices (
   double* e_weight,
   double* v1,
   double* v2) {
     *v1 += f(e_weight)
     *v2 += f(e_weight)
}
```

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

INDIRECT MEMORY ACCESSES (A[B[i]])!

```
op_par_loop (X, cells, …)
```

The OP2 programming model



```
void incrVertices (
    double* e_weight,
    double* v1,
    double* v2) {
      *v1 += f(e_weight)
      *v2 += f(e_weight)
}
```

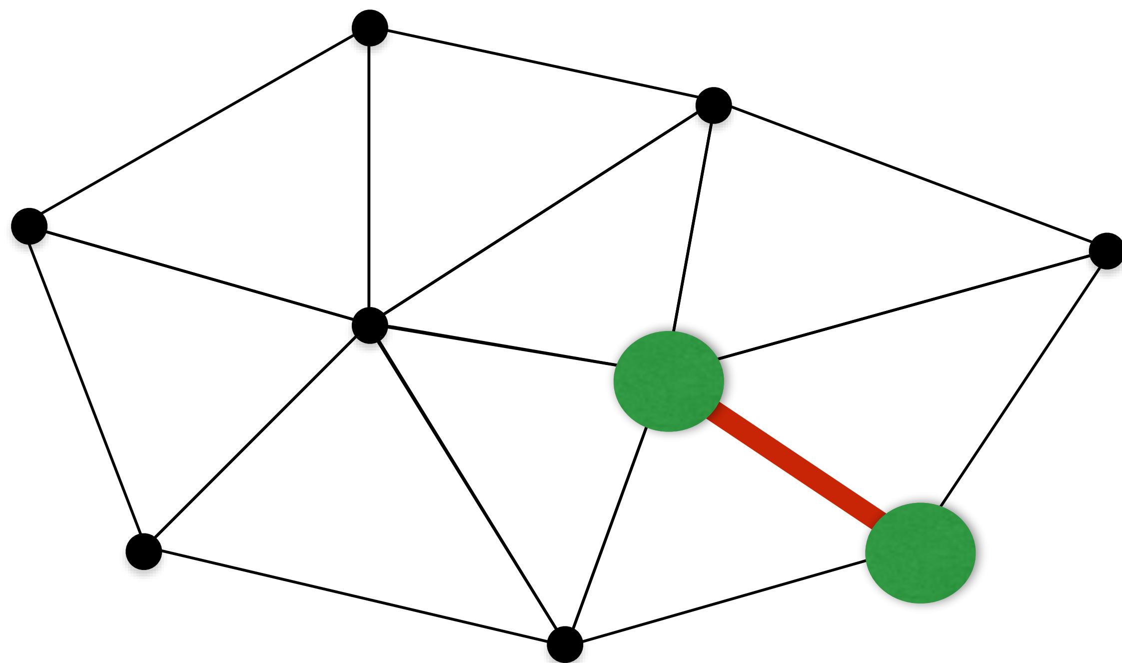
```
op_par_loop (incrVertices, edges,
  op_arg_dat (edgeWeight, -1, OP_ID,               OP_READ),
  op_arg_dat (vertexDat, 0, edges2vertices, OP_INC),
  op_arg_dat (vertexDat, 1, edges2vertices, OP_INC));
```

INDIRECT MEMORY ACCESSES (A[B[i]])!

```
op_par_loop (X, cells, …)
```

Synchronization point (function call e.g., PETSc)

The OP2 programming model



```
void incrVertices (
   double* e_weight,
   double* v1,
   double* v2) {
     *v1 += f(e_weight)
     *v2 += f(e_weight)
}
```

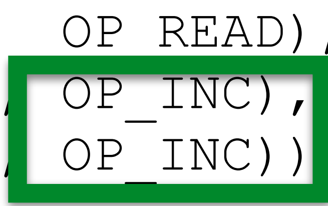
```
op_par_loop (incrVertices, edges,
  op_arg_dat (edgeWeight, -1, OP_ID,               OP_READ),
  op_arg_dat (vertexDat, 0, edges2vertices,  OP_INC),
  op_arg_dat (vertexDat, 1, edges2vertices,  OP_INC));
```

INDIRECT MEMORY ACCESSES (`A[B[i]]`)!

```
op_par_loop (X, cells, …)
Synchronization point (function call e.g., PETSc)
op_par_loop (Y, vertices, …)
```

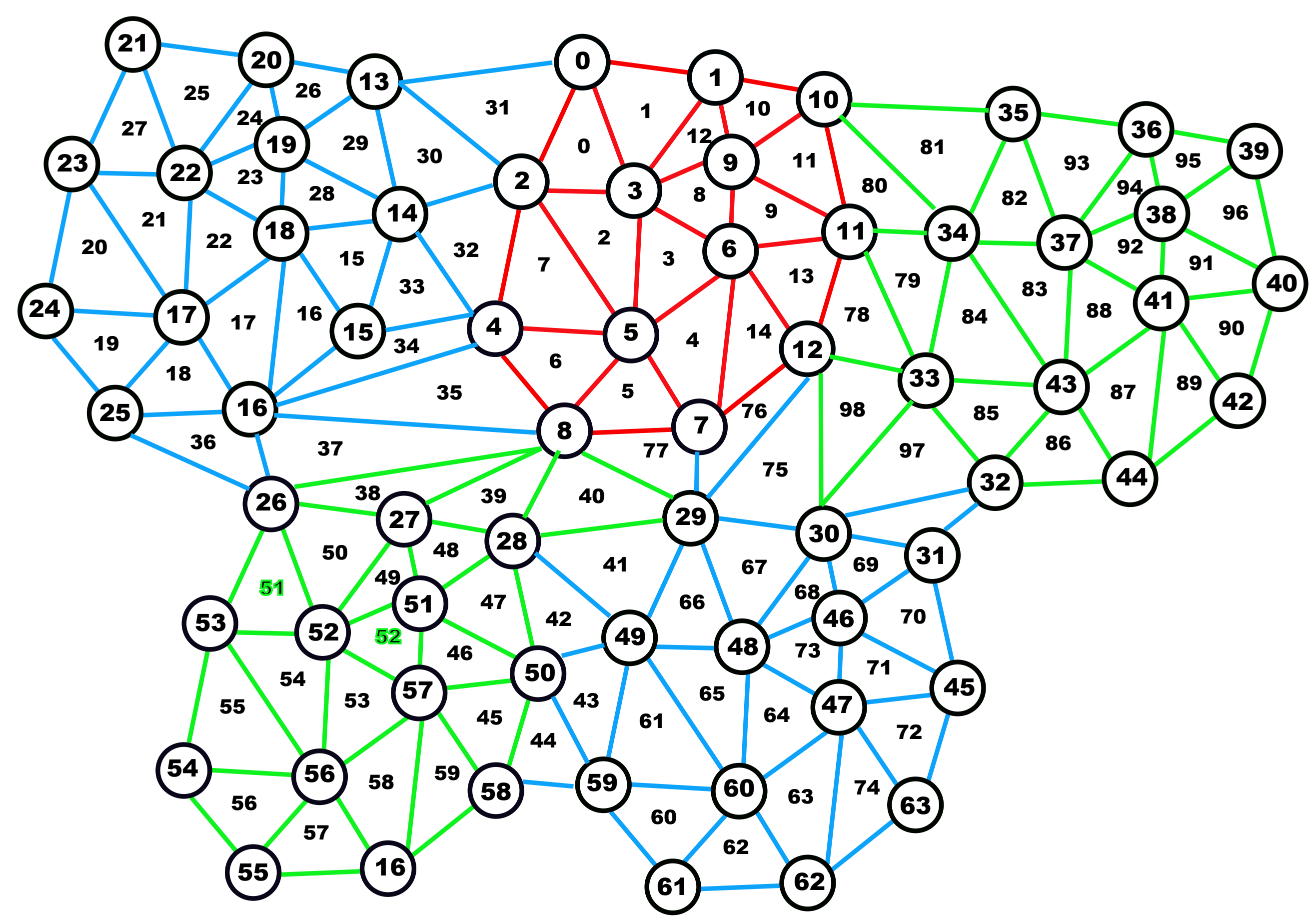
Implementation of an op_par_loop in CUDA

```
op_par_loop (incrVertices, edges,
  op_arg_dat (edgeWeight, -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;
}
```

Coloring used for avoiding race conditions in shared memory parallel execution

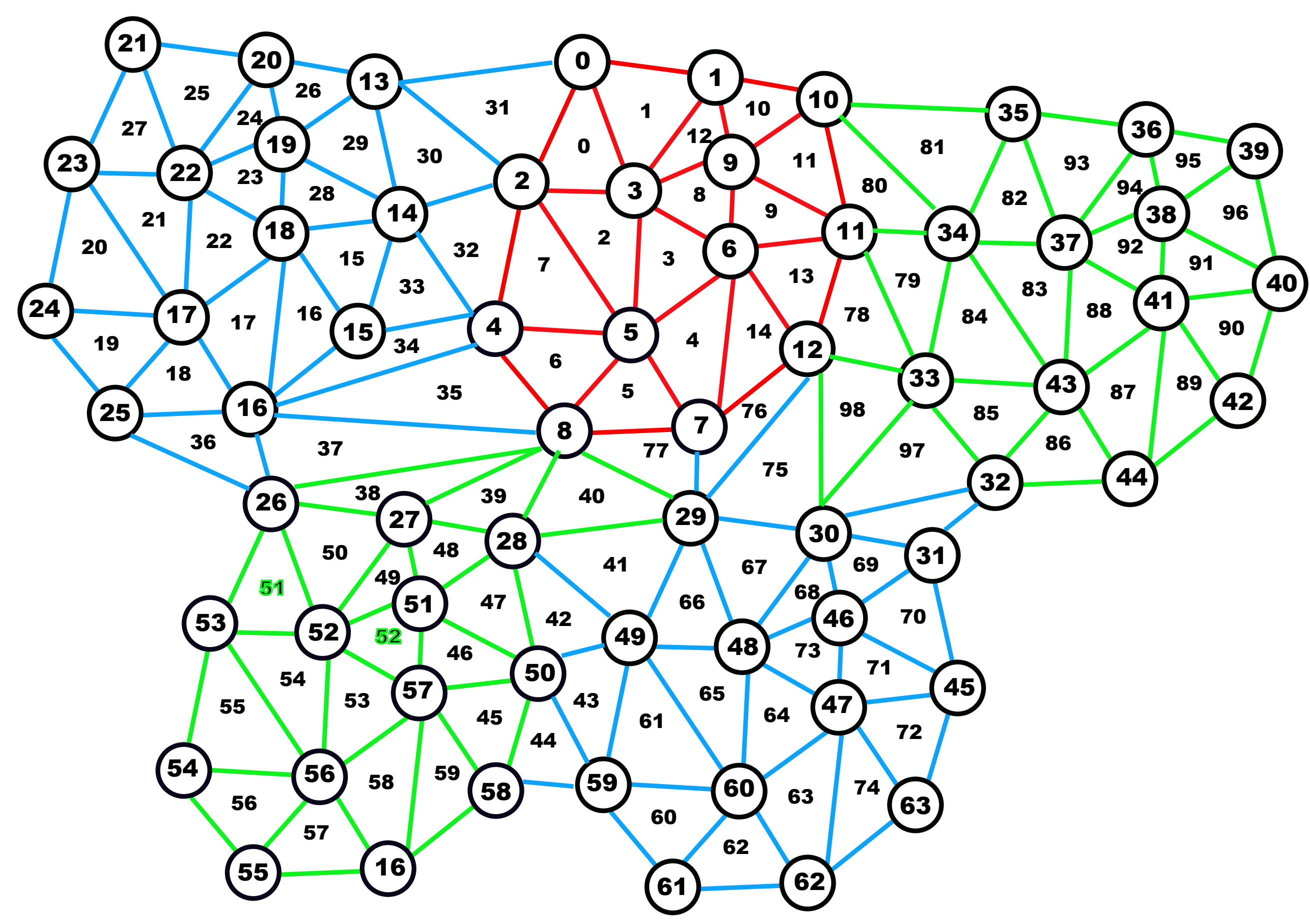


Implementation of an op_par_loop in CUDA

```
op_par_loop (incrVertices, edges,
  op_arg_dat (edgeWeight, -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;
}
```

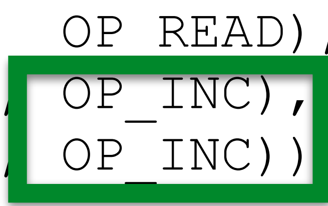
Coloring used for avoiding race conditions in shared memory parallel execution



Each partition assigned to a Thread Block and further colored

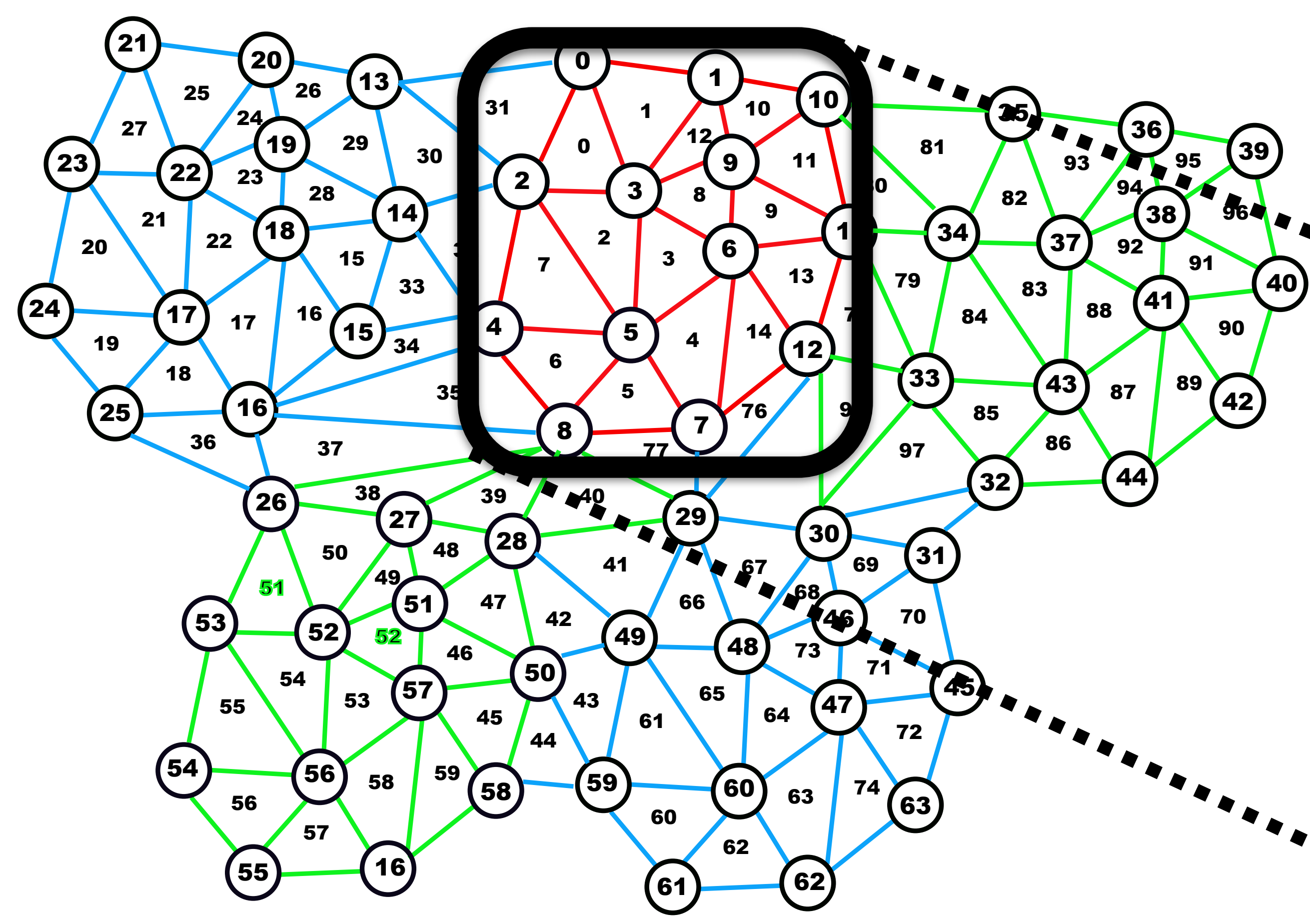
Implementation of an op_par_loop in CUDA

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

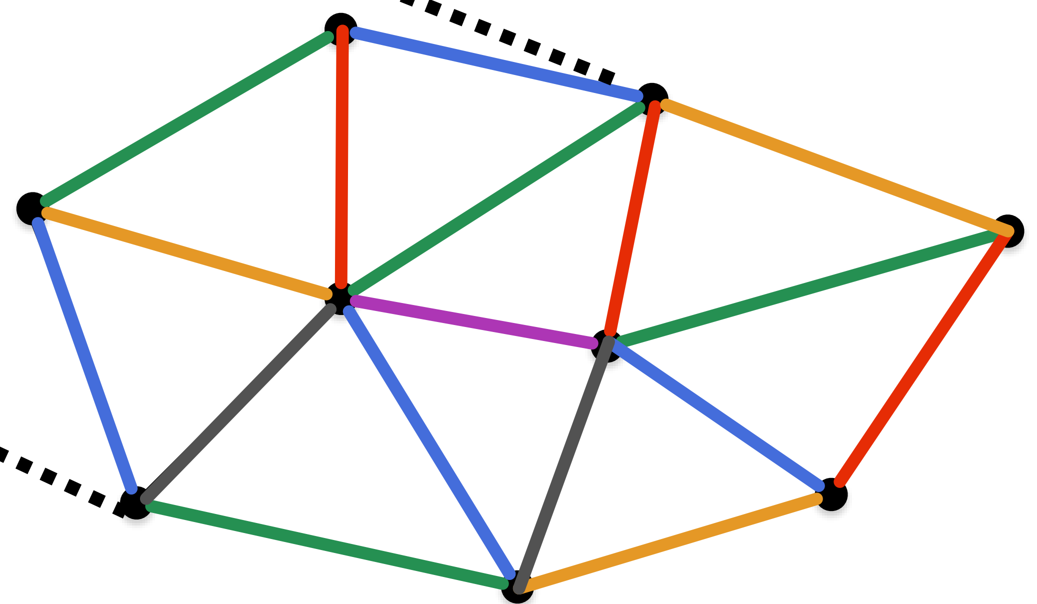


```
void incrVertices (
  double* e,
  double* v1,
  double* v2) {
  *v1 += *e;
  *v2 += *e;
}
```

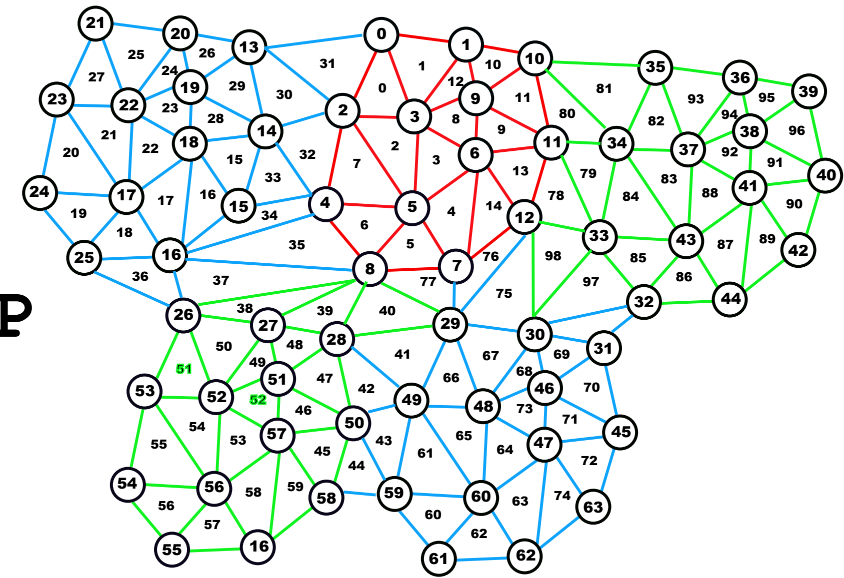
Coloring used for avoiding race conditions in shared memory parallel execution



Each partition assigned to a Thread Block and further colored



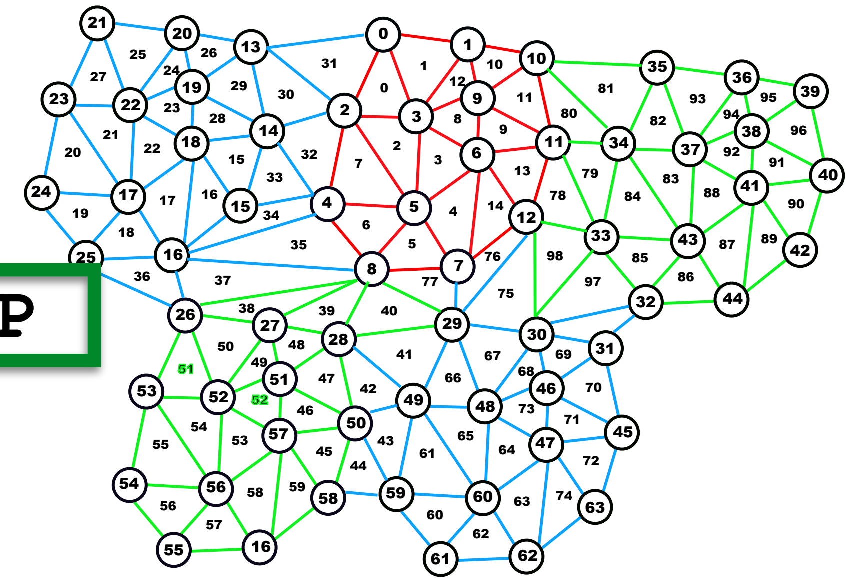
Implementation of an op_par_loop in CUDA



```
loop over colors C //seq
  loop over partitions P in C //par
    stage in all data for partition p in P
    loop over edges e in P //par
      tmp = contributions (edgeWeight)
      update ( e(v1), tmp)
      update ( e(v2), tmp)
    stage out all data for partition p in P
  end loop
end loop
```

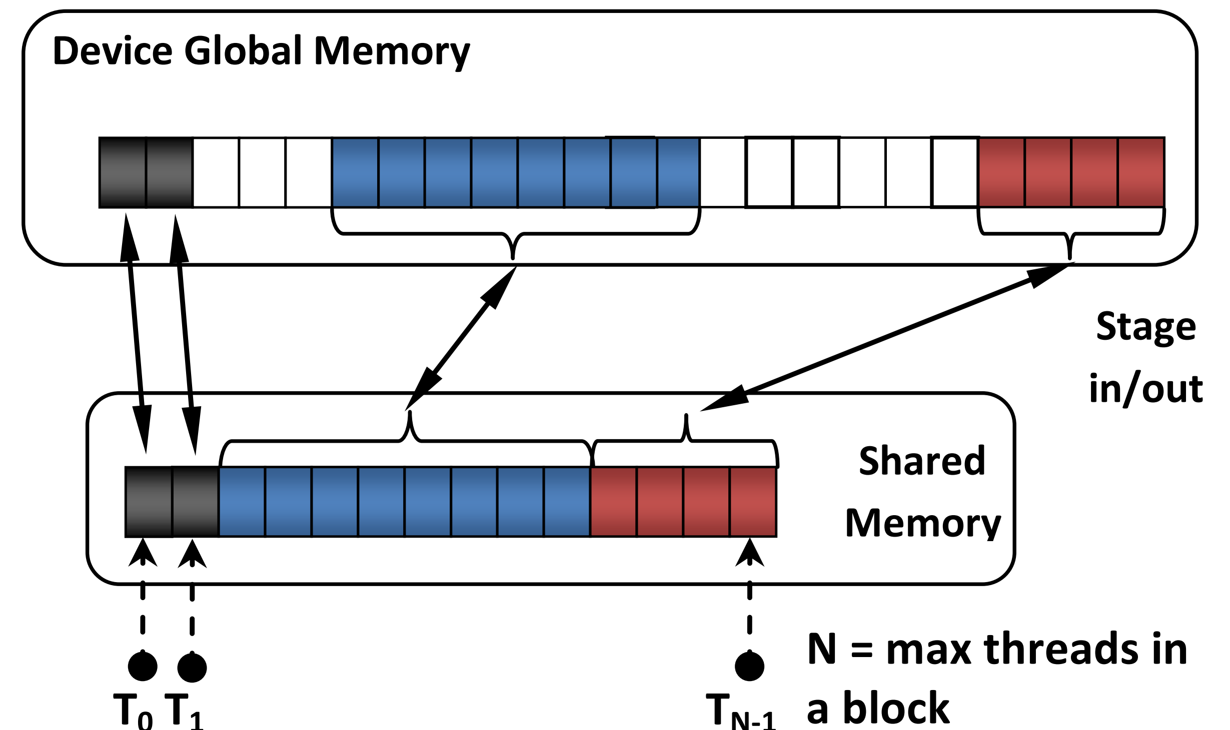
Implementation of an op_par_loop in CUDA

```
loop over colors C //seq
  loop over partitions P in C //par
    stage in all data for partition p in P
    loop over edges e in P //par
      tmp = contributions (edgeWeight)
      update ( e(v1), tmp)
      update ( e(v2), tmp)
    stage out all data for partition p in P
  end loop
end loop
```



Data is <u>usually</u> **staged** in shared memory

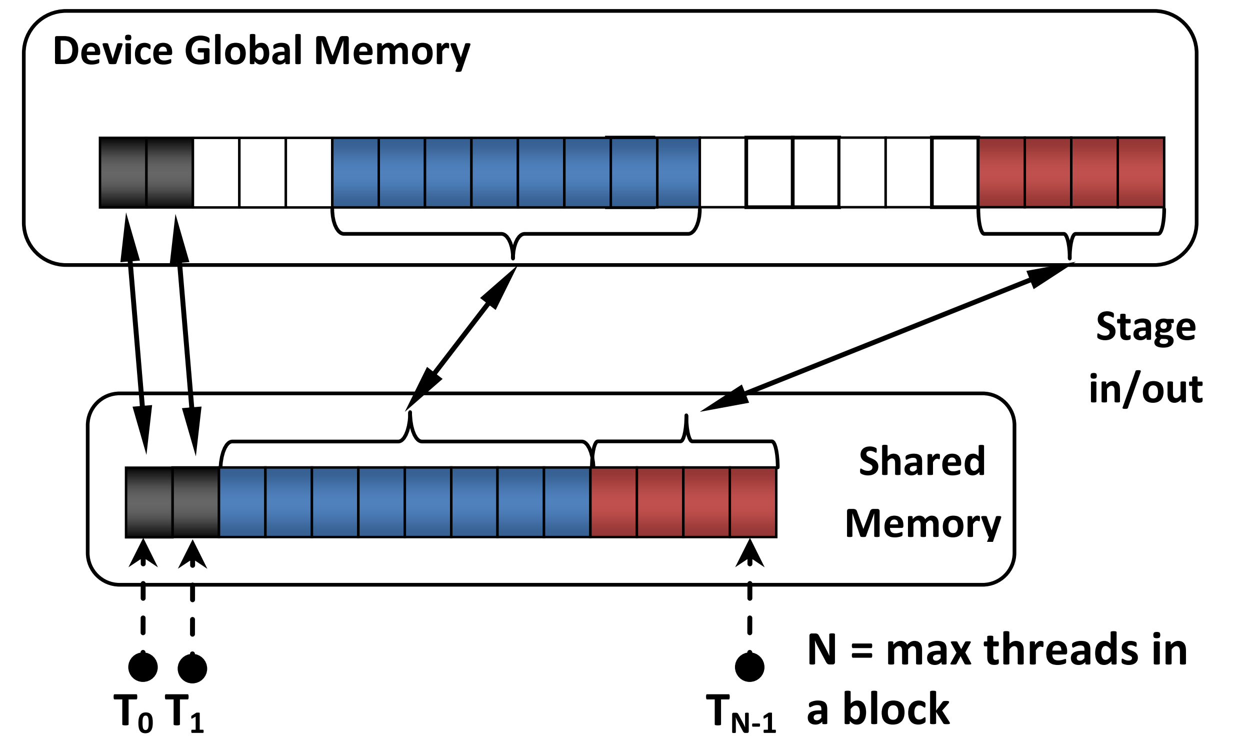
- spatial locality (SoA)
  => coalescing
- temporal locality



Challenge for effective GPU execution

Data is usually staged in shared memory

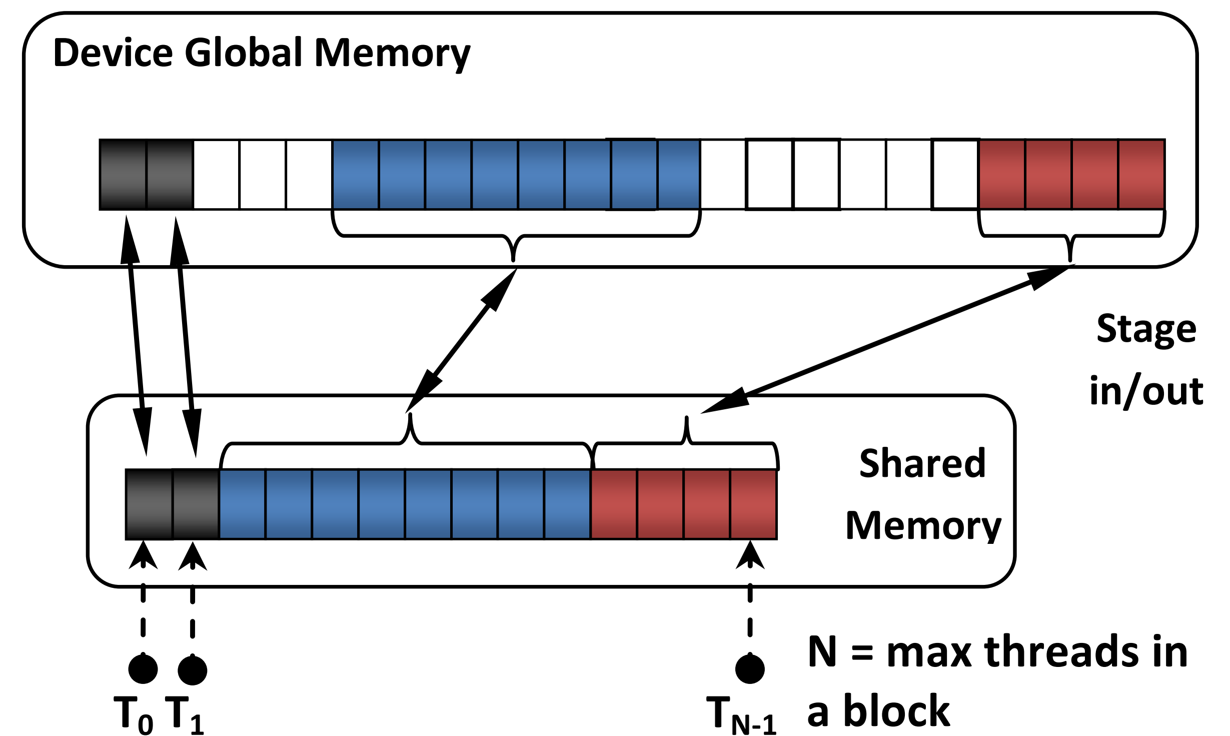
- spatial locality (SoA)
  => coalescing
- temporal locality



Challenge for effective GPU execution

Data is usually **staged** in shared memory

- spatial locality (SoA)
  => coalescing
- temporal locality

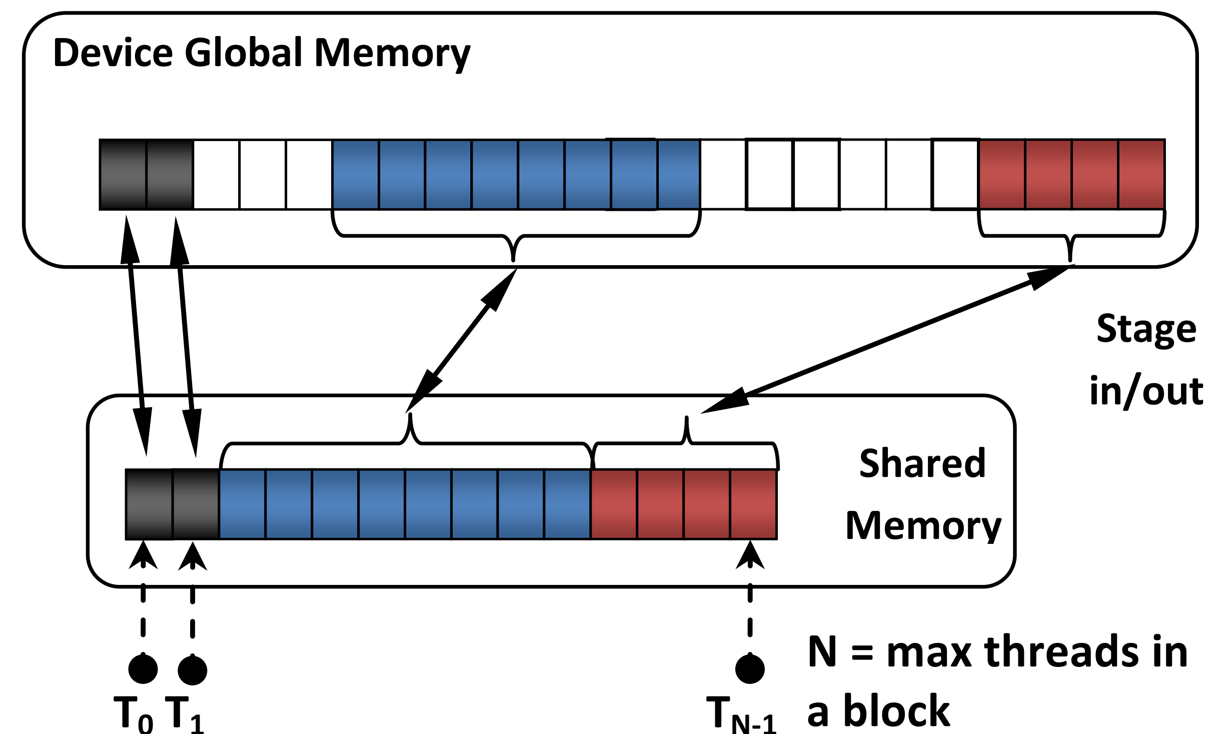


Time-expensive parallel loops in real-world computations can have a large memory footprint => Less parallelism!

Challenge for effective GPU execution

Data is usually **staged** in shared memory

- **spatial locality (SoA)** => coalescing
- **temporal locality**



Time-expensive parallel loops in real-world computations can have a large memory footprint => Less parallelism!

HYDRA example: viscous or smoothing fluxes calculation (Vflux), which accesses indirectly **808 bytes and directly 24 bytes for each iteration**. The incremented datasets are 96 bytes in size per iteration.

Loop splitting - version 1 (simplest)

```
loop over colors C //seq
  loop over partitions P in C //par
    stage in all data for contrib
    loop over edges e in P //par
      tmp = contributions (edgeWeight)
   stage out all tmp data
  end loop
end loop

loop over colors C //seq
  loop over partitions P in C //par
    stage in all data for update1 (tmp)
    loop over edges e in P //par
      update ( e(v1), tmp)
   stage out all data for partition p in P
  end loop
end loop

loop over colors C //seq
  loop over partitions P in C //par
    stage in all data for update2 (tmp)
    loop over edges e in P //par
      update ( e(v2), tmp)
   stage out all data for partition p in P
  end loop
end loop
```

Loop splitting - version 1 (simplest)

```
loop over colors C //seq
  loop over partitions P in C //par
    stage in all data for contrib
    loop over edges e in P //par
      tmp = contributions (edgeWeight)
   stage out all tmp data
  end loop
end loop

loop over colors C //seq
  loop over partitions P in C //par
    stage in all data for update1 (tmp)
    loop over edges e in P //par
      update ( e(v1), tmp)
   stage out all data for partition p in P
  end loop
end loop

loop over colors C //seq
  loop over partitions P in C //par
    stage in all data for update2 (tmp)
    loop over edges e in P //par
      update ( e(v2), tmp)
   stage out all data for partition p in P
  end loop
end loop
```

Advantages:

- Less data required by staging, so more parallelism

Loop splitting - version 1 (simplest)

```
loop over colors C //seq
  loop over partitions P in C //par
    stage in all data for contrib
    loop over edges e in P //par
      tmp = contributions (edgeWeight)
   stage out all tmp data
  end loop
end loop

loop over colors C //seq
  loop over partitions P in C //par
    stage in all data for update1 (tmp)
    loop over edges e in P //par
      update ( e(v1), tmp)
   stage out all data for partition p in P
  end loop
end loop

loop over colors C //seq
  loop over partitions P in C //par
    stage in all data for update2 (tmp)
    loop over edges e in P //par
      update ( e(v2), tmp)
   stage out all data for partition p in P
  end loop
end loop
```

Advantages:

- Less data required by staging, so more parallelism

Disadvantages:

- User intervention
- Multiple staging of same dataset (tmp)

Loop splitting - version 2

```
loop over colors C //seq
  loop over partitions P in C //par
    stage in all data for contrib
    loop over edges e in P //par
      tmp(e) = contrib (edgeData[e:0…n])
    stage in all data for update1
    loop over edges e in P //par
      update ( e(v1), tmp(e) )
    stage out all data for update1
    stage in all data for update2
    loop over edges e in P //par
      update ( e(v2), tmp(e) )
    stage out all data for update2
  end loop
end loop
```

Loop splitting - version 2

```
loop over colors C //seq
  loop over partitions P in C //par
    stage in all data for contrib
    loop over edges e in P //par
      tmp(e) = contrib (edgeData[e:0…n])
    stage in all data for update1
    loop over edges e in P //par
      update ( e(v1), tmp(e) )
    stage out all data for update1
    stage in all data for update2
    loop over edges e in P //par
      update ( e(v2), tmp(e) )
    stage out all data for update2
  end loop
end loop
```

Unary data, so kept in global memory and coalescing

Loop splitting - version 2

```
loop over colors C //seq
  loop over partitions P in C //par
    stage in all data for contrib
    loop over edges e in P //par
      tmp(e) = contrib (edgeData[e:0…n])
    stage in all data for update1
    loop over edges e in P //par
      update ( e(v1), tmp(e) )
    stage out all data for update1
    stage in all data for update2
    loop over edges e in P //par
      update ( e(v2), tmp(e) )
    stage out all data for update2
  end loop
end loop
```

Unary data, so kept in global memory and coalescing

Can overwrite previously used data

Loop splitting - version 2

```
loop over colors C //seq
  loop over partitions P in C //par
    stage in all data for contrib
    loop over edges e in P //par
      tmp(e) = contrib (edgeData[e:0…n])
    stage in all data for update1
    loop over edges e in P //par
      update ( e(v1), tmp(e) )
    stage out all data for update1
    stage in all data for update2
    loop over edges e in P //par
      update ( e(v2), tmp(e) )
    stage out all data for update2
  end loop
end loop
```

Unary data, so kept in global memory and coalescing

Can overwrite previously used data

Advantage: Does not require user intervention, just an alternative code synthesis

Loop splitting - version 3

Was a single loop "contrib"

```
loop over colors C //seq
  loop over partitions P in C //par
    stage in all data for contrib1
    loop over edges e in P //par
      tmp1(e) = contrib1 (e); tmp(e) = tmp1(e);
    stage in all data for contrib2
    loop over edges e in P //par
      tmp2(e) = contrib2 (e); tmp(e) += tmp2(e);
    ...
    stage in all data for contribN
    loop over edges e in P //par
      tmpN(e) = contribN (e); tmp(e) += tmpN(e);
    ...
    stage in all data for update1
    loop over edges e in P //par
      update ( e(v1), tmp(e) )
    stage out all data for update1
    stage in all data for update2
    loop over edges e in P //par
      update ( e(v2), tmp(e) )
    stage out all data for update2
  end loop
end loop
```

Loop splitting - version 3

Was a single loop "contrib"

```
loop over colors C //seq
  loop over partitions P in C //par
    stage in all data for contrib1
    loop over edges e in P //par
      tmp1(e) = contrib1 (e); tmp(e) = tmp1(e);
    stage in all data for contrib2
    loop over edges e in P //par
      tmp2(e) = contrib2 (e); tmp(e) += tmp2(e);
    ...
    stage in all data for contribN
    loop over edges e in P //par
      tmpN(e) = contribN (e); tmp(e) += tmpN(e);
    ...
    stage in all data for update1
    loop over edges e in P //par
      update ( e(v1), tmp(e) )
    stage out all data for update1
    stage in all data for update2
    loop over edges e in P //par
      update ( e(v2), tmp(e) )
    stage out all data for update2
  end loop
end loop
```

Relatively "simple" in HYDRA, but can also be much more complex…

How “contrib” could look like

```
loop over colors C //seq
  loop over partitions P in C //par
    stage in all data for computing A
    …
    for (int ip  = 0; ip < m; ++ip) {
      …
      for (int j  = 0; j < n; ++j) {
        for (int k  = 0; k < o; ++k) {
          A[j][k] += (det * W[ip] * B[ip][k] * B[ip][j]);
        }
      }
    }
    …
    stage out A
  end loop
end loop
```

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

Finite element - mass matrix operator

How “contrib” could look like

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) {
        A[j][k] += (((B[ip][k] * B[ip][j]) + (((((K[2] * B0[ip]
[k]) + (K[5] * B1[ip][k]) + (K[8] * B2[ip][k])) * ((K[2] *
B0[ip][j]) + (K[5] * B1[ip][j]) + (K[8] * B2[ip][j]))) +
(((K[1] * B0[ip][k]) + (K[4] * B1[ip][k]) + (K[7] * B2[ip]
[k])) * ((K[1] * B0[ip][j]) + (K[4] * B1[ip][j]) + (K[7] *
B2[ip][j]))) + (((K[0] * B0[ip][k]) + (K[3] * B1[ip][k]) +
(K[6] * B2[ip][k])) * ((K[0] * B0[ip][j]) + (K[3] * B1[ip]
[j]) + (K[6] * B2[ip][j])))) * F1 * F0)) * det * W[ip]);
      }
   }
}
...
```

Finite element -Helmholtz operator

How “contrib” could look like

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) {
        A[j][k] += ((((K[2] * BC10[0][j]) + (K[5] * BC11[0][j]) + (K[8] * BC12[0][j])) * (((K[1] * BC10[0][k]) + (K[4] * BC11[0][k]) + (K[7] * BC12[0][k])) * (((((((K[8] * F2) + (K[5] *
F1) + (K[2] * F0)) * ((K[7] * F2) + (K[4] * F1) + (K[1] * F0))) + (((K[7] * F8) + (K[4] * F7) + (K[1] * F6)) * ((K[8] * F8) + (K[5] * F7) + (K[2] * F6) + 1.0)) + (((K[8] * F5) + (K[5] * F4) + (K[2] *
F3)) * ((K[7] * F5) + (K[4] * F4) + (K[1] * F3) + 1.0))) / 2.0)) + ((((((K[8] * F2) + (K[5] * F1) + (K[2] * F0)) * ((K[7] * F2) + (K[4] * F1) + (K[1] * F0))) + (((K[7] * F8) + (K[4] * F7) + (K[1] *
F6)) * ((K[8] * F8) + (K[5] * F7) + (K[2] * F6) + 1.0)) + (((K[8] * F5) + (K[5] * F4) + (K[2] * F3)) * ((K[7] * F5) + (K[4] * F4) + (K[1] * F3) + 1.0))) / 2.0))) * F9) + (((K[6] * F5) + (K[3] * F4) +
(K[0] * F3)) * ((((((K[2] * BC20[0][k]) + (K[5] * BC21[0][k]) + (K[8] * BC22[0][k])) * ((K[6] * F8) + (K[3] * F7) + (K[0] * F6))) + (((K[0] * BC10[0][k]) + (K[3] * BC11[0][k]) + (K[6] * BC12[0][k]))
* ((K[8] * F5) + (K[5] * F4) + (K[2] * F3))) + (((K[2] * BC10[0][k]) + (K[5] * BC11[0][k]) + (K[8] * BC12[0][k])) * ((K[6] * F5) + (K[3] * F4) + (K[0] * F3))) + (((K[0] * BC00[0][k]) + (K[3] *
BC01[0][k]) + (K[6] * BC02[0][k])) * ((K[8] * F2) + (K[5] * F1) + (K[2] * F0))) + (((K[0] * BC20[0][k]) + (K[3] * BC21[0][k]) + (K[6] * BC22[0][k])) * ((K[8] * F8) + (K[5] * F7) + (K[2] * F6) + 1.0))
+ (((K[2] * BC00[0][k]) + (K[5] * BC01[0][k]) + (K[8] * BC02[0][k])) * ((K[6] * F2) + (K[3] * F1) + (K[0] * F0) + 1.0))) / 2.0)) + ((((((K[2] * BC20[0][k]) + (K[5] * BC21[0][k]) + (K[8] * BC22[0]
[k])) * ((K[6] * F8) + (K[3] * F7) + (K[0] * F6))) + (((K[0] * BC10[0][k]) + (K[3] * BC11[0][k]) + (K[6] * BC12[0][k])) * ((K[8] * F5) + (K[5] * F4) + (K[2] * F3))) + (((K[2] * BC10[0][k]) + (K[5] *
BC11[0][k]) + (K[8] * BC12[0][k])) * ((K[6] * F5) + (K[3] * F4) + (K[0] * F3))) + (((K[0] * BC00[0][k]) + (K[3] * BC01[0][k]) + (K[6] * BC02[0][k])) * ((K[8] * F2) + (K[5] * F1) + (K[2] * F0))) +
(((K[0] * BC20[0][k]) + (K[3] * BC21[0][k]) + (K[6] * BC22[0][k])) * ((K[8] * F8) + (K[5] * F7) + (K[2] * F6) + 1.0)) + (((K[2] * BC00[0][k]) + (K[5] * BC01[0][k]) + (K[8] * BC02[0][k])) * ((K[6] *
F2) + (K[3] * F1) + (K[0] * F0) + 1.0))) / 2.0))) * F9) + (((K[0] * BC10[0][k]) + (K[3] * BC11[0][k]) + (K[6] * BC12[0][k])) * ((((((K[8] * F5) + (K[5] * F4) + (K[2] * F3)) * ((K[6] * F5) + (K[3] *
F4) + (K[0] * F3))) + (((K[6] * F8) + (K[3] * F7) + (K[0] * F6)) * ((K[8] * F8) + (K[5] * F7) + (K[2] * F6) + 1.0)) + (((K[8] * F2) + (K[5] * F1) + (K[2] * F0)) * ((K[6] * F2) + (K[3] * F1) + (K[0] *
F0) + 1.0))) / 2.0)) + ((((((K[8] * F5) + (K[5] * F4) + (K[2] * F3)) * ((K[6] * F5) + (K[3] * F4) + (K[0] * F3))) + (((K[6] * F8) + (K[3] * F7) + (K[0] * F6)) * ((K[8] * F8) + (K[5] * F7) + (K[2] *
F6) + 1.0)) + (((K[8] * F2) + (K[5] * F1) + (K[2] * F0)) * ((K[6] * F2) + (K[3] * F1) + (K[0] * F0) + 1.0))) / 2.0))) * F9) + (((((((K[2] * BC20[0][k]) + (K[5] * BC21[0][k]) + (K[8] * BC22[0][k])) *
((K[7] * F8) + (K[4] * F7) + (K[1] * F6))) + (((K[1] * BC10[0][k]) + (K[4] * BC11[0][k]) + (K[7] * BC12[0][k])) * ((K[8] * F5) + (K[5] * F4) + (K[2] * F3))) + (((K[1] * BC00[0][k]) + (K[4] * BC01[0]
[k]) + (K[7] * BC02[0][k])) * ((K[8] * F2) + (K[5] * F1) + (K[2] * F0))) + (((K[2] * BC00[0][k]) + (K[5] * BC01[0][k]) + (K[8] * BC02[0][k])) * ((K[7] * F2) + (K[4] * F1) + (K[1] * F0))) + (((K[1] *
BC20[0][k]) + (K[4] * BC21[0][k]) + (K[7] * BC22[0][k])) * ((K[8] * F8) + (K[5] * F7) + (K[2] * F6) + 1.0)) + (((K[2] * BC10[0][k]) + (K[5] * BC11[0][k]) + (K[8] * BC12[0][k])) * ((K[7] * F5) + (K[4]
* F4) + (K[1] * F3) + 1.0))) / 2.0)) + (((((K[2] * BC20[0][k]) + (K[5] * BC21[0][k]) + (K[8] * BC22[0][k])) * ((K[7] * F8) + (K[4] * F7) + (K[1] * F6))) + (((K[1] * BC10[0][k]) + (K[4] * BC11[0][k])
+ (K[7] * BC12[0][k])) * ((K[8] * F5) + (K[5] * F4) + (K[2] * F3))) + (((K[1] * BC00[0][k]) + (K[4] * BC01[0][k]) + (K[7] * BC02[0][k])) * ((K[8] * F2) + (K[5] * F1) + (K[2] * F0))) + (((K[2] *
BC00[0][k]) + (K[5] * BC01[0][k]) + (K[8] * BC02[0][k])) * ((K[7] * F2) + (K[4] * F1) + (K[1] * F0))) + (((K[1] * BC20[0][k]) + (K[4] * BC21[0][k]) + (K[7] * BC22[0][k])) * ((K[8] * F8) + (K[5] * F7)
+ (K[2] * F6) + 1.0)) + (((K[2] * BC10[0][k]) + (K[5] * BC11[0][k]) + (K[8] * BC12[0][k])) * ((K[7] * F5) + (K[4] * F4) + (K[1] * F3) + 1.0))) / 2.0))) * ((K[7] * F5) + (K[4] * F4) + (K[1] * F3) +
1.0) * F9) + (((K[8] * F5) + (K[5] * F4) + (K[2] * F3)) * ((((((((K[2] * BC10[0][k]) + (K[5] * BC11[0][k]) + (K[8] * BC12[0][k])) * ((K[8] * F5) + (K[5] * F4) + (K[2] * F3))) + (((K[2] * BC10[0][k])
+ (K[5] * BC11[0][k]) + (K[8] * BC12[0][k])) * ((K[8] * F5) + (K[5] * F4) + (K[2] * F3))) + (((K[2] * BC00[0][k]) + (K[5] * BC01[0][k]) + (K[8] * BC02[0][k])) * ((K[8] * F2) + (K[5] * F1) + (K[2] *
F0))) + (((K[2] * BC00[0][k]) + (K[5] * BC01[0][k]) + (K[8] * BC02[0][k])) * ((K[8] * F2) + (K[5] * F1) + (K[2] * F0))) + (((K[2] * BC20[0][k]) + (K[5] * BC21[0][k]) + (K[8] * BC22[0][k])) * ((K[8] *
F8) + (K[5] * F7) + (K[2] * F6) + 1.0)) + (((K[2] * BC20[0][k]) + (K[5] * BC21[0][k]) + (K[8] * BC22[0][k])) * ((K[8] * F8) + (K[5] * F7) + (K[2] * F6) + 1.0))) / 2.0)) + ((((((K[2] * BC10[0][k]) +
(K[5] * BC11[0][k]) + (K[8] * BC12[0][k])) * ((K[8] * F5) + (K[5] * F4) + (K[2] * F3))) + (((K[2] * BC10[0][k]) + (K[5] * BC11[0][k]) + (K[8] * BC12[0][k])) * ((K[8] * F5) + (K[5] * F4) + (K[2] *
F3))) + (((K[2] * BC00[0][k]) + (K[5] * BC01[0][k]) + (K[8] * BC02[0][k])) * ((K[8] * F2) + (K[5] * F1) + (K[2] * F0))) + (((K[2] * BC00[0][k]) + (K[5] * BC01[0][k]) + (K[8] * BC02[0][k])) * ((K[8] *
F2) + (K[5] * F1) + (K[2] * F0))) + (((K[2] * BC20[0][k]) + (K[5] * BC21[0][k]) + (K[8] * BC22[0][k])) * ((K[8] * F8) + (K[5] * F7) + (K[2] * F6) + 1.0)) + (((K[2] * BC20[0][k]) + (K[5] * BC21[0][k])
+ (K[8] * BC22[0][k])) * ((K[8] * F8) + (K[5] * F7) + (K[2] * F6) + 1.0))) / 2.0))) * F9) + (((F10) / 2.0) * ((1.0)) * ((((((K[2] * BC10[0][k]) + (K[5] * BC11[0][k]) + (K[8] * BC12[0][k])) * ((K[8] *
F5) + (K[5] * F4) + (K[2] * F3))) + (((K[2] * BC10[0][k]) + (K[5] * BC11[0][k]) + (K[8] * BC12[0][k])) * ((K[8] * F5) + (K[5] * F4) + (K[2] * F3))) + (((K[2] * BC00[0][k]) + (K[5] * BC01[0][k]) +
(K[8] * BC02[0][k])) * ((K[8] * F2) + (K[5] * F1) + (K[2] * F0))) + (((K[2] * BC00[0][k]) + (K[5] * BC01[0][k]) + (K[8] * BC02[0][k])) * ((K[8] * F2) + (K[5] * F1) + (K[2] * F0))) + (((K[2] * BC20[0]
[k]) + (K[5] * BC21[0][k]) + (K[8] * BC22[0][k])) * ((K[8] * F8) + (K[5] * F7) + (K[2] * F6) + 1.0)) + (((K[2] * BC20[0][k]) + (K[5] * BC21[0][k]) + (K[8] * BC22[0][k])) * ((K[8] * F8) + (K[5] * F7)
+ (K[2] * F6) + 1.0))) / 2.0) + ((((K[1] * BC20[0][k]) + (K[4] * BC21[0][k]) + (K[7] * BC22[0][k])) * ((K[7] * F8) + (K[4] * F7) + (K[1] * F6))) + (((K[1] * BC20[0][k]) + (K[4] * BC21[0][k]) + (K[7]
* BC22[0][k])) * ((K[7] * F8) + (K[4] * F7) + (K[1] * F6))) + (((K[1] * BC00[0][k]) + (K[4] * BC01[0][k]) + (K[7] * BC02[0][k])) * ((K[7] * F2) + (K[4] * F1) + (K[1] * F0))) + (((K[1] * BC00[0][k]) +
(K[4] * BC01[0][k]) + (K[7] * BC02[0][k])) * ((K[7] * F2) + (K[4] * F1) + (K[1] * F0))) + (((K[1] * BC10[0][k]) + (K[4] * BC11[0][k]) + (K[7] * BC12[0][k])) * ((K[7] * F5) + (K[4] * F4) + (K[1] * F3)
+ 1.0)) + (((K[1] * BC10[0][k]) + (K[4] * BC11[0][k]) + (K[7] * BC12[0][k])) * ((K[7] * F5) + (K[4] * F4) + (K[1] * F3) + 1.0))) / 2.0) + (((((K[0] * BC20[0][k]) + (K[3] * BC21[0][k]) + (K[6] *
BC22[0][k])) * ((K[6] * F8) + (K[3] * F7) + (K[0] * F6))) + (((K[0] * BC20[0][k]) + (K[3] * BC21[0][k]) + (K[6] * BC22[0][k])) * ((K[6] * F8) + (K[3] * F7) + (K[0] * F6))) + (((K[0] * BC10[0][k]) +
(K[3] * BC11[0][k]) + (K[6] * BC12[0][k])) * ((K[6] * F5) + (K[3] * F4) + (K[0] * F3))) + (((K[0] * BC10[0][k]) + (K[3] * BC11[0][k]) + (K[6] * BC12[0][k])) * ((K[6] * F5) ….
    }
  }
}
…
```

Finite element - Hyperelasticity operator

How “contrib” could look like

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) {
        A[j][k] += ((((K[2] * BC10[0][j]) + (K[5] * BC11[0][j]) + (K[8] * BC12[0][j])) * (((K[1] * BC10[0][k]) + (K[4] * BC11[0][k]) + (K[7] * BC12[0][k])) * ((((((K[8] * F2) + (K[5] *
F1) + (K[2] * F0)) * ((K[7] * F2) + (K[4] * F1) + (K[1] * F0))) + (((K[7] * F8) + (K[4] * F7) + (K[1] * F6)) * ((K[8] * F8) + (K[5] * F7) + (K[2] * F6) + 1.0)) + (((K[8] * F5) + (K[5] * F4) + (K[2] *
F3)) * ((K[7] * F5) + (K[4] * F4) + (K[1] * F3) + 1.0))) / 2.0)) + (((((K[8] * F2) + (K[5] * F1) + (K[2] * F0)) * ((K[7] * F2) + (K[4] * F1) + (K[1] * F0))) + (((K[7] * F8) + (K[4] * F7) + (K[1] *
F6)) * ((K[8] * F8) + (K[5] * F7) + (K[2] * F6) + 1.0)) + (((K[8] * F5) + (K[5] * F4) + (K[2] * F3)) * ((K[7] * F5) + (K[4] * F4) + (K[1] * F3) + 1.0))) / 2.0))) * F9) + (((K[6] * F5) + (K[3] * F4) +
(K[0] * F3)) * ((((((K[2] * BC20[0][k]) + (K[5] * BC21[0][k]) + (K[8] * BC22[0][k])) * ((K[6] * F8) + (K[3] * F7) + (K[0] * F6))) + (((K[0] * BC10[0][k]) + (K[3] * BC11[0][k]) + (K[6] * BC12[0][k]))
* ((K[8] * F5) + (K[5] * F4) + (K[2] * F3))) + (((K[2] * BC10[0][k]) + (K[5] * BC11[0][k]) + (K[8] * BC12[0][k])) * ((K[6] * F5) + (K[3] * F4) + (K[0] * F3))) + (((K[0] * BC00[0][k]) + (K[3] *
BC01[0][k]) + (K[6] * BC02[0][k])) * ((K[8] * F2) + (K[5] * F1) + (K[2] * F0))) + (((K[0] * BC20[0][k]) + (K[3] * BC21[0][k]) + (K[6] * BC22[0][k])) * ((K[8] * F8) + (K[5] * F7) + (K[2] * F6) + 1.0))
+ (((K[2] * BC00[0][k]) + (K[5] * BC01[0][k]) + (K[8] * BC02[0][k])) * ((K[6] * F2) + (K[3] * F1) + (K[0] * F0) + 1.0))) / 2.0)) + ((((((K[2] * BC20[0][k]) + (K[5] * BC21[0][k]) + (K[8] * BC22[0]
[k])) * ((K[6] * F8) + (K[3] * F7) + (K[0] * F6))) + (((K[0] * BC10[0][k]) + (K[3] * BC11[0][k]) + (K[6] * BC12[0][k])) * ((K[8] * F5) + (K[5] * F4) + (K[2] * F3))) + (((K[2] * BC10[0][k]) + (K[5] *
BC11[0][k]) + (K[8] * BC12[0][k])) * ((K[6] * F5) + (K[3] * F4) + (K[0] * F3))) + (((K[0] * BC00[0][k]) + (K[3] * BC01[0][k]) + (K[6] * BC02[0][k])) * ((K[8] * F2) + (K[5] * F1) + (K[2] * F0))) +
(((K[0] * BC20[0][k]) + (K[3] * BC21[0][k]) + (K[6] * BC22[0][k])) * ((K[8] * F8) + (K[5] * F7) + (K[2] * F6) + 1.0)) + (((K[2] * BC00[0][k]) + (K[5] * BC01[0][k]) + (K[8] * BC02[0][k])) * ((K[6] *
F2) + (K[3] * F1) + (K[0] * F0) + 1.0))) / 2.0))) * F9) + (((K[0] * BC10[0][k]) + (K[3] * BC11[0][k]) + (K[6] * BC12[0][k])) * ((((((K[8] * F5) + (K[5] * F4) + (K[2] * F3)) * ((K[6] * F5) + (K[3] *
F4) + (K[0] * F3))) + (((K[6] * F8) + (K[3] * F7) + (K[0] * F6)) * ((K[8] * F8) + (K[5] * F7) + (K[2] * F6) + 1.0)) + (((K[8] * F2) + (K[5] * F1) + (K[2] * F0)) * ((K[6] * F2) + (K[3] * F1) + (K[0] *
F0) + 1.0))) / 2.0)) + (((((K[8] * F5) + (K[5] * F4) + (K[2] * F3)) * ((K[6] * F5) + (K[3] * F4) + (K[0] * F3))) + (((K[6] * F8) + (K[3] * F7) + (K[0] * F6)) * ((K[8] * F8) + (K[5] * F7) + (K[2] *
F6) + 1.0)) + (((K[8] * F2) + (K[5] * F1) + (K[2] * F0)) * ((K[6] * F2) + (K[3] * F1) + (K[0] * F0) + 1.0))) / 2.0))) * F9) + (((((((K[2] * BC20[0][k]) + (K[5] * BC21[0][k]) + (K[8] * BC22[0][k])) *
((K[7] * F8) + (K[4] * F7) + (K[1] * F6))) + (((K[1] * BC10[0][k]) + (K[4] * BC11[0][k]) + (K[7] * BC12[0][k])) * ((K[8] * F5) + (K[5] * F4) + (K[2] * F3))) + (((K[1] * BC00[0][k]) + (K[4] * BC01[0]
[k]) + (K[7] * BC02[0][k])) * ((K[8] * F2) + (K[5] * F1) + (K[2] * F0))) + (((K[2] * BC00[0][k]) + (K[5] * BC01[0][k]) + (K[8] * BC02[0][k])) * ((K[7] * F2) + (K[4] * F1) + (K[1] * F0))) + (((K[1] *
BC20[0][k]) + (K[4] * BC21[0][k]) + (K[7] * BC22[0][k])) * ((K[8] * F8) + (K[5] * F7) + (K[2] * F6) + 1.0)) + (((K[2] * BC10[0][k]) + (K[5] * BC11[0][k]) + (K[8] * BC12[0][k])) * ((K[7] * F5) + (K[4]
* F4) + (K[1] * F3) + 1.0))) / 2.0)) + (((((K[2] * BC20[0][k]) + (K[5] * BC21[0][k]) + (K[8] *              ) + (K[1] * F6))) + (((K[1] * BC10[0][k]) + (K[4] * BC11[0][k])
+ (K[7] * BC12[0][k])) * ((K[8] * F5) + (K[5] * F4) + (K[2] * F3))) + (((K[1] * BC00[0][k])                 k])) * ((K[8] * F2) + (K[5] * F1) + (K[2] * F0))) + (((K[2] *
BC00[0][k]) + (K[5] * BC01[0][k]) + (K[8] * BC02[0][k])) * ((K[7] * F2) + (K[4] * F1) + (K[1] *             BC21[0][k]) + (K[7] * BC22[0][k])) * ((K[8] * F8) + (K[5] * F7)
+ (K[2] * F6) + 1.0)) + (((K[2] * BC10[0][k]) + (K[5] * BC11[0][k]) + (K[8] * BC12[0][k])) * (             + 1.0))) / 2.0))) * ((K[7] * F5) + (K[4] * F4) + (K[1] * F3) +
1.0) * F9) + (((K[8] * F5) + (K[5] * F4) + (K[2] * F3)) * ((((((((K[2] * BC10[0][k]) + (K[5] *              [8] * F5) + (K[5] * F4) + (K[2] * F3))) + (((K[2] * BC10[0][k])
+ (K[5] * BC11[0][k]) + (K[8] * BC12[0][k])) * ((K[8] * F5) + (K[5] * F4) + (K[2] * F3))) + ((              + (K[8] * BC02[0][k])) * ((K[8] * F2) + (K[5] * F1) + (K[2] *
F0))) + (((K[2] * BC00[0][k]) + (K[5] * BC01[0][k]) + (K[8] * BC02[0][k])) * ((K[8] * F2) + (K[             [0][k]) + (K[5] * BC21[0][k]) + (K[8] * BC22[0][k])) * ((K[8] *
F8) + (K[5] * F7) + (K[2] * F6) + 1.0)) + (((K[2] * BC20[0][k]) + (K[5] * BC21[0][k]) + (K[8]               7) + (K[2] * F6) + 1.0))) / 2.0)) + ((((((K[2] * BC10[0][k]) +
(K[5] * BC11[0][k]) + (K[8] * BC12[0][k])) * ((K[8] * F5) + (K[5] * F4) + (K[2] * F3))) + (((               + (K[8] * BC12[0][k])) * ((K[8] * F5) + (K[5] * F4) + (K[2] *
F3))) + (((K[2] * BC00[0][k]) + (K[5] * BC01[0][k]) + (K[8] * BC02[0][k])) * ((K[8] * F2) + (K[             [0][k]) + (K[5] * BC01[0][k]) + (K[8] * BC02[0][k])) * ((K[8] *
F2) + (K[5] * F1) + (K[2] * F0))) + (((K[2] * BC20[0][k]) + (K[5] * BC21[0][k]) + (K[8] * BC22[             [2] * F6) + 1.0)) + (((K[2] * BC20[0][k]) + (K[5] * BC21[0][k])
+ (K[8] * BC22[0][k])) * ((K[8] * F8) + (K[5] * F7) + (K[2] * F6) + 1.0))) / 2.0))) * F9) + (((             * BC11[0][k]) + (K[8] * BC12[0][k])) * ((K[8] *
F5) + (K[5] * F4) + (K[2] * F3))) + (((K[2] * BC10[0][k]) + (K[5] * BC11[0][k]) + (K[8] * BC12[0][k])) * ((K[8] * F5) + (K[5] * F4) +          + (((K[2] * BC00[0][k]) + (K[5] * BC01[0][k]) +
(K[8] * BC02[0][k])) * ((K[8] * F2) + (K[5] * F1) + (K[2] * F0))) + (((K[2] * BC00[0][k]) + (K[5] * BC01[0][k]) + (K[8] * BC02[0][k])) *          (K[5] * F1) + (K[2] * F0))) + (((K[2] * BC20[0]
[k]) + (K[5] * BC21[0][k]) + (K[8] * BC22[0][k])) * ((K[8] * F8) + (K[5] * F7) + (K[2] * F6) + 1.0)) + (((K[2] * BC20[0][k]) + (K[5] * BC          [0][k])) * ((K[8] * F8) + (K[5] * F7)
+ (K[2] * F6) + 1.0))) / 2.0) + ((((K[1] * BC20[0][k]) + (K[4] * BC21[0][k]) + (K[7] * BC22[0][k])) * ((K[7] * F8) + (K[4] * F7) + (K[1]          [0][k]) + (K[4] * BC21[0][k]) + (K[7]
* BC22[0][k])) * ((K[7] * F8) + (K[4] * F7) + (K[1] * F6))) + (((K[1] * BC00[0][k]) + (K[4] * BC01[0][k]) + (K[7] * BC02[0][k])) * ((K[7] * F2) + (K[4]          [1] * F0))) + (((K[1] * BC00[0][k]) +
(K[4] * BC01[0][k]) + (K[7] * BC02[0][k])) * ((K[7] * F2) + (K[4] * F1) + (K[1] * F0))) + (((K[1] * BC10[0][k]) + (K[4] * BC11[0][k]) + (K[7] * BC12[0][          [7] * F5) + (K[4] * F4) + (K[1] * F3)
+ 1.0)) + (((K[1] * BC10[0][k]) + (K[4] * BC11[0][k]) + (K[7] * BC12[0][k])) * ((K[7] * F5) + (K[4] * F4) + (K[1] * F3) + 1.0))) / 2.0) + (((((K[0] * BC20[0][       + (K[3] * BC21[0][k]) + (K[6] *
BC22[0][k])) * ((K[6] * F8) + (K[3] * F7) + (K[0] * F6))) + (((K[0] * BC20[0][k]) + (K[3] * BC21[0][k]) + (K[6] * BC22[0][k])) * ((K[6] * F8) + (K[3] * F7) + (K[0       F6))) + (((K[0] * BC10[0][k]) +
(K[3] * BC11[0][k]) + (K[6] * BC12[0][k])) * ((K[6] * F5) + (K[3] * F4) + (K[0] * F3))) + (((K[0] * BC10[0][k]) + (K[3] * BC11[0][k]) + (K[6] * BC12[0][k])) * ((K[6] * F5    ....
    }
  }
}
…
```

• • •

Finite element - Hyperelasticity operator

How “contrib” could look like

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) {
```

Work in progress

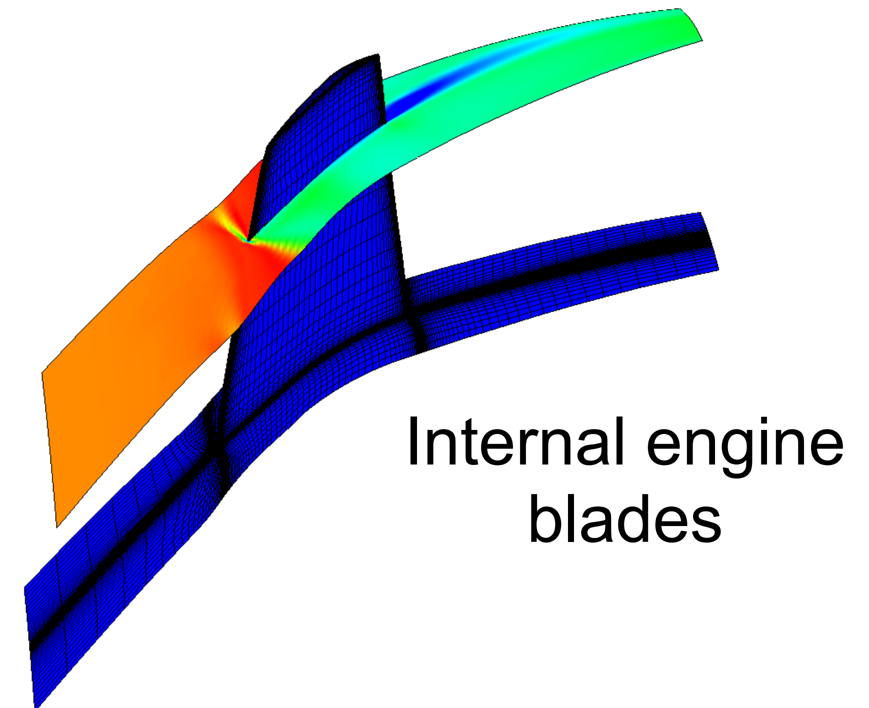
```
[j])) * ((((K[1] * BC10[0][k]) + (K[4] * BC11[0][k]) + (K[7] * BC12[0][k])) * (((((((K[8] * F2) + (K[5] *
+ (K[1] * F6)) * ((K[8] * F8) + (K[5] * F7) + (K[2] * F6) + 1.0)) + (((K[8] * F5) + (K[5] * F4) + (K[2] *
F1) + (K[2] * F0)) * ((K[7] * F2) + (K[4] * F1) + (K[1] * F0))) + (((K[7] * F8) + (K[4] * F7) + (K[1] *
* F3)) * ((K[7] * F5) + (K[4] * F4) + (K[1] * F3) + 1.0))) / 2.0))) * F9) + (((K[6] * F5) + (K[3] * F4) +
* F8) + (K[3] * F7) + (K[0] * F6))) + (((K[0] * BC10[0][k]) + (K[3] * BC11[0][k]) + (K[6] * BC12[0][k]))
+ (K[8] * BC12[0][k])) * ((K[6] * F5) + (K[3] * F4) + (K[0] * F3))) + (((K[0] * BC00[0][k]) + (K[3] *
20[0][k]) + (K[3] * BC21[0][k]) + (K[6] * BC22[0][k])) * ((K[8] * F8) + (K[5] * F7) + (K[2] * F6) + 1.0))
* F1) + (K[0] * F0) + 1.0))) / 2.0)) + ((((((K[2] * BC20[0][k]) + (K[5] * BC21[0][k]) + (K[8] * BC22[0]
k]) + (K[6] * BC12[0][k])) * ((K[8] * F5) + (K[5] * F4) + (K[2] * F3))) + (((K[2] * BC10[0][k]) + (K[5] *
BC00[0][k]) + (K[3] * BC01[0][k]) + (K[6] * BC02[0][k])) * ((K[8] * F2) + (K[5] * F1) + (K[2] * F0))) +
7) + (K[2] * F6) + 1.0)) + (((K[2] * BC00[0][k]) + (K[5] * BC01[0][k]) + (K[8] * BC02[0][k])) * ((K[6] *
1[0][k]) + (K[6] * BC12[0][k])) * ((((((K[8] * F5) + (K[5] * F4) + (K[2] * F3)) * ((K[6] * F5) + (K[3] *
+ (K[2] * F6) + 1.0)) + (((K[8] * F2) + (K[5] * F1) + (K[2] * F0)) * ((K[6] * F2) + (K[3] * F1) + (K[0] *
F4) + (K[0] * F3))) + (((K[6] * F8) + (K[3] * F7) + (K[0] * F6)) * ((K[8] * F8) + (K[5] * F7) + (K[2] *
* F0) + 1.0))) / 2.0))) * F9) + (((((((K[2] * BC20[0][k]) + (K[5] * BC21[0][k]) + (K[8] * BC22[0][k])) *
[7] * BC12[0][k])) * ((K[8] * F5) + (K[5] * F4) + (K[2] * F3))) + (((K[1] * BC00[0][k]) + (K[4] * BC01[0]
k]) + (K[5] * BC01[0][k]) + (K[8] * BC02[0][k])) * ((K[7] * F2) + (K[4] * F1) + (K[1] * F0))) + (((K[1] *
* F6) + 1.0)) + (((K[2] * BC10[0][k]) + (K[5] * BC11[0][k]) + (K[8] * BC12[0][k])) * ((K[7] * F5) + (K[4]
) + (K[1] * F6))) + (((K[1] * BC10[0][k]) + (K[4] * BC11[0][k])
k])) * ((K[8] * F2) + (K[5] * F1) + (K[2] * F0))) + (((K[2] *
BC21[0][k]) + (K[7] * BC22[0][k])) * ((K[8] * F8) + (K[5] * F7)
+ 1.0))) / 2.0))) * ((K[7] * F5) + (K[4] * F4) + (K[1] * F3) +
[8] * F5) + (K[5] * F4) + (K[2] * F3))) + (((K[2] * BC10[0][k])
+ (K[8] * BC02[0][k])) * ((K[8] * F2) + (K[5] * F1) + (K[2] *
F0))) + (((K[2] * BC00[0][k]) + (K[5] * BC01[0][k]) + (K[8] * BC02[0][k])) * ((K[8] * F2) + (K[ ... [0][k]) + (K[5] * BC21[0][k]) + (K[8] * BC22[0][k])) * ((K[8] *
F8) + (K[5] * F7) + (K[2] * F6) + 1.0)) + (((K[2] * BC20[0][k]) + (K[5] * BC21[0][k]) + (K[8] ... 7) + (K[2] * F6) + 1.0))) / 2.0)) + (((((K[2] * BC10[0][k]) +
(K[5] * BC11[0][k]) + (K[8] * BC12[0][k])) * ((K[8] * F5) + (K[5] * F4) + (K[2] * F3))) + ((( ... + (K[8] * BC12[0][k])) * ((K[8] * F5) + (K[5] * F4) + (K[2] *
F3))) + (((K[2] * BC00[0][k]) + (K[5] * BC01[0][k]) + (K[8] * BC02[0][k])) * ((K[8] * F2) + (K[ ... [0][k]) + (K[5] * BC01[0][k]) + (K[8] * BC02[0][k])) * ((K[8] *
F2) + (K[5] * F1) + (K[2] * F0))) + (((K[2] * BC20[0][k]) + (K[5] * BC21[0][k]) + (K[8] * BC22[ ... [2] * F6) + 1.0)) + (((K[2] * BC20[0][k]) + (K[5] * BC21[0][k])
+ (K[8] * BC22[0][k])) * ((K[8] * F8) + (K[5] * F7) + (K[2] * F6) + 1.0))) / 2.0))) * F9) + ((( ... * BC11[0][k]) + (K[8] * BC12[0][k])) * ((K[8] *
F5) + (K[5] * F4) + (K[2] * F3))) + (((K[2] * BC10[0][k]) + (K[5] * BC11[0][k]) + (K[8] * BC12[0][k])) * ((K[8] * F5) + (K[5] * F4) + ... + (((K[2] * BC00[0][k]) + (K[5] * BC01[0][k]) +
(K[8] * BC02[0][k])) * ((K[8] * F2) + (K[5] * F1) + (K[2] * F0))) + (((K[2] * BC00[0][k]) + (K[5] * BC01[0][k]) + (K[8] * BC02[0][k])) * ... + (K[5] * F1) + (K[2] * F0))) + (((K[2] * BC20[0]
[k]) + (K[5] * BC21[0][k]) + (K[8] * BC22[0][k])) * ((K[8] * F8) + (K[5] * F7) + (K[2] * F6) + 1.0)) + (((K[2] * BC20[0][k]) + (K[5] * BC ... [0][k])) * ((K[8] * F8) + (K[5] * F7)
+ (K[2] * F6) + 1.0))) / 2.0) + (((((K[1] * BC20[0][k]) + (K[4] * BC21[0][k]) + (K[7] * BC22[0][k])) * ((K[7] * F8) + (K[4] * F7) + (K[1] ... [0][k]) + (K[4] * BC21[0][k]) + (K[7]
* BC22[0][k])) * ((K[7] * F8) + (K[4] * F7) + (K[1] * F6))) + (((K[1] * BC00[0][k]) + (K[4] * BC01[0][k]) + (K[7] * BC02[0][k])) * ((K[7] * F2) + (K[4] ... K[1] * F0))) + (((K[1] * BC00[0][k]) +
(K[4] * BC01[0][k]) + (K[7] * BC02[0][k])) * ((K[7] * F2) + (K[4] * F1) + (K[1] * F0))) + (((K[1] * BC10[0][k]) + (K[4] * BC11[0][k]) + (K[7] * BC12[0][ ... K[7] * F5) + (K[4] * F4) + (K[1] * F3)
+ 1.0)) + (((K[1] * BC10[0][k]) + (K[4] * BC11[0][k]) + (K[7] * BC12[0][k])) * ((K[7] * F5) + (K[4] * F4) + (K[1] * F3) + 1.0))) / 2.0) + (((((K[0] * BC20[0] ... (K[3] * BC21[0][k]) + (K[6] *
BC22[0][k])) * ((K[6] * F8) + (K[3] * F7) + (K[0] * F6))) + (((K[0] * BC20[0][k]) + (K[3] * BC21[0][k]) + (K[6] * BC22[0][k])) * ((K[6] * F8) + (K[3] * F7) + (K[ ... F6))) + (((K[0] * BC10[0][k]) +
(K[3] * BC11[0][k]) + (K[6] * BC12[0][k])) * ((K[6] * F5) + (K[3] * F4) + (K[0] * F3))) + (((K[0] * BC10[0][k]) + (K[3] * BC11[0][k]) + (K[6] * BC12[0][k])) * ((K[6] * F5....
    }
  }
}
...
```

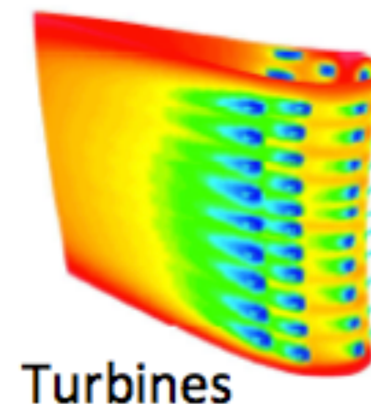
...

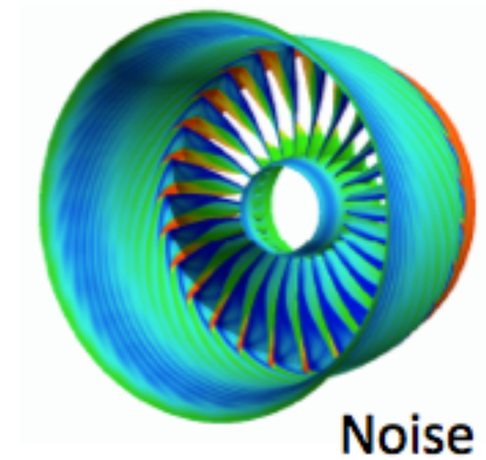
Finite element - Hyperelasticity operator

Performance results - HYDRA

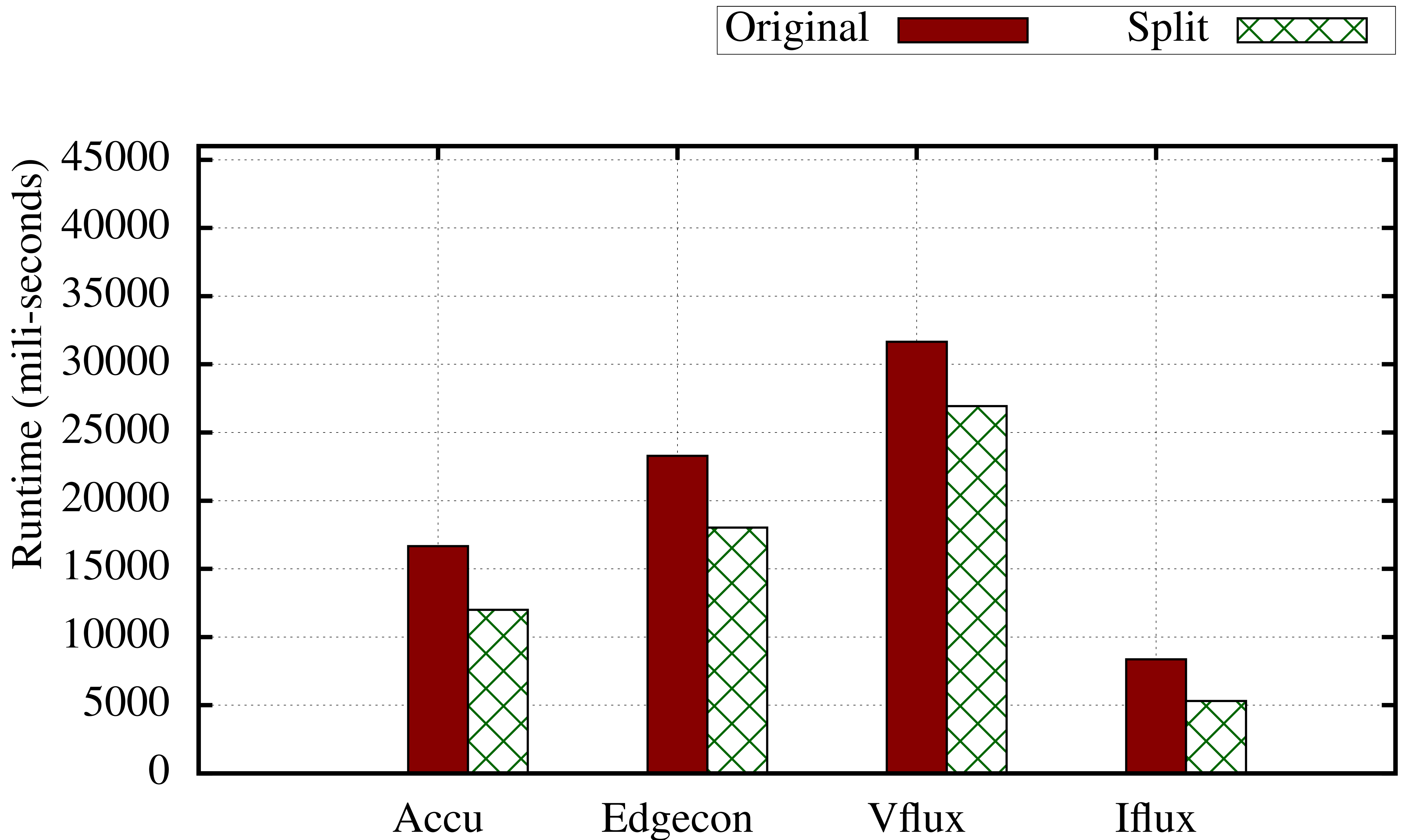
- Hydra is an unstructured mesh production CFD application used at Rolls-Royce for simulating turbo-machinery of aircraft engines
- Used for the design of turbomachinery
  - Key CFD production code
  - Steady and unsteady flow
  - Reynolds Averaged Navier-Stokes
- **In development for >15 years**
  - **Fortran 77**
  - **> 50k lines of code**
  - **> 300 computational op_par_loops**







Performance results - HYDRA



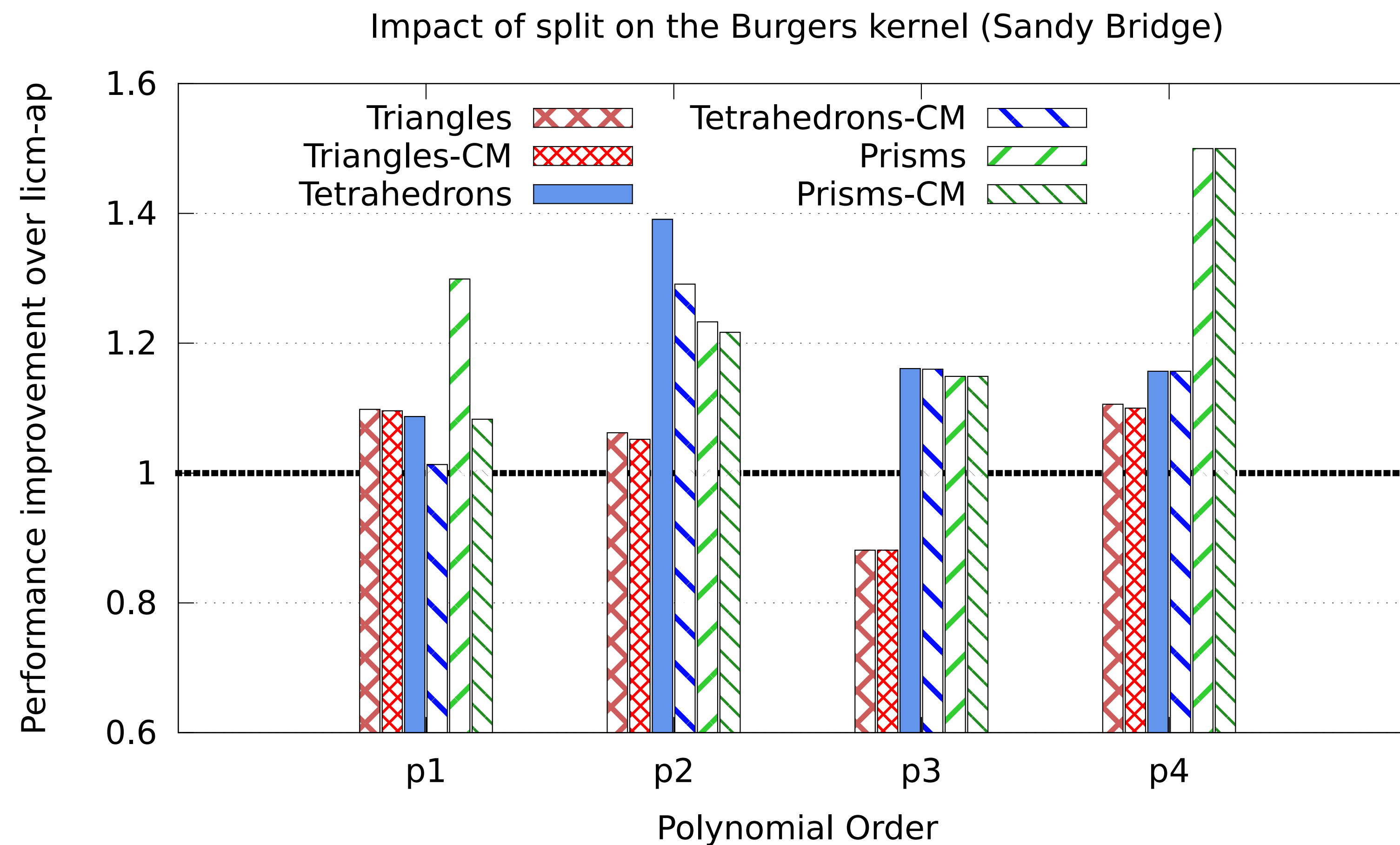
GPU: NVIDIA Tesla C2070

Conclusions and Future Work

- *"Effective Resource-Driven Loop Splitting for Large Unstructured Mesh Applications on GPUs" — "Resource-driven" is notably important*
- *Methodology:*
  1. *prototypes to assess loop splitting, at different levels*
  2. *implementation in the OP2 compiler (on going work)*
  3. *implementation in Firedrake (expression splitting)*
- *This study: driven by real computations' memory requirement*
- *Simple but effective transformations, up to ~35% speedup over the original op_par_loop implementation.*

Spare slides

Towards generalized loop splitting (finite element)



- Problem:
  - **Burgers equation**
  - polynomial orders 1 to 4
  - speedup over original (i.e., non-transformed) code
  - expression restructured (split) through COFFEE for improving register reuse
  - heuristic: minimize repeated accesses to same arrays in the split loops
- Setup:
  - Single core of an Intel Sandy Bridge (I7-2600 CPU @ 3.40GHz)
  - Intel compiler (version 13.1, -O3, -xAVX, -ip, -xHost)

Generalized loop splitting

- Remember the title of the talk:
  "*Effective Resource-Driven Loop Splitting for Large Unstructured Mesh Applications on GPUs*"
- *Resource can be shared memory in a GPU…*
- *…or even registers in a CPU!*
- *Future work: (after suitable optimization for FLOPS,) create an expression graph and find splitting points that improve memory usage*